



IDL Reference Guide

RSI
Research Systems Inc.

IDL Version 6.1
July 2004 Edition
Copyright © Research Systems, Inc.
All Rights Reserved

Restricted Rights Notice

The IDL[®], ION Script[™], and ION Java[™] software programs and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. Research Systems, Inc., reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

Research Systems, Inc. makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

Research Systems, Inc. shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the IDL or ION software packages or their documentation.

Permission to Reproduce this Manual

If you are a licensed user of this product, Research Systems, Inc. grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

Acknowledgments

IDL[®] is a registered trademark and ION[™], ION Script[™], ION Java[™], are trademarks of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein.

Numerical Recipes[™] is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2[™] is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright 1988-2001 The Board of Trustees of the University of Illinois
All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998-2002 by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library
Copyright © 2002 National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1999 University Corporation for Atmospheric Research/Unidata

HDF EOS Library
Copyright © 1996 Hughes and Applied Research Corporation

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by DataDirect Technologies, 1991-2003.

Portions of this software were developed using Unisearch's Kakadu software, for which Kodak has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

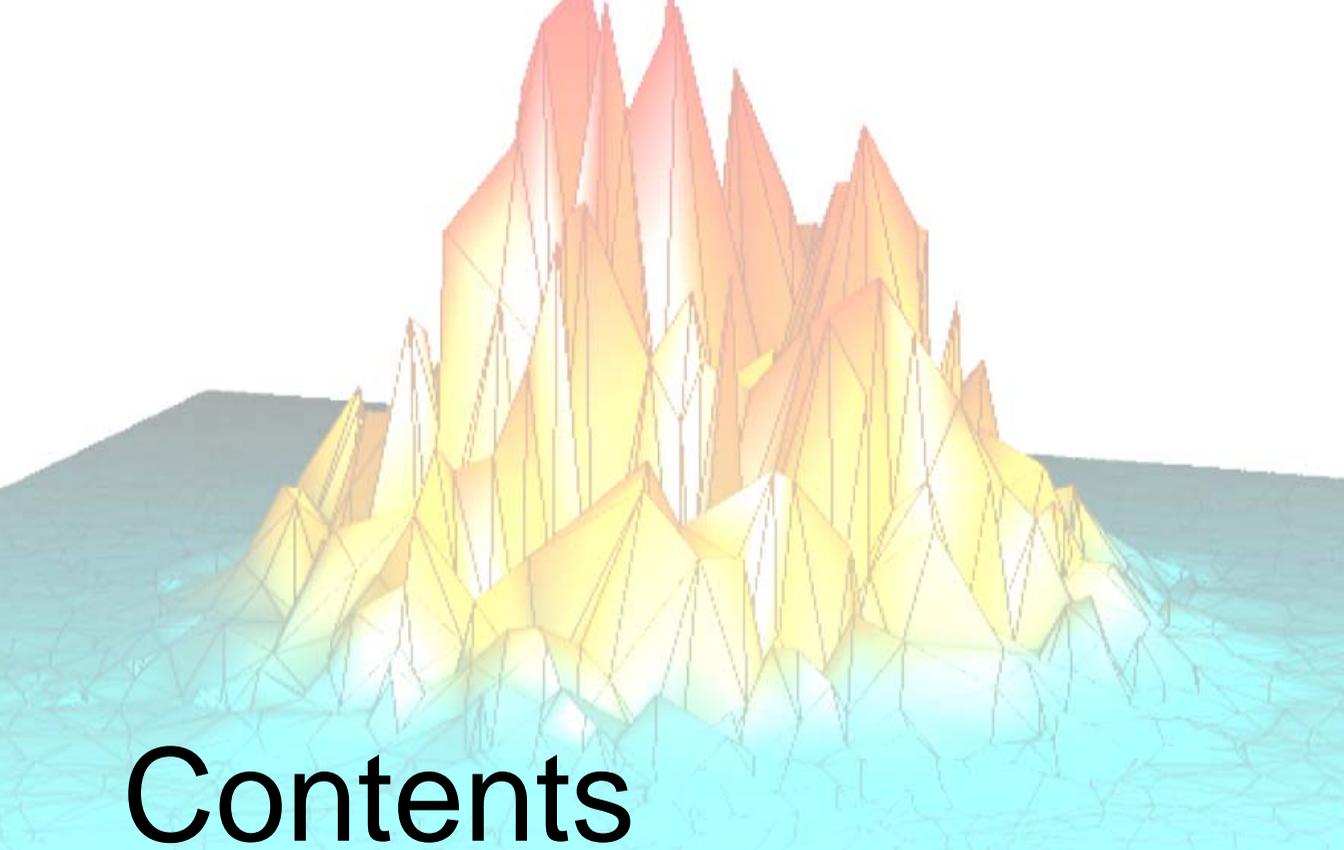
Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this software are copyrighted by Merge Technologies Incorporated.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents

Chapter 1:	
Overview of IDL Syntax	57
IDL Syntax	58
Elements of Syntax	59
Procedures	60
Functions	60
Arguments	61
Keywords	61

Part I: IDL Command Reference

Chapter 2:	
Dot Commands	66
.COMPILE	67
.CONTINUE	68
.EDIT	69

.FULL_RESET_SESSION	70
.GO	71
.OUT	72
.RESET_SESSION	73
.RETURN	75
.RNEW	76
.RUN	78
.SKIP	80
.STEP	82
.STEPOVER	83
.TRACE	84

Chapter 3: Procedures and Functions 85

A_CORRELATE	86
ABS	88
ACOS	90
ADAPT_HIST_EQUAL	92
ALOG	95
ALOG10	97
AMOEBA	99
ANNOTATE	103
APP_USER_DIR	105
APP_USER_DIR_QUERY	114
ARG_PRESENT	119
ARRAY_EQUAL	121
ARRAY_INDICES	123
ARROW	126
ASCII_TEMPLATE	128
ASIN	132
ASSOC	134
ATAN	137
AXIS	141
BAR_PLOT	145
BEGIN...END	149
BESELI	152

BESELJ	155
BESELK	160
BESELY	163
BETA	166
BILINEAR	168
BIN_DATE	171
BINARY_TEMPLATE	173
BINDGEN	178
BINOMIAL	180
BLAS_AXPY	182
BLK_CON	186
BOX_CURSOR	188
BREAK	190
BREAKPOINT	191
BROYDEN	194
BYTARR	197
BYTE	199
BYTEORDER	201
BYTSCL	206
C_CORRELATE	209
CALDAT	212
CALENDAR	216
CALL_EXTERNAL	217
CALL_FUNCTION	228
CALL_METHOD	230
CALL_PROCEDURE	232
CASE	234
CATCH	236
CD	239
CDF Routines	241
CEIL	242
CHEBYSHEV	244
CHECK_MATH	245
CHISQR_CVF	251
CHISQR_PDF	253
CHOLDC	255

CHOLSOL	257
CINDGEN	259
CIR_3PNT	261
CLOSE	263
CLUST_WTS	265
CLUSTER	267
CLUSTER_TREE	270
COLOR_CONVERT	275
COLOR_QUAN	277
COLORMAP_APPLICABLE	281
COMFIT	283
COMMON	286
COMPILE_OPT	287
COMPLEX	292
COMPLEXARR	295
COMPLEXROUND	297
COMPUTE_MESH_NORMALS	299
COND	300
CONGRID	302
CONJ	305
CONSTRAINED_MIN	307
CONTINUE	314
CONTOUR	315
CONVERT_COORD	328
CONVOL	331
COORD2TO3	336
COPY_LUN	338
CORRELATE	341
COS	343
COSH	345
CPU	347
CRAMER	352
CREATE_CURSOR	354
CREATE_STRUCT	356
CREATE_VIEW	359
CROSSP	363

CRVLENGTH	364
CT_LUMINANCE	366
CTI_TEST	368
CURSOR	371
CURVEFIT	374
CV_COORD	379
CVTTOBM	382
CW_ANIMATE	384
CW_ANIMATE_GETP	390
CW_ANIMATE_LOAD	392
CW_ANIMATE_RUN	394
CW_ARCBALL	396
CW_BGROUPE	402
CW_CLR_INDEX	409
CW_COLORSEL	413
CW_DEFROI	416
CW_FIELD	421
CW_FILESEL	427
CW_FORM	433
CW_FSLIDER	442
CW_LIGHT_EDITOR	448
CW_LIGHT_EDITOR_GET	453
CW_LIGHT_EDITOR_SET	456
CW_ORIENT	458
CW_PALETTE_EDITOR	462
CW_PALETTE_EDITOR_GET	470
CW_PALETTE_EDITOR_SET	471
CW_PDMENU	472
CW_RGBSLIDER	481
CW_TMPL	485
CW_ZOOM	487
DBLARR	493
DCINDGEN	495
DCOMPLEX	497
DCOMPLEXARR	500
DEFINE_KEY	502

DEFINE_MSGBLK	511
DEFINE_MSGBLK_FROM_FILE	514
DEFROI	519
DEFSYSV	521
DELVAR	523
DENDRO_PLOT	524
DENDROGRAM	529
DERIV	532
DERIVSIG	533
DETERM	535
DEVICE	537
DFPMIN	541
DIAG_MATRIX	545
DIALOG_MESSAGE	547
DIALOG_PICKFILE	550
DIALOG_PRINTERSETUP	557
DIALOG_PRINTJOB	559
DIALOG_READ_IMAGE	561
DIALOG_WRITE_IMAGE	564
DIGITAL_FILTER	567
DILATE	569
DINDGEN	575
DISSOLVE	577
DIST	579
DISTANCE_MEASURE	581
DLM_LOAD	586
DLM_REGISTER	587
DOC_LIBRARY	588
DOUBLE	590
DRAW_ROI	592
EFONT	594
EIGENQL	596
EIGENVEC	599
ELMHES	602
EMPTY	604
ENABLE_SYSRTN	605

EOF	607
EOS_* Routines	609
ERASE	610
ERF	612
ERFC	614
ERFCX	616
ERODE	618
ERRPLOT	624
EXECUTE	626
EXIT	629
EXP	631
EXPAND	633
EXPAND_PATH	635
EXPINT	640
EXTRAC	643
EXTRACT_SLICE	646
F_CVF	650
F_PDF	652
FACTORIAL	654
FFT	656
FILE_BASENAME	661
FILE_CHMOD	664
FILE_COPY	669
FILE_DELETE	673
FILE_DIRNAME	676
FILE_EXPAND_PATH	679
FILE_INFO	681
FILE_LINES	685
FILE_LINK	688
FILE_MKDIR	691
FILE_MOVE	692
FILE_READLINK	695
FILE_SAME	697
FILE_SEARCH	700
FILE_TEST	717
FILE_WHICH	721

FILEPATH	723
FINDGEN	725
FINITE	727
FIX	731
FLICK	734
FLOAT	735
FLOOR	737
FLOW3	739
FLTARR	741
FLUSH	743
FOR	744
FORMAT_AXIS_VALUES	745
FORWARD_FUNCTION	747
FREE_LUN	748
FSTAT	750
FULSTR	753
FUNCT	755
FUNCTION	757
FV_TEST	758
FX_ROOT	760
FZ_ROOTS	763
GAMMA	766
GAMMA_CT	768
GAUSS_CVF	769
GAUSS_PDF	771
GAUSS2DFIT	773
GAUSSFIT	777
GAUSSINT	782
GET_DRIVE_LIST	784
GET_KBRD	786
GET_LUN	788
GET_SCREEN_SIZE	790
GETENV	792
GOTO	795
GRID_INPUT	796
GRID_TPS	801

GRID3	805
GRIDDATA	808
GS_ITER	833
H_EQ_CT	836
H_EQ_INT	837
H5_* Routines	839
H5_BROWSER	840
HANNING	843
HDF_* Routines	845
HDF_BROWSER	846
HDF_READ	850
HEAP_FREE	854
HEAP_GC	857
HELP	859
HILBERT	867
HIST_2D	869
HIST_EQUAL	871
HISTOGRAM	874
HLS	880
HOUGH	882
HQR	890
HSV	892
IBETA	894
ICONTOUR	898
IDENTITY	923
IDL_Container Object Class	925
IDL_VALIDNAME	926
IDLan* Object Class	928
IDLcom* Object Class	929
IDLff* Object Class	930
IDLgr* Object Classes	931
IDLit* Object Classes	932
IDLITSYS_CREATETOOL	933
IF...THEN...ELSE	937
IGAMMA	939
IIMAGE	942

IMAGE_CONT	958
IMAGE_STATISTICS	960
IMAGINARY	963
IMAP	965
INDGEN	990
INT_2D	993
INT_3D	997
INT_TABULATED	1000
INTARR	1002
INTERPOL	1004
INTERPOLATE	1007
INTERVAL_VOLUME	1011
INVERT	1016
IOCTL	1018
ILOT	1021
ISHFT	1041
ISOCONTOUR	1043
ISOSURFACE	1048
ISURFACE	1052
ITCURRENT	1073
ITDELETE	1075
ITGETCURRENT	1077
ITREGISTER	1079
ITRESET	1083
ITRESOLVE	1085
IVOLUME	1087
JOURNAL	1110
JULDAY	1112
KEYWORD_SET	1115
KRIG2D	1117
KURTOSIS	1122
KW_TEST	1124
L64INDGEN	1127
LA_CHOLDC	1129
LA_CHOLMPROVE	1132
LA_CHOLSOL	1136

LA_DETERM	1139
LA_EIGENPROBLEM	1141
LA_EIGENQL	1147
LA_EIGENVEC	1153
LA_ELMHES	1157
LA_GM_LINEAR_MODEL	1160
LA_HQR	1163
LA_INVERT	1166
LA_LEAST_SQUARE_EQUALITY	1168
LA_LEAST_SQUARES	1171
LA_LINEAR_EQUATION	1175
LA_LUDC	1178
LA_LUMPROVE	1181
LA_LUSOL	1184
LA_SVD	1187
LA_TRIDC	1191
LA_TRIMPROVE	1195
LA_TRIQL	1199
LA_TRIRED	1202
LA_TRISOL	1204
LABEL_DATE	1207
LABEL_REGION	1211
LADFIT	1214
LAGUERRE	1217
LEEFILT	1219
LEGENDRE	1221
LINBCG	1224
LINDGEN	1227
LINFIT	1229
LINKIMAGE	1232
LL_ARC_DISTANCE	1236
LMFIT	1238
LMGR	1243
LNGAMMA	1246
LNP_TEST	1248
LOADCT	1251

LOCALE_GET	1253
LOGICAL_AND	1254
LOGICAL_OR	1256
LOGICAL_TRUE	1258
LON64ARR	1260
LONARR	1262
LONG	1264
LONG64	1266
LSODE	1268
LU_COMPLEX	1273
LUDC	1275
LUMPROVE	1277
LUSOL	1280
M_CORRELATE	1283
MACHAR	1286
MAKE_ARRAY	1288
MAKE_DLL	1292
MAP_2POINTS	1298
MAP_CONTINENTS	1302
MAP_GRID	1307
MAP_IMAGE	1313
MAP_PATCH	1318
MAP_PROJ_FORWARD	1322
MAP_PROJ_IMAGE	1327
MAP_PROJ_INFO	1332
MAP_PROJ_INIT	1335
MAP_PROJ_INVERSE	1351
MAP_SET	1353
MATRIX_MULTIPLY	1365
MATRIX_POWER	1368
MAX	1370
MD_TEST	1374
MEAN	1376
MEANABSDEV	1378
MEDIAN	1380
MEMORY	1383

MESH_CLIP	1387
MESH_DECIMATE	1392
MESH_ISSOLID	1399
MESH_MERGE	1400
MESH_NUMTRIANGLES	1405
MESH_OBJ	1406
MESH_SMOOTH	1413
MESH_SURFACEAREA	1418
MESH_VALIDATE	1420
MESH_VOLUME	1422
MESSAGE	1424
MIN	1430
MIN_CURVE_SURF	1433
MK_HTML_HELP	1438
MODIFYCT	1441
MOMENT	1443
MORPH_CLOSE	1446
MORPH_DISTANCE	1449
MORPH_GRADIENT	1452
MORPH_HITORMISS	1455
MORPH_OPEN	1458
MORPH_THIN	1461
MORPH_TOPHAT	1463
MPEG_CLOSE	1466
MPEG_OPEN	1467
MPEG_PUT	1471
MPEG_SAVE	1473
MULTI	1474
N_ELEMENTS	1475
N_PARAMS	1477
N_TAGS	1478
NCDF_* Routines	1480
NEWTON	1481
NORM	1484
OBJ_CLASS	1487
OBJ_DESTROY	1489

OBJ_ISA	1490
OBJ_NEW	1491
OBJ_VALID	1493
OBJARR	1495
ON_ERROR	1497
ON_IOERROR	1498
ONLINE_HELP	1500
ONLINE_HELP_PDF_INDEX	1506
OPEN	1511
OPLOT	1520
OPLOTERR	1523
P_CORRELATE	1525
PARTICLE_TRACE	1527
PATH_CACHE	1530
PATH_SEP	1537
PCOMP	1538
PLOT	1543
PLOT_3DBOX	1547
PLOT_FIELD	1551
PLOTERR	1553
PLOTS	1555
PNT_LINE	1558
POINT_LUN	1560
POLAR_CONTOUR	1562
POLAR_SURFACE	1564
POLY	1566
POLY_2D	1567
POLY_AREA	1571
POLY_FIT	1573
POLYFILL	1577
POLYFILLV	1582
POLYSHADE	1584
POLYWARP	1588
POPD	1591
POWELL	1592
PRIMES	1595

PRINT/PRINTF	1596
PRINTD	1599
PRO	1600
PRODUCT	1602
PROFILE	1606
PROFILER	1609
PROFILES	1612
PROJECT_VOL	1614
PS_SHOW_FONTS	1618
PSAFM	1619
PSEUDO	1620
PTR_FREE	1622
PTR_NEW	1623
PTR_VALID	1625
PTRARR	1627
PUSHD	1629
QGRID3	1630
QHULL	1635
QROMB	1640
QROMO	1645
QSIMP	1648
QUERY_* Routines	1651
QUERY_BMP	1655
QUERY_DICOM	1656
QUERY_GIF	1658
QUERY_IMAGE	1660
QUERY_JPEG	1664
QUERY_JPEG2000	1665
QUERY_MRSID	1667
QUERY_PICT	1670
QUERY_PNG	1671
QUERY_PPM	1673
QUERY_SRF	1675
QUERY_TIFF	1676
QUERY_WAV	1678
R_CORRELATE	1679

R_TEST	1682
RADON	1684
RANDOMN	1693
RANDOMU	1698
RANKS	1703
RDPIX	1705
READ/READF	1706
READ_ASCII	1709
READ_BINARY	1712
READ_BMP	1714
READ_DICOM	1717
READ_GIF	1719
READ_IMAGE	1721
READ_INTERFILE	1723
READ_JPEG	1725
READ_JPEG2000	1729
READ_MRSID	1733
READ_PICT	1736
READ_PNG	1738
READ_PPM	1741
READ_SPR	1743
READ_SRF	1744
READ_SYLK	1746
READ_TIFF	1750
READ_WAV	1759
READ_WAVE	1760
READ_X11_BITMAP	1762
READ_XWD	1764
READS	1766
READU	1768
REAL_PART	1770
REBIN	1771
RECALL_COMMANDS	1774
RECON3	1775
REDUCE_COLORS	1781
REFORM	1783

REGION_GROW	1785
REGISTER_CURSOR	1788
REGRESS	1790
REPEAT...UNTIL	1794
REPLICATE	1795
REPLICATE_INPLACE	1797
RESOLVE_ALL	1799
RESOLVE_ROUTINE	1802
RESTORE	1804
RETALL	1807
RETURN	1808
REVERSE	1810
RK4	1812
ROBERTS	1815
ROT	1817
ROTATE	1820
ROUND	1823
ROUTINE_INFO	1825
RS_TEST	1828
S_TEST	1831
SAVE	1833
SAVGOL	1837
SCALE3	1842
SCALE3D	1844
SCOPE_LEVEL	1845
SCOPE_VARFETCH	1846
SCOPE_VARNAME	1852
SEARCH2D	1858
SEARCH3D	1861
SET_PLOT	1865
SET_SHADING	1867
SETENV	1869
SETUP_KEYS	1870
SFIT	1873
SHADE_SURF	1876
SHADE_SURF_IRR	1881

SHADE_VOLUME	1884
SHIFT	1887
SHMDEBUG	1889
SHMMAP	1891
SHMUNMAP	1906
SHMVAR	1908
SHOW3	1912
SHOWFONT	1914
SIMPLEX	1916
SIN	1921
SINDGEN	1923
SINH	1924
SIZE	1926
SKEWNESS	1931
SKIP_LUN	1933
SLICER3	1936
SLIDE_IMAGE	1956
SMOOTH	1960
SOBEL	1963
SOCKET	1965
SORT	1970
SPAWN	1972
SPH_4PNT	1980
SPH_SCAT	1982
SPHER_HARM	1985
SPL_INIT	1988
SPL_INTERP	1990
SPLINE	1992
SPLINE_P	1994
SPRSAB	1997
SPRSAX	2000
SPRSIN	2002
SPRSTP	2005
SQRT	2006
STANDARDIZE	2008
STDDEV	2010

STOP	2012
STRARR	2013
STRCMP	2014
STRCOMPRESS	2016
STREAMLINE	2018
STREGEX	2020
STRETCH	2024
STRING	2026
STRJOIN	2029
STRLEN	2031
STRLOWCASE	2032
STRMATCH	2034
STRMESSAGE	2037
STRMID	2039
STRPOS	2041
STRPUT	2044
STRSPLIT	2046
STRTRIM	2051
STRUCT_ASSIGN	2053
STRUCT_HIDE	2055
STRUPCASE	2057
SURFACE	2058
SURFR	2064
SVDC	2065
SVDFIT	2068
SVSOL	2074
SWAP_ENDIAN	2076
SWAP_ENDIAN_INPLACE	2078
SWITCH	2080
SYSTIME	2082
T_CVF	2085
T_PDF	2087
T3D	2089
TAG_NAMES	2092
TAN	2094
TANH	2096

TEK_COLOR	2098
TEMPORARY	2099
TETRA_CLIP	2101
TETRA_SURFACE	2103
TETRA_VOLUME	2104
THIN	2106
THREED	2108
TIME_TEST2	2110
TIMEGEN	2111
TM_TEST	2116
TOTAL	2118
TRACE	2123
TrackBall Object	2125
TRANSPOSE	2126
TRI_SURF	2128
TRIANGULATE	2132
TRIGRID	2136
TRIQL	2146
TRIRED	2149
TRISOL	2151
TRUNCATE_LUN	2154
TS_COEF	2156
TS_DIFF	2158
TS_FCAST	2160
TS_SMOOTH	2162
TV	2165
TVCRS	2169
TVLCT	2171
TVRD	2174
TVSCL	2178
UINDGEN	2181
UINT	2183
UINTARR	2185
UL64INDGEN	2187
ULINDGEN	2189
ULON64ARR	2191

ULONARR	2193
ULONG	2195
ULONG64	2197
UNIQ	2199
UNSHARP_MASK	2201
USERSYM	2205
VALUE_LOCATE	2207
VARIANCE	2209
VECTOR_FIELD	2211
VEL	2213
VELOVECT	2215
VERT_T3D	2218
VOIGT	2220
VORONOI	2223
VOXEL_PROJ	2225
WAIT	2231
WARP_TRI	2232
WATERSHED	2234
Wavelet Toolkit	2237
WDELETE	2238
WF_DRAW	2239
WHERE	2242
WHILE...DO	2246
WIDGET_ACTIVEX	2247
WIDGET_BASE	2254
WIDGET_BUTTON	2280
WIDGET_COMBOBOX	2295
WIDGET_CONTROL	2305
WIDGET_DISPLAYCONTEXTMENU	2351
WIDGET_DRAW	2353
WIDGET_DROPLIST	2370
WIDGET_EVENT	2379
WIDGET_INFO	2383
WIDGET_LABEL	2407
WIDGET_LIST	2414
WIDGET_PROPERTY SHEET	2424

WIDGET_SLIDER	2441
WIDGET_TAB	2451
WIDGET_TABLE	2461
WIDGET_TEXT	2480
WIDGET_TREE	2492
WINDOW	2503
WRITE_BMP	2507
WRITE_GIF	2510
WRITE_IMAGE	2512
WRITE_JPEG	2514
WRITE_JPEG2000	2517
WRITE_NRIF	2519
WRITE_PICT	2521
WRITE_PNG	2523
WRITE_PPM	2526
WRITE_SPR	2528
WRITE_SRF	2530
WRITE_SYLK	2532
WRITE_TIFF	2534
WRITE_WAV	2545
WRITE_WAVE	2546
WRITEU	2548
WSET	2550
WSHOW	2552
WTN	2554
WV_* Routines	2558
XBM_EDIT	2559
XDISPLAYFILE	2561
XDXF	2564
XFONT	2568
XINTERANIMATE	2570
XLOADCT	2578
XMANAGER	2581
XMNG_TMPL	2589
XMTOOL	2591
XOBJVIEW	2593

XOBJVIEW_ROTATE	2603
XOBJVIEW_WRITE_IMAGE	2605
XPALETTE	2607
XPCOLOR	2611
XPLOT3D	2612
XREGISTERED	2619
XROI	2621
XSQ_TEST	2637
XSURFACE	2640
XVAREEDIT	2642
XVOLUME	2644
XVOLUME_ROTATE	2650
XVOLUME_WRITE_IMAGE	2653
XYOUTS	2655
ZOOM	2659
ZOOM_24	2661

Part II: Object Class and Method Reference

Chapter 4:	
IDL Object Class Overview	2665
Using the Class Reference	2666
Syntax	2666
Arguments	2667
Creating Objects from the Class Library	2668
Object Properties	2669
Properties and the Property Sheet Interface	2669
Setting Properties at Initialization	2670
Setting Properties of Existing Objects	2670
Retrieving Property Settings	2670
About Object Property Descriptions	2671
Registered Properties	2673
Registering a Property	2673
Registering All Available Properties	2673
Registered Property Data Types	2674
Undocumented Object Classes	2677

Chapter 5:		
Analysis Object Classes	2679
IDLanROI	2680
IDLanROI Properties	2682
IDLanROI::AppendData	2686
IDLanROI::Cleanup	2688
IDLanROI::ComputeGeometry	2689
IDLanROI::ComputeMask	2691
IDLanROI::ContainsPoints	2694
IDLanROI::GetProperty	2696
IDLanROI::Init	2697
IDLanROI::RemoveData	2699
IDLanROI::ReplaceData	2701
IDLanROI::Rotate	2704
IDLanROI::Scale	2705
IDLanROI::SetProperty	2706
IDLanROI::Translate	2707
IDLanROIGroup	2708
IDLanROIGroup Properties	2710
IDLanROIGroup::Add	2712
IDLanROIGroup::Cleanup	2713
IDLanROIGroup::ComputeMask	2714
IDLanROIGroup::ComputeMesh	2717
IDLanROIGroup::ContainsPoints	2719
IDLanROIGroup::GetProperty	2721
IDLanROIGroup::Init	2722
IDLanROIGroup::Rotate	2723
IDLanROIGroup::Scale	2724
IDLanROIGroup::Translate	2725
Chapter 6:		
File Format Object Classes	2727
IDLffDICOM	2728
IDL DICOM v3.0 Conformance Summary	2730
IDLffDICOM Properties	2734
IDLffDICOM::Cleanup	2735

IDLffDICOM::DumpElements	2736
IDLffDICOM::GetChildren	2737
IDLffDICOM::GetDescription	2739
IDLffDICOM::GetElement	2741
IDLffDICOM::GetGroup	2743
IDLffDICOM::GetLength	2745
IDLffDICOM::GetParent	2747
IDLffDICOM::GetPreamble	2749
IDLffDICOM::GetReference	2750
IDLffDICOM::GetValue	2752
IDLffDICOM::GetVR	2755
IDLffDICOM::Init	2757
IDLffDICOM::Read	2759
IDLffDICOM::Reset	2760
IDLffDXF	2761
IDLffDXF Properties	2763
IDLffDXF::Cleanup	2764
IDLffDXF::GetContents	2765
IDLffDXF::GetEntity	2768
Fields Common to All Structures	2769
Structure Formats	2770
IDLffDXF::GetPalette	2781
IDLffDXF::Init	2782
IDLffDXF::PutEntity	2783
IDLffDXF::Read	2784
IDLffDXF::RemoveEntity	2785
IDLffDXF::Reset	2786
IDLffDXF::SetPalette	2787
IDLffDXF::Write	2788
IDLffJPEG2000	2790
Overview of IDL and JPEG2000	2792
IDLffJPEG2000 Properties	2795
IDLffJPEG2000::Cleanup	2806
IDLffJPEG2000::GetData	2807
IDLffJPEG2000::GetProperty	2811
IDLffJPEG2000::GetTileProperty	2812

IDLffJPEG2000::Init	2814
IDLffJPEG2000::SetData	2816
IDLffJPEG2000::SetProperty	2820
IDLffLangCat	2821
IDLffLangCat Properties	2823
IDLffLangCat::AppendCatalog	2827
IDLffLangCat::Cleanup	2829
IDLffLangCat::GetProperty	2830
IDLffLangCat::Init	2831
IDLffLangCat::Query	2833
IDLffLangCat::SetProperty	2834
IDLffMrSID	2835
IDLffMrSID Properties	2837
IDLffMrSID::Cleanup	2838
IDLffMrSID::GetDimsAtLevel	2839
IDLffMrSID::GetImageData	2841
IDLffMrSID::GetProperty	2844
IDLffMrSID::Init	2847
IDLffShape	2849
Overview of ESRI Shapefiles	2851
Accessing Shapefiles	2856
Creating New Shapefiles	2858
Updating Existing Shapefiles	2859
IDLffShape Properties	2861
IDLffShape::AddAttribute	2865
IDLffShape::Cleanup	2868
IDLffShape::Close	2869
IDLffShape::DestroyEntity	2870
IDLffShape::GetAttributes	2872
IDLffShape::GetEntity	2874
IDLffShape::GetProperty	2876
IDLffShape::Init	2878
IDLffShape::Open	2880
IDLffShape::PutEntity	2882
IDLffShape::SetAttributes	2884
IDLffXMLDOM Classes	2887

IDLffXMLDOMAttr	2888
IDLffXMLDOMAttr Properties	2890
IDLffXMLDOMAttr::Cleanup	2891
IDLffXMLDOMAttr::GetName	2892
IDLffXMLDOMAttr::GetSpecified	2893
IDLffXMLDOMAttr::GetValue	2894
IDLffXMLDOMAttr::Init	2895
IDLffXMLDOMAttr::SetValue	2896
IDLffXMLDOMCDATASection	2897
IDLffXMLDOMCDATASection Properties	2898
IDLffXMLDOMCDATASection::Cleanup	2899
IDLffXMLDOMCDATASection::Init	2900
IDLffXMLDOMCharacterData	2901
IDLffXMLDOMCharacterData Properties	2903
IDLffXMLDOMCharacterData::AppendData	2904
IDLffXMLDOMCharacterData::Cleanup	2905
IDLffXMLDOMCharacterData::DeleteData	2906
IDLffXMLDOMCharacterData::GetData	2907
IDLffXMLDOMCharacterData::GetLength	2908
IDLffXMLDOMCharacterData::Init	2909
IDLffXMLDOMCharacterData::InsertData	2910
IDLffXMLDOMCharacterData::ReplaceData	2911
IDLffXMLDOMCharacterData::SetData	2912
IDLffXMLDOMCharacterData::SubstringData	2913
IDLffXMLDOMComment	2914
IDLffXMLDOMComment Properties	2915
IDLffXMLDOMComment::Cleanup	2916
IDLffXMLDOMComment::Init	2917
IDLffXMLDOMDocument	2918
IDLffXMLDOMDocument Properties	2920
IDLffXMLDOMDocument::Cleanup	2921
IDLffXMLDOMDocument::CreateAttribute	2922
IDLffXMLDOMDocument::CreateCDATASection	2923
IDLffXMLDOMDocument::CreateComment	2924
IDLffXMLDOMDocument::CreateDocumentFragment	2925
IDLffXMLDOMDocument::CreateElement	2926

IDLffXMLDOMDocument::CreateEntityReference	2927
IDLffXMLDOMDocument::CreateProcessingInstruction	2928
IDLffXMLDOMDocument::CreateTextNode	2929
IDLffXMLDOMDocument::GetDoctype	2930
IDLffXMLDOMDocument::GetDocumentElement	2931
IDLffXMLDOMDocument::GetElementsByTagName	2932
IDLffXMLDOMDocument::Init	2933
IDLffXMLDOMDocument::Load	2936
IDLffXMLDOMDocument::Save	2939
IDLffXMLDOMDocumentFragment	2940
IDLffXMLDOMDocumentFragment Properties	2942
IDLffXMLDOMDocumentFragment::Cleanup	2943
IDLffXMLDOMDocumentFragment::Init	2944
IDLffXMLDOMDocumentType	2945
IDLffXMLDOMDocumentType Properties	2946
IDLffXMLDOMDocumentType::Cleanup	2947
IDLffXMLDOMDocumentType::GetEntities	2948
IDLffXMLDOMDocumentType::GetName	2949
IDLffXMLDOMDocumentType::GetNotations	2950
IDLffXMLDOMDocumentType::Init	2951
IDLffXMLDOMEElement	2952
IDLffXMLDOMEElement Properties	2954
IDLffXMLDOMEElement::Cleanup	2955
IDLffXMLDOMEElement::GetAttribute	2956
IDLffXMLDOMEElement::GetAttributeNode	2957
IDLffXMLDOMEElement::GetElementsByTagName	2958
IDLffXMLDOMEElement::GetTagName	2959
IDLffXMLDOMEElement::Init	2960
IDLffXMLDOMEElement::RemoveAttribute	2961
IDLffXMLDOMEElement::RemoveAttributeNode	2962
IDLffXMLDOMEElement::SetAttribute	2963
IDLffXMLDOMEElement::SetAttributeNode	2964
IDLffXMLDOMEntity	2965
IDLffXMLDOMEntity Properties	2966
IDLffXMLDOMEntity::Cleanup	2967
IDLffXMLDOMEntity::GetNotationName	2968

IDLffXMLDOMEntity::GetPublicId	2969
IDLffXMLDOMEntity::GetSystemId	2970
IDLffXMLDOMEntity::Init	2971
IDLffXMLDOMEntityReference	2972
IDLffXMLDOMEntityReference Properties	2973
IDLffXMLDOMEntityReference::Cleanup	2974
IDLffXMLDOMEntityReference::Init	2975
IDLffXMLDOMNamedNodeMap	2976
IDLffXMLDOMNamedNodeMap Properties	2978
IDLffXMLDOMNamedNodeMap::Cleanup	2979
IDLffXMLDOMNamedNodeMap::GetLength	2980
IDLffXMLDOMNamedNodeMap::GetNamedItem	2981
IDLffXMLDOMNamedNodeMap::Init	2982
IDLffXMLDOMNamedNodeMap::Item	2983
IDLffXMLDOMNamedNodeMap::RemoveNamedItem	2984
IDLffXMLDOMNamedNodeMap::SetNamedItem	2985
IDLffXMLDOMNode	2986
IDLffXMLDOMNode Properties	2988
IDLffXMLDOMNode::AppendChild	2989
IDLffXMLDOMNode::Cleanup	2990
IDLffXMLDOMNode::CloneNode	2991
IDLffXMLDOMNode::GetAttributes	2992
IDLffXMLDOMNode::GetChildNodes	2993
IDLffXMLDOMNode::GetFirstChild	2994
IDLffXMLDOMNode::GetLastChild	2995
IDLffXMLDOMNode::GetNextSibling	2996
IDLffXMLDOMNode::GetNodeName	2997
IDLffXMLDOMNode::GetNodeType	2999
IDLffXMLDOMNode::GetNodeValue	3001
IDLffXMLDOMNode::GetOwnerDocument	3003
IDLffXMLDOMNode::GetParentNode	3004
IDLffXMLDOMNode::GetPreviousSibling	3005
IDLffXMLDOMNode::HasChildNodes	3006
IDLffXMLDOMNode::Init	3007
IDLffXMLDOMNode::InsertBefore	3008
IDLffXMLDOMNode::RemoveChild	3010

IDLffXMLDOMNode::ReplaceChild	3011
IDLffXMLDOMNode::SetNodeValue	3013
IDLffXMLDOMNodeList	3014
IDLffXMLDOMNodeList Properties	3015
IDLffXMLDOMNodeList::Cleanup	3016
IDLffXMLDOMNodeList::GetLength	3017
IDLffXMLDOMNodeList::Init	3018
IDLffXMLDOMNodeList::Item	3019
IDLffXMLDOMNotation	3020
IDLffXMLDOMNotation Properties	3021
IDLffXMLDOMNotation::Cleanup	3022
IDLffXMLDOMNotation::GetPublicID	3023
IDLffXMLDOMNotation::GetSystemID	3024
IDLffXMLDOMNotation::Init	3025
IDLffXMLDOMProcessingInstruction	3026
IDLffXMLDOMProcessingInstruction Properties	3027
IDLffXMLDOMProcessingInstruction::Cleanup	3028
IDLffXMLDOMProcessingInstruction::GetData	3029
IDLffXMLDOMProcessingInstruction::GetTarget	3030
IDLffXMLDOMProcessingInstruction::Init	3031
IDLffXMLDOMProcessingInstruction::SetData	3032
IDLffXMLDOMText	3033
IDLffXMLDOMText Properties	3034
IDLffXMLDOMText::Cleanup	3035
IDLffXMLDOMText::Init	3036
IDLffXMLDOMText::IsIgnorableWhitespace	3037
IDLffXMLDOMText::SplitText	3038
IDLffXMLSAX	3039
IDLffXMLSAX Properties	3042
IDLffXMLSAX::AttributeDecl	3046
IDLffXMLSAX::Characters	3048
IDLffXMLSAX::Cleanup	3049
IDLffXMLSAX::Comment	3050
IDLffXMLSAX::ElementDecl	3051
IDLffXMLSAX::EndCDATA	3052
IDLffXMLSAX::EndDocument	3053

IDLffXMLSAX::EndDTD	3054
IDLffXMLSAX::EndElement	3055
IDLffXMLSAX::EndEntity	3056
IDLffXMLSAX::EndPrefixMapping	3057
IDLffXMLSAX::Error	3058
IDLffXMLSAX::ExternalEntityDecl	3060
IDLffXMLSAX::FatalError	3061
IDLffXMLSAX::GetProperty	3062
IDLffXMLSAX::IgnorableWhitespace	3063
IDLffXMLSAX::Init	3064
IDLffXMLSAX::InternalEntityDecl	3065
IDLffXMLSAX::NotationDecl	3066
IDLffXMLSAX::ParseFile	3067
IDLffXMLSAX::ProcessingInstruction	3068
IDLffXMLSAX::SetProperty	3069
IDLffXMLSAX::SkippedEntity	3070
IDLffXMLSAX::StartCDATA	3071
IDLffXMLSAX::StartDocument	3072
IDLffXMLSAX::StartDTD	3073
IDLffXMLSAX::StartElement	3074
IDLffXMLSAX::StartEntity	3076
IDLffXMLSAX::StartPrefixmapping	3077
IDLffXMLSAX::StopParsing	3078
IDLffXMLSAX::UnparsedEntityDecl	3079
IDLffXMLSAX::Warning	3080
Chapter 7:	
iTools Object Classes	3082
IDLitCommand	3083
IDLitCommand Properties	3085
IDLitCommand::AddItem	3086
IDLitCommand::Cleanup	3088
IDLitCommand::GetItem	3089
IDLitCommand::GetProperty	3090
IDLitCommand::GetSize	3091
IDLitCommand::Init	3092

IDLitCommand::SetProperty	3094
IDLitCommandSet	3095
IDLitCommandSet Properties	3097
IDLitCommandSet::Cleanup	3098
IDLitCommandSet::GetSize	3099
IDLitCommandSet::Init	3100
IDLitComponent	3101
IDLitComponent Properties	3103
IDLitComponent::Cleanup	3106
IDLitComponent::EditUserDefProperty	3107
IDLitComponent::GetFullIdentifier	3109
IDLitComponent::GetProperty	3110
IDLitComponent::GetPropertyAttribute	3111
IDLitComponent::GetPropertyByIdentifier	3112
IDLitComponent::Init	3113
IDLitComponent::QueryProperty	3115
IDLitComponent::RegisterProperty	3116
IDLitComponent::Restore	3122
IDLitComponent::SetProperty	3123
IDLitComponent::SetPropertyAttribute	3124
IDLitComponent::SetPropertyByIdentifier	3125
IDLitComponent::UpdateComponentVersion	3126
IDLitContainer	3127
IDLitContainer Properties	3129
IDLitContainer::Add	3130
IDLitContainer::AddByIdentifier	3131
IDLitContainer::Cleanup	3132
IDLitContainer::FindIdentifiers	3133
IDLitContainer::Get	3135
IDLitContainer::GetByIdentifier	3137
IDLitContainer::Init	3138
IDLitContainer::Remove	3139
IDLitContainer::RemoveByIdentifier	3140
IDLitData	3141
IDLitData Properties	3143
IDLitData::AddDataObserver	3145

IDLitData::Cleanup	3146
IDLitData::Copy	3147
IDLitData::GetByType	3148
IDLitData::GetData	3149
IDLitData::GetProperty	3150
IDLitData::GetSize	3151
IDLitData::Init	3152
IDLitData::NotifyDataChange	3154
IDLitData::NotifyDataComplete	3155
IDLitData::RemoveDataObserver	3156
IDLitData::SetData	3157
IDLitData::SetProperty	3158
IDLitDataContainer	3159
IDLitDataContainer Properties	3161
IDLitDataContainer::Add	3162
IDLitDataContainer::Cleanup	3163
IDLitDataContainer::GetData	3164
IDLitDataContainer::GetIdentifiers	3165
IDLitDataContainer::GetProperty	3166
IDLitDataContainer::Init	3167
IDLitDataContainer::SetData	3169
IDLitDataContainer::SetProperty	3171
IDLitDataOperation	3172
IDLitDataOperation Properties	3175
IDLitDataOperation::Cleanup	3176
IDLitDataOperation::DoExecuteUI	3177
IDLitDataOperation::Execute	3179
IDLitDataOperation::GetProperty	3181
IDLitDataOperation::Init	3182
IDLitDataOperation::SetProperty	3184
IDLitDataOperation::UndoExecute	3185
IDLitIMessaging	3187
IDLitIMessaging Properties	3189
IDLitIMessaging::AddOnNotifyObserver	3190
IDLitIMessaging::DoOnNotify	3192
IDLitIMessaging::ErrorMessage	3194

IDLitMessaging::GetTool	3196
IDLitMessaging::ProbeStatusMessage	3197
IDLitMessaging::ProgressBar	3198
IDLitMessaging::PromptUserText	3200
IDLitMessaging::PromptUserYesNo	3201
IDLitMessaging::RemoveOnNotifyObserver	3202
IDLitMessaging::SignalError	3203
IDLitMessaging::StatusMessage	3204
IDLitManipulator	3205
IDLitManipulator Properties	3208
IDLitManipulator::Cleanup	3213
IDLitManipulator::CommitUndoValues	3214
IDLitManipulator::GetCursorType	3216
IDLitManipulator::GetProperty	3218
IDLitManipulator::Init	3219
IDLitManipulator::OnKeyboard	3221
IDLitManipulator::OnLoseCurrentManipulator	3223
IDLitManipulator::OnMouseDown	3224
IDLitManipulator::OnMouseMove	3226
IDLitManipulator::OnMouseUp	3228
IDLitManipulator::RecordUndoValues	3229
IDLitManipulator::RegisterCursor	3231
IDLitManipulator::SetCurrentManipulator	3233
IDLitManipulator::SetProperty	3234
IDLitManipulatorContainer	3235
IDLitManipulatorContainer Properties	3237
IDLitManipulatorContainer::Add	3238
IDLitManipulatorContainer::GetCurrent	3239
IDLitManipulatorContainer::GetCurrentManipulator	3240
IDLitManipulatorContainer::GetProperty	3241
IDLitManipulatorContainer::Init	3242
IDLitManipulatorContainer::OnKeyboard	3244
IDLitManipulatorContainer::OnMouseDown	3246
IDLitManipulatorContainer::OnMouseMove	3248
IDLitManipulatorContainer::OnMouseUp	3250
IDLitManipulatorContainer::SetCurrent	3251

IDLitManipulatorContainer::SetCurrentManipulator	3252
IDLitManipulatorContainer::SetProperty	3253
IDLitManipulatorManager	3254
IDLitManipulatorManager Properties	3256
IDLitManipulatorManager::Add	3257
IDLitManipulatorManager::AddManipulatorObserver	3258
IDLitManipulatorManager::GetDefaultManipulator	3259
IDLitManipulatorManager::Init	3260
IDLitManipulatorManager::RemoveManipulatorObserver	3262
IDLitManipulatorVisual	3263
IDLitManipulatorVisual Properties	3265
IDLitManipulatorVisual::Cleanup	3266
IDLitManipulatorVisual::GetProperty	3267
IDLitManipulatorVisual::Init	3268
IDLitManipulatorVisual::SetProperty	3270
IDLitOperation	3271
IDLitOperation Properties	3275
IDLitOperation::Cleanup	3277
IDLitOperation::DoAction	3278
IDLitOperation::GetProperty	3280
IDLitOperation::Init	3281
IDLitOperation::RecordFinalValues	3283
IDLitOperation::RecordInitialValues	3285
IDLitOperation::RedoOperation	3287
IDLitOperation::SetProperty	3289
IDLitOperation::UndoOperation	3290
IDLitParameter	3292
IDLitParameter Properties	3294
IDLitParameter::Cleanup	3295
IDLitParameter::GetParameter	3296
IDLitParameter::GetParameterAttribute	3297
IDLitParameter::GetParameterSet	3299
IDLitParameter::Init	3300
IDLitParameter::OnDataChangeUpdate	3301
IDLitParameter::OnDataDisconnect	3303
IDLitParameter::QueryParameter	3305

IDLitParameter::RegisterParameter	3307
IDLitParameter::SetData	3309
IDLitParameter::SetParameterAttribute	3311
IDLitParameter::SetParameterSet	3313
IDLitParameterSet	3315
IDLitParameterSet Properties	3317
IDLitParameterSet::Add	3318
IDLitParameterSet::Cleanup	3320
IDLitParameterSet::Copy	3321
IDLitParameterSet::Get	3322
IDLitParameterSet::GetByName	3324
IDLitParameterSet::GetParameterName	3326
IDLitParameterSet::Init	3327
IDLitParameterSet::Remove	3329
IDLitReader	3330
IDLitReader Properties	3332
IDLitReader::Cleanup	3333
IDLitReader::GetData	3334
IDLitReader::GetFileExtensions	3335
IDLitReader::GetFilename	3336
IDLitReader::GetProperty	3337
IDLitReader::Init	3338
IDLitReader::IsA	3340
IDLitReader::SetFilename	3341
IDLitReader::SetProperty	3342
IDLitTool	3343
IDLitTool Properties	3347
IDLitTool::ActivateManipulator	3350
IDLitTool::Add	3351
IDLitTool::AddService	3352
IDLitTool::Cleanup	3353
IDLitTool::CommitActions	3354
IDLitTool::DisableUpdates	3355
IDLitTool::DoAction	3356
IDLitTool::DoSetProperty	3357
IDLitTool::DoUIService	3359

IDLitTool::EnableUpdates	3360
IDLitTool::FindIdentifiers	3361
IDLitTool::GetCurrentManipulator	3364
IDLitTool::GetFileReader	3365
IDLitTool::GetFileWriter	3366
IDLitTool::GetManipulators	3367
IDLitTool::GetOperations	3368
IDLitTool::GetProperty	3369
IDLitTool::GetSelectedItems	3370
IDLitTool::GetService	3371
IDLitTool::GetVisualization	3372
IDLitTool::Init	3374
IDLitTool::RefreshCurrentWindow	3376
IDLitTool::Register	3377
IDLitTool::RegisterCustomization	3380
IDLitTool::RegisterFileReader	3381
IDLitTool::RegisterFileWriter	3383
IDLitTool::RegisterManipulator	3386
IDLitTool::RegisterOperation	3388
IDLitTool::RegisterStatusBarSegment	3392
IDLitTool::RegisterVisualization	3393
IDLitTool::SetProperty	3395
IDLitTool::UnRegister	3396
IDLitTool::UnRegisterCustomization	3397
IDLitTool::UnRegisterFileReader	3398
IDLitTool::UnRegisterFileWriter	3399
IDLitTool::UnRegisterManipulator	3400
IDLitTool::UnRegisterOperation	3401
IDLitTool::UnRegisterStatusBarSegment	3402
IDLitTool::UnRegisterVisualization	3403
IDLitToolbase	3404
IDLitUI	3406
IDLitUI Properties	3408
IDLitUI::AddOnNotifyObserver	3409
IDLitUI::Cleanup	3411
IDLitUI::DoAction	3412

IDLitUI::GetProperty	3413
IDLitUI::GetTool	3414
IDLitUI::GetWidgetByName	3415
IDLitUI::Init	3416
IDLitUI::RegisterUIService	3417
IDLitUI::RegisterWidget	3419
IDLitUI::RemoveOnNotifyObserver	3421
IDLitUI::SetProperty	3422
IDLitUI::UnRegisterUIService	3423
IDLitUI::UnRegisterWidget	3424
IDLitVisualization	3425
IDLitVisualization Properties	3429
IDLitVisualization::Add	3432
IDLitVisualization::Aggregate	3434
IDLitVisualization::BeginManipulation	3435
IDLitVisualization::Cleanup	3436
IDLitVisualization::EndManipulation	3437
IDLitVisualization::Get	3438
IDLitVisualization::GetCenterRotation	3440
IDLitVisualization::GetCurrentSelectionVisual	3442
IDLitVisualization::GetDataSpace	3443
IDLitVisualization::GetDataString	3444
IDLitVisualization::GetDefaultSelectionVisual	3445
IDLitVisualization::GetManipulatorTarget	3446
IDLitVisualization::GetProperty	3447
IDLitVisualization::GetRequestedAxesStyle	3448
IDLitVisualization::GetSelectionVisual	3449
IDLitVisualization::GetTypes	3450
IDLitVisualization::GetXYZRange	3451
IDLitVisualization::Init	3453
IDLitVisualization::Is3D	3454
IDLitVisualization::IsIsotropic	3455
IDLitVisualization::IsManipulatorTarget	3456
IDLitVisualization::IsSelected	3457
IDLitVisualization::Move	3458
IDLitVisualization::On2DRotate	3459

IDLitVisualization::OnAxesRequestChange	3460
IDLitVisualization::OnAxesStyleRequestChange	3461
IDLitVisualization::OnDataChange	3462
IDLitVisualization::OnDataComplete	3463
IDLitVisualization::OnDataRangeChange	3464
IDLitVisualization::OnDimensionChange	3465
IDLitVisualization::OnWorldDimensionChange	3466
IDLitVisualization::Remove	3467
IDLitVisualization::RequestsAxes	3468
IDLitVisualization::Restore	3469
IDLitVisualization::Rotate	3470
IDLitVisualization::Scale	3472
IDLitVisualization::Select	3474
IDLitVisualization::Set3D	3476
IDLitVisualization::SetAxesRequest	3477
IDLitVisualization::SetAxesStyleRequest	3479
IDLitVisualization::SetCurrentSelectionVisual	3481
IDLitVisualization::SetData	3482
IDLitVisualization::SetDefaultSelectionVisual	3483
IDLitVisualization::SetParameterSet	3484
IDLitVisualization::SetProperty	3485
IDLitVisualization::UpdateSelectionVisual	3486
IDLitVisualization::VisToWindow	3487
IDLitVisualization::WindowToVis	3489
IDLitWindow	3491
IDLitWindow Properties	3493
IDLitWindow::Add	3494
IDLitWindow::AddWindowEventObserver	3495
IDLitWindow::Cleanup	3496
IDLitWindow::ClearSelections	3497
IDLitWindow::DoHitTest	3498
IDLitWindow::GetEventMask	3500
IDLitWindow::GetProperty	3502
IDLitWindow::GetSelectedItems	3503
IDLitWindow::Init	3504
IDLitWindow::OnKeyboard	3506

IDLitWindow::OnMouseDown	3509
IDLitWindow::OnMouseMotion	3511
IDLitWindow::OnMouseUp	3513
IDLitWindow::OnScroll	3515
IDLitWindow::Remove	3516
IDLitWindow::RemoveWindowEventObserver	3517
IDLitWindow::SetCurrentZoom	3518
IDLitWindow::SetEventMask	3519
IDLitWindow::SetManipulatorManager	3521
IDLitWindow::SetProperty	3522
IDLitWindow::ZoomIn	3523
IDLitWindow::ZoomOut	3524
IDLitWriter	3525
IDLitWriter Properties	3527
IDLitWriter::Cleanup	3528
IDLitWriter::GetFileExtensions	3529
IDLitWriter::GetFilename	3530
IDLitWriter::GetProperty	3531
IDLitWriter::Init	3532
IDLitWriter::IsA	3534
IDLitWriter::SetData	3535
IDLitWriter::SetFilename	3536
IDLitWriter::SetProperty	3537
Chapter 8:	
Graphics Object Classes	3539
IDLgrAxis	3540
IDLgrAxis Properties	3542
IDLgrAxis::Cleanup	3565
IDLgrAxis::GetCTM	3566
IDLgrAxis::GetProperty	3568
IDLgrAxis::Init	3569
IDLgrAxis::SetProperty	3571
IDLgrBuffer	3572
IDLgrBuffer Properties	3574
IDLgrBuffer::Cleanup	3579

IDLgrBuffer::Draw	3580
IDLgrBuffer::Erase	3581
IDLgrBuffer::GetContiguousPixels	3582
IDLgrBuffer::GetDeviceInfo	3583
IDLgrBuffer::GetFontnames	3585
IDLgrBuffer::GetProperty	3587
IDLgrBuffer::GetTextDimensions	3588
IDLgrBuffer::Init	3590
IDLgrBuffer::PickData	3592
IDLgrBuffer::Read	3595
IDLgrBuffer::Select	3596
IDLgrBuffer::SetProperty	3599
IDLgrClipboard	3600
IDLgrClipboard Properties	3602
IDLgrClipboard::Cleanup	3607
IDLgrClipboard::Draw	3608
IDLgrClipboard::GetContiguousPixels	3616
IDLgrClipboard::GetDeviceInfo	3617
IDLgrClipboard::GetFontnames	3619
IDLgrClipboard::GetProperty	3621
IDLgrClipboard::GetTextDimensions	3622
IDLgrClipboard::Init	3624
IDLgrClipboard::SetProperty	3626
IDLgrColorbar	3627
IDLgrColorbar Properties	3630
IDLgrColorbar::Cleanup	3640
IDLgrColorbar::ComputeDimensions	3641
IDLgrColorbar::GetProperty	3643
IDLgrColorbar::Init	3644
IDLgrColorbar::SetProperty	3646
IDLgrContour	3647
IDLgrContour Properties	3650
IDLgrContour::AdjustLabelOffsets	3677
IDLgrContour::Cleanup	3678
IDLgrContour::GetCTM	3679
IDLgrContour::GetLabelInfo	3681

IDLgrContour::GetProperty	3683
IDLgrContour::Init	3684
IDLgrContour::SetProperty	3686
IDLgrFont	3687
IDLgrFont Properties	3688
IDLgrFont::Cleanup	3690
IDLgrFont::GetProperty	3691
IDLgrFont::Init	3692
IDLgrFont::SetProperty	3694
IDLgrImage	3695
IDLgrImage Properties	3698
IDLgrImage::Cleanup	3712
IDLgrImage::GetCTM	3713
IDLgrImage::GetProperty	3715
IDLgrImage::Init	3716
IDLgrImage::SetProperty	3718
IDLgrLegend	3719
IDLgrLegend Properties	3722
IDLgrLegend::Cleanup	3733
IDLgrLegend::ComputeDimensions	3734
IDLgrLegend::GetProperty	3736
IDLgrLegend::Init	3737
IDLgrLegend::SetProperty	3739
IDLgrLight	3740
IDLgrLight Properties	3742
IDLgrLight::Cleanup	3749
IDLgrLight::GetCTM	3750
IDLgrLight::GetProperty	3752
IDLgrLight::Init	3753
IDLgrLight::SetProperty	3755
IDLgrModel	3756
IDLgrModel Properties	3758
IDLgrModel::Add	3764
IDLgrModel::Cleanup	3765
IDLgrModel::Draw	3766
IDLgrModel::GetByName	3767

IDLgrModel::GetCTM	3769
IDLgrModel::GetProperty	3771
IDLgrModel::Init	3772
IDLgrModel::Reset	3774
IDLgrModel::Rotate	3775
IDLgrModel::Scale	3776
IDLgrModel::SetProperty	3777
IDLgrModel::Translate	3778
IDLgrMPEG	3779
IDLgrMPEG Properties	3781
IDLgrMPEG::Cleanup	3788
IDLgrMPEG::GetProperty	3789
IDLgrMPEG::Init	3790
IDLgrMPEG::Put	3792
IDLgrMPEG::Save	3793
IDLgrMPEG::SetProperty	3794
IDLgrPalette	3795
IDLgrPalette Properties	3797
IDLgrPalette::Cleanup	3800
IDLgrPalette::GetRGB	3801
IDLgrPalette::GetProperty	3802
IDLgrPalette::Init	3803
IDLgrPalette::LoadCT	3805
IDLgrPalette::NearestColor	3806
IDLgrPalette::SetRGB	3807
IDLgrPalette::SetProperty	3808
IDLgrPattern	3809
IDLgrPattern Properties	3811
IDLgrPattern::Cleanup	3814
IDLgrPattern::GetProperty	3815
IDLgrPattern::Init	3816
IDLgrPattern::SetProperty	3818
IDLgrPlot	3819
IDLgrPlot Properties	3821
IDLgrPlot::Cleanup	3837
IDLgrPlot::GetCTM	3838

IDLgrPlot::GetProperty	3840
IDLgrPlot::Init	3841
IDLgrPlot::SetProperty	3843
IDLgrPolygon	3844
IDLgrPolygon Properties	3846
IDLgrPolygon::Cleanup	3868
IDLgrPolygon::GetCTM	3869
IDLgrPolygon::GetProperty	3871
IDLgrPolygon::Init	3872
IDLgrPolygon::SetProperty	3874
IDLgrPolyline	3875
IDLgrPolyline Properties	3877
IDLgrPolyline::Cleanup	3895
IDLgrPolyline::GetCTM	3896
IDLgrPolyline::GetProperty	3898
IDLgrPolyline::Init	3899
IDLgrPolyline::SetProperty	3901
IDLgrPrinter	3902
IDLgrPrinter Properties	3904
IDLgrPrinter::Cleanup	3910
IDLgrPrinter::Draw	3911
IDLgrPrinter::GetContiguousPixels	3916
IDLgrPrinter::GetFontnames	3917
IDLgrPrinter::GetProperty	3919
IDLgrPrinter::GetTextDimensions	3920
IDLgrPrinter::Init	3922
IDLgrPrinter::NewDocument	3924
IDLgrPrinter::NewPage	3925
IDLgrPrinter::SetProperty	3926
IDLgrROI	3927
IDLgrROI Properties	3929
IDLgrROI::Cleanup	3939
IDLgrROI::GetProperty	3940
IDLgrROI::Init	3941
IDLgrROI::PickVertex	3943
IDLgrROI::SetProperty	3945

IDLgrROIGroup	3946
IDLgrROIGroup Properties	3948
IDLgrROIGroup::Add	3955
IDLgrROIGroup::Cleanup	3956
IDLgrROIGroup::GetProperty	3957
IDLgrROIGroup::Init	3958
IDLgrROIGroup::PickRegion	3960
IDLgrROIGroup::SetProperty	3962
IDLgrScene	3963
IDLgrScene Properties	3965
IDLgrScene::Add	3968
IDLgrScene::Cleanup	3969
IDLgrScene::GetByName	3970
IDLgrScene::GetProperty	3972
IDLgrScene::Init	3973
IDLgrScene::SetProperty	3975
IDLgrSurface	3976
IDLgrSurface Properties	3978
IDLgrSurface::Cleanup	4003
IDLgrSurface::GetCTM	4004
IDLgrSurface::GetProperty	4006
IDLgrSurface::Init	4007
IDLgrSurface::SetProperty	4009
IDLgrSymbol	4010
IDLgrSymbol Properties	4012
IDLgrSymbol::Cleanup	4016
IDLgrSymbol::GetProperty	4017
IDLgrSymbol::Init	4018
IDLgrSymbol::SetProperty	4020
IDLgrTessellator	4021
IDLgrTessellator Properties	4024
IDLgrTessellator::AddPolygon	4025
IDLgrTessellator::Cleanup	4027
IDLgrTessellator::Init	4028
IDLgrTessellator::Reset	4029
IDLgrTessellator::Tessellate	4030

IDLgrText	4032
IDLgrText Properties	4034
IDLgrText::Cleanup	4049
IDLgrText::GetCTM	4050
IDLgrText::GetProperty	4052
IDLgrText::Init	4053
IDLgrText::SetProperty	4055
IDLgrView	4056
IDLgrView Properties	4058
IDLgrView::Add	4065
IDLgrView::Cleanup	4066
IDLgrView::GetByName	4067
IDLgrView::GetProperty	4069
IDLgrView::Init	4070
IDLgrView::SetProperty	4072
IDLgrViewgroup	4073
IDLgrViewgroup Properties	4075
IDLgrViewgroup::Add	4077
IDLgrViewgroup::Cleanup	4078
IDLgrViewgroup::GetByName	4079
IDLgrViewgroup::GetProperty	4081
IDLgrViewgroup::Init	4082
IDLgrViewgroup::SetProperty	4084
IDLgrVolume	4085
IDLgrVolume Properties	4087
IDLgrVolume::Cleanup	4104
IDLgrVolume::ComputeBounds	4105
IDLgrVolume::GetCTM	4106
IDLgrVolume::GetProperty	4108
IDLgrVolume::Init	4109
IDLgrVolume::PickVoxel	4111
IDLgrVolume::SetProperty	4113
IDLgrVRML	4114
IDLgrVRML Properties	4117
IDLgrVRML::Cleanup	4123
IDLgrVRML::Draw	4124

IDLgrVRML::GetDeviceInfo	4125
IDLgrVRML::GetFontnames	4127
IDLgrVRML::GetProperty	4129
IDLgrVRML::GetTextDimensions	4130
IDLgrVRML::Init	4132
IDLgrVRML::SetProperty	4134
IDLgrWindow	4135
IDLgrWindow Properties	4138
IDLgrWindow::Cleanup	4149
IDLgrWindow::Draw	4150
IDLgrWindow::Erase	4152
IDLgrWindow::GetContiguousPixels	4153
IDLgrWindow::GetDeviceInfo	4154
IDLgrWindow::GetDimensions	4156
IDLgrWindow::GetFontnames	4158
IDLgrWindow::GetProperty	4160
IDLgrWindow::GetTextDimensions	4161
IDLgrWindow::Iconify	4163
IDLgrWindow::Init	4164
IDLgrWindow::PickData	4166
IDLgrWindow::Read	4169
IDLgrWindow::Select	4170
IDLgrWindow::SetCurrentCursor	4173
IDLgrWindow::SetCurrentZoom	4175
IDLgrWindow::SetProperty	4176
IDLgrWindow::Show	4177
IDLgrWindow::ZoomIn	4178
IDLgrWindow::ZoomOut	4179

Chapter 9: Miscellaneous Object Classes 4181

IDL_Container	4182
IDL_Container Properties	4183
IDL_Container::Add	4184
IDL_Container::Cleanup	4185
IDL_Container::Count	4186

IDL_Container::Get	4187
IDL_Container::Init	4189
IDL_Container::IsContained	4190
IDL_Container::Move	4191
IDL_Container::Remove	4192
IDL_Savefile	4193
IDL_Savefile Properties	4196
IDL_Savefile::Cleanup	4197
IDL_Savefile::Contents	4198
IDL_Savefile::Init	4201
IDL_Savefile::Names	4203
IDL_Savefile::Restore	4206
IDL_Savefile::Size	4213
IDLcomActiveX	4215
IDLcomActiveX Properties	4216
IDLcomIDispatch	4217
IDLcomIDispatch Properties	4219
IDLcomIDispatch::GetProperty	4220
IDLcomIDispatch::Init	4221
IDLcomIDispatch::SetProperty	4222
IDLjavaObject	4223
IDLjavaObject Properties	4225
IDLjavaObject::GetProperty	4226
IDLjavaObject::Init	4227
IDLjavaObject::SetProperty	4229
TrackBall	4231
TrackBall Properties	4232
TrackBall::Init	4234
TrackBall::Reset	4235
TrackBall::Update	4237

Part III: Appendices

Appendix A: IDL Graphics Devices	4243
Supported Devices	4244

Keywords Accepted by the IDL Devices	4246
Window Systems	4286
Backing Store	4286
Image Display On Monochrome Devices	4288
Printing Graphics Output Files	4289
Setting Up The Printer	4290
Positioning Graphics Output	4291
Image Background Color	4291
The CGM Device	4292
Abilities and Limitations	4292
The HP-GL Device	4294
Abilities And Limitations	4295
HP-GL Linestyles	4295
The Metafile Display Device	4296
The Null Display Device	4298
The PCL Device	4299
The Printer Device	4301
The PostScript Device	4302
Using PostScript Fonts	4303
Color PostScript	4303
PostScript Positioning	4305
Importing IDL Plots into Other Documents	4310
The Regis Terminal Device	4315
Defaults for Regis Devices	4315
Regis Limitations	4315
The Tektronix Device	4316
The DEVICE Procedure For Tektronix Terminals	4316
Tektronix Limitations	4316
Tektronix Device Limitations	4317
The Microsoft Windows Device	4318
The X Windows Device	4319
X Windows Visuals	4319
Using Color Under X	4322
Using Pixmaps	4324
How Color is Interpreted for a TrueColor Visual	4326
Setting the X Window Defaults	4327

The Z-Buffer Device	4328
Reading and Writing Buffers	4329
Z-Axis Scaling	4329
Polyfill Procedure	4329
Examples Using the Z-Buffer	4330

Appendix B: Graphics Keywords 4333

BACKGROUND	4334
CHANNEL	4335
CHARSIZE	4335
CHARTHICK	4335
CLIP	4335
COLOR	4336
DATA	4336
DEVICE	4336
FONT	4337
LINestyle	4337
NOCLIP	4338
NODATA	4338
NOERASE	4339
NORMAL	4339
ORIENTATION	4339
POSITION	4339
PSYM	4340
SUBTITLE	4341
SYMSIZE	4341
T3D	4342
THICK	4342
TICKLEN	4342
TITLE	4343
[XYZ]CHARSIZE	4343
[XYZ]GRIDSTYLE	4343
[XYZ]MARGIN	4343
[XYZ]MINOR	4343
[XYZ]RANGE	4344

[XYZ]STYLE	4344
[XYZ]THICK	4344
[XYZ]TICK_GET	4345
[XYZ]TICKFORMAT	4345
[XYZ]TICKINTERVAL	4348
[XYZ]TICKLAYOUT	4348
[XYZ]TICKLEN	4348
[XYZ]TICKNAME	4349
[XYZ]TICKS	4349
[XYZ]TICKUNITS	4349
[XYZ]TICKV	4350
[XYZ]TITLE	4350
Z	4351
ZVALUE	4351

Appendix C: Thread Pool Keywords 4353

Appendix D: System Variables 4355

What Are System Variables?	4356
Constant System Variables	4357
!DPI	4357
!DTOR	4357
!MAP	4357
!PI	4357
!RADEG	4357
!VALUES	4357
Error Handling System Variables	4359
!ERR	4359
!ERROR_STATE	4359
!ERROR	4360
!ERR_STRING	4360
!EXCEPT	4361
!MOUSE	4361
!MSG_PREFIX	4362
!SYSERROR	4362

!SYSERR_STRING	4362
!WARN	4362
IDL Environment System Variables	4363
!CPU	4363
!DIR	4365
!DLM_PATH	4365
!EDIT_INPUT	4367
!HELP_PATH	4367
!JOURNAL	4367
!MAKE_DLL	4368
!MORE	4370
!PATH	4371
!PROMPT	4372
!QUIET	4372
!VERSION	4372
Graphics System Variables	4375
!C System Variable	4375
!D System Variable	4375
!ORDER System Variable	4379
!P System Variable	4379
!X, !Y, !Z System Variables	4383
Appendix E:	
IDL Operators	4391
Mathematical Operators	4392
Minimum and Maximum Operators	4394
Matrix Operators	4395
Logical Operators	4396
Bitwise Operators	4397
Relational Operators	4399
Other Operators	4400
Operator Precedence	4402
Appendix F:	
Special Characters	4405
Exclamation Point (!)	4406
Apostrophe (')	4407

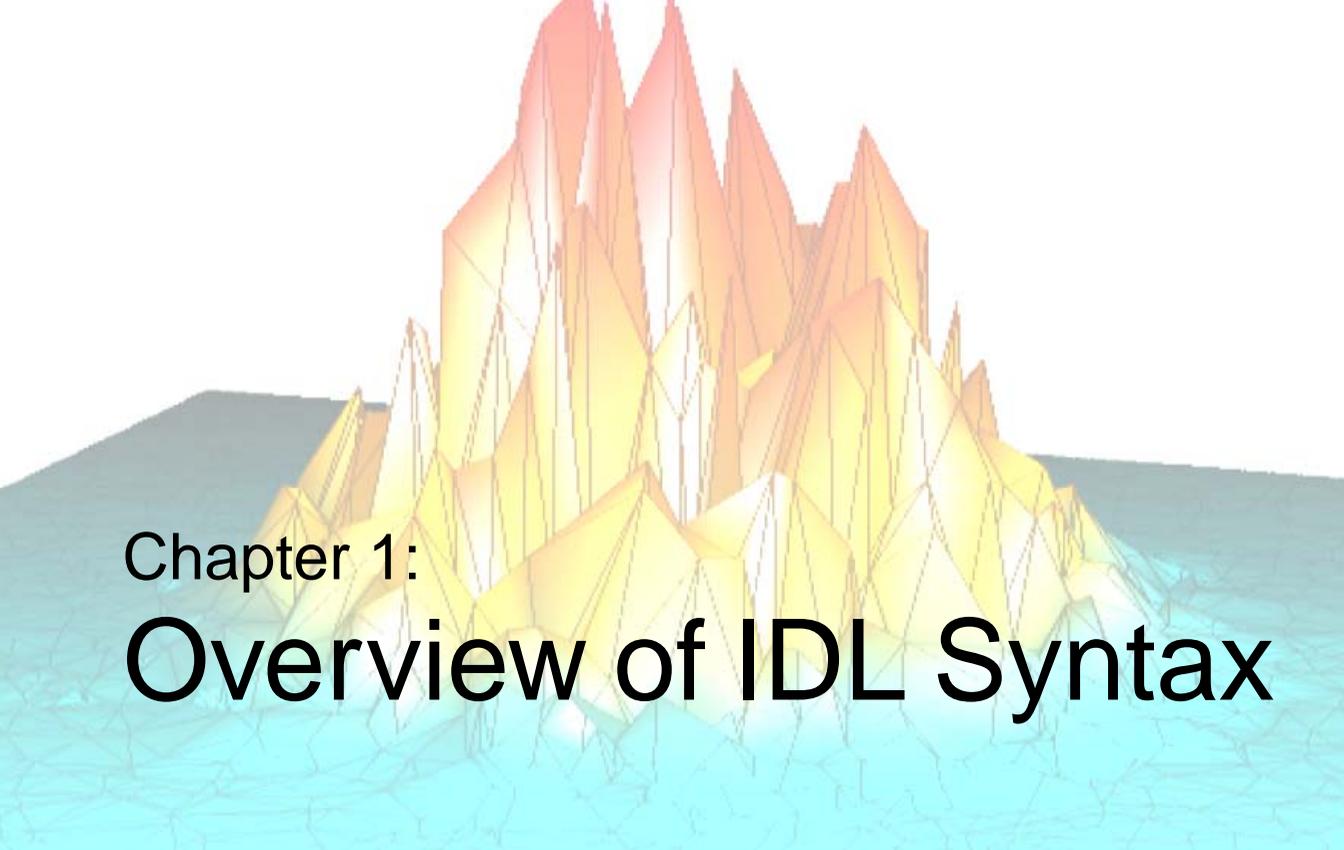
Semicolon (;)	4407
Dollar Sign (\$)	4407
Quotation Mark (")	4407
Period (.)	4407
Ampersand (&)	4408
Colon (:)	4408
Asterisk (*)	4408
At Sign (@)	4408
Question Mark (?)	4409

Appendix G: Reserved Words 4411

Appendix H: Fonts 4413

Overview	4414
Fonts in IDL Direct vs. Object Graphics	4415
IDL Direct Graphics	4415
IDL Object Graphics	4415
About Vector Fonts	4416
Using Vector Fonts	4416
Specifying Font Size	4416
ISO Latin 1 Encoding	4417
Customizing the Vector Fonts	4418
About TrueType Fonts	4419
Using TrueType Fonts	4420
Specifying Font Size	4420
Using Embedded Formatting Commands	4421
IDL TrueType Font Resource Files	4421
Adding Your Own Fonts	4422
Where IDL Searches for Fonts	4422
About Device Fonts	4424
Which Device Fonts Are Available?	4424
Using Device Fonts	4425
Fonts and the PostScript Device	4426
Choosing a Font Type	4431
Appearance	4431

Three-Dimensional Transformations	4431
Portability	4431
Computational Time	4432
Flexibility	4432
Print Quality	4432
Embedded Formatting Commands	4433
Changing Fonts within a String	4433
Positioning Commands	4435
Formatting Command Examples	4437
A Complex Equation	4438
Vector-Drawn Font Example	4439
TrueType Font Samples	4442
Vector Font Samples	4445
Appendix I:	
Obsolete Features	4455
What Are Obsolete Features?	4456
Features Obsolete in IDL 6.1	4457
Features Obsolete in IDL 6.0	4458
Features Obsolete in IDL 5.6	4459
Features Obsolete in IDL 5.5	4460
Features Obsolete in IDL 5.4	4461
Features Obsolete in IDL 5.3	4462
SDF Routines Obsolete in IDL 5.3	4463
What is DFSD and Why Are We Obsoleting It?	4463
Features Obsolete in IDL 5.2	4464
Features Obsolete in IDL 5.1	4465
Features Obsolete in IDL 5.0	4466
Features Obsolete in IDL 4.0 or Earlier	4467
Obsolete Arguments and Keywords	4473
Obsolete System Variables	4479
Obsolete Graphics Devices	4481
Index	4483



Chapter 1: Overview of IDL Syntax

This reference is a complete listing of all built-in IDL functions, procedures, statements, executive commands, and objects, collectively referred to as “commands.” Every IDL language element that can be used either at the command line or in a program is listed alphabetically. A description of each routine follows its name.

Note

Descriptions of Scientific Data Formats routines (CDF_*, EOS_*, HDF_*, and NCDF_* routines) can be found in the *Scientific Data Formats* book.

Routines written in the IDL language are noted as such, and the location of the `.pro` file within the IDL distribution is specified. You may wish to inspect the IDL source code for some of these routines to gain further insight into their inner workings.

Conventions used in this reference guide are described below.

IDL Syntax

The following table lists the elements used in IDL syntax listings:

Element	Description
[] (Square brackets)	Indicates that the contents are optional. Do not include the brackets in your call.
[] (Italicized square brackets)	Indicates that the square brackets are part of the statement (used to define an array).
Argument	Arguments are shown in italics, and must be specified in the order listed.
KEYWORD	Keywords are all caps, and can be specified in any order. For functions, all arguments and keywords must be contained within parentheses.
/KEYWORD	Indicates a boolean keyword.
<i>Italics</i>	Indicates arguments, expressions, or statements for which you must provide values.
{ } (Braces)	<ul style="list-style-type: none"> Indicates that you must choose one of the values they contain Encloses a list of possible values, separated by vertical lines () Encloses useful information about a keyword Defines an IDL structure (this is the only case in which the braces are included in the call).
(Vertical lines)	Separates multiple values or keywords.
[, <i>Value</i> ₁ , ... , <i>Value</i> _{<i>n</i>}]	Indicates that any number of values can be specified.
[, <i>Value</i> ₁ , ... , <i>Value</i> ₈]	Indicates the maximum number of values that can be specified.

Table 1-1: Elements of IDL Syntax

Elements of Syntax

Square Brackets ([])

- Content between square brackets is optional. Pay close attention to the grouping of square brackets. Consider the following examples:

ROUTINE_NAME, *Value1* [*Value2*] [*Value3*]: You must include *Value1*. You do not have to include *Value2* or *Value3*. *Value2* and *Value3* can be specified independently.

ROUTINE_NAME, *Value1* [*Value2*, *Value3*]: You must include *Value1*. You do not have to include *Value2* or *Value3*, but you must include both *Value2* and *Value3*, or neither.

ROUTINE_NAME [*Value1* [*Value2*]]: You can specify *Value1* without specifying *Value2*, but if you specify *Value2*, you must also specify *Value1*.

- Do not include square brackets in your statement unless the brackets are italicized. Consider the following syntax:

Result = KRIG2D(*Z* [*X*, *Y*] [*BOUNDS*=[*xmin*, *ymin*, *xmax*, *ymax*])

An example of a valid statement is:

R = KRIG2D(*Z*, *X*, *Y*, *BOUNDS*=[0,0,1,1])

- Note that when [*Value1*, ... , *Value_n*] is listed, you can specify any number of arguments. When an explicit number is listed, as in [*Value1*, ... , *Value₈*], you can specify only as many arguments as are listed.

Braces ({ })

- For certain keywords, a list of the possible values is provided. This list is enclosed in braces, and the choices are separated by a vertical line (|). Do not include the braces in your statement. For example, consider the following syntax:

READ_JPEG [*TRUE*={ 1 | 2 | 3 }]

In this example, you must choose either 1, 2, or 3. An example of a valid statement is:

READ_JPEG, *TRUE*=1

- Braces are used to enclose the allowable range for a keyword value. Unless otherwise noted, ranges provided are inclusive. Consider the following syntax:

Result = CVTT0BM(*Array* [*THRESHOLD*=*value*{0 to 255}])

An example of a valid statement is:

Result = CVTTOBM(A, THRESHOLD=150)

- Braces are also used to provide useful information about a keyword. For example:

[, LABEL=*n*{label every *n*th gridline}]

Do not include the braces or their content in your statement.

- Certain keywords are prefaced by X, Y, or Z. Braces are used for these keywords to indicate that you must choose one of the values it contains. For example, [{X | Y}RANGE=*array*] indicates that you can specify either XRANGE=*array* or YRANGE=*array*.
- Note that in IDL, braces are used to define structures. When defining a structure, you *do* want to include the braces in your statement.

Italics

- Italicized words are arguments, expressions, or statements for which you must provide values. The value you provide can be a numerical value, such as 10, an expression, such as DIST(100), or a named variable. For keywords that expect a string value, the syntax is listed as KEYWORD=*string*. The value you provide can be a string, such as 'Hello' (enclosed in single quotation marks), or a variable that holds a string value.
- The italicized values that must be provided for keywords are listed in the most helpful terms possible. For example, [, XSIZE=*pixels*] indicates that the XSIZE keyword expects a value in pixels, while [, ORIENTATION=*ccw_degrees_from_horiz*] indicates that you must provide a value in degrees, measured counter-clockwise from horizontal.

Procedures

IDL procedures use the following general syntax:

PROCEDURE_NAME, *Argument* [, *Optional_Argument*]

where PROCEDURE_NAME is the name of the procedure, *Argument* is a required parameter, and *Optional_Argument* is an optional parameter to the procedure.

Functions

IDL functions use the following general syntax:

$Result = FUNCTION_NAME(Argument [, Optional_Argument])$

where *Result* is the returned value of the function, *FUNCTION_NAME* is the name of the function, *Argument* is a required parameter, and *Optional_Argument* is an optional parameter. Note that all arguments and keyword arguments to functions should be supplied *within* the parentheses that follow the function's name.

Functions do not always have to be used in assignment statements (i.e., $A = SIN(10.2)$), they can be used just like any other IDL expression. For example, you could print the result of $SIN(10.2)$ by entering the command:

```
PRINT, SIN(10.2)
```

Arguments

The “Arguments” section describes each valid argument to the routine. Note that these arguments are positional parameters that must be supplied in the order indicated by the routine's syntax.

Named Variables

Often, arguments that contain values upon return from the function or procedure (“output arguments”) are described as accepting “named variables”. A named variable is simply a valid IDL variable name. This variable *does not* need to be defined before being used as an output argument. Note, however that when an argument calls for a named variable, only a named variable can be used—sending an expression causes an error.

Keywords

The “Keywords” section describes each valid keyword argument to the routine. Note that keyword arguments are formal parameters that can be supplied in any order.

Keyword arguments are supplied to IDL routines by including the keyword name followed by an equal sign (“=”) and the value to which the keyword should be set. The value can be a value, an expression, or a *named variable* (a named variable is simply a valid IDL variable name).

Note

If you set a keyword equal to an *undefined* named variable, IDL will quietly ignore the value.

For example, to produce a plot with diamond-shaped plotting symbols, the *PSYM* keyword should be set to 4 as follows:

```
PLOT, FINDGEN(10), PSYM=4
```

Note the following when specifying keywords:

- Certain keywords are boolean, meaning they can be set to either 0 or 1. These keywords are switches used to turn an option on and off. Usually, setting such keywords equal to 1 causes the option to be turned on. Explicitly setting the keyword to 0 (or not including the keyword) turns the option off. In the syntax listings in this reference, all keywords that are preceded by a slash can be set by prefacing them by the slash. For example, SURFACE, DIST(10), /SKIRT is a shortcut for SURFACE, DIST(10), SKIRT=1. To turn the option back off, you must set the keyword equal to 0, as in SURFACE, DIST(10), SKIRT=0.

In rare cases, a keyword's default value is 1. In these cases, the syntax is listed as `KEYWORD=0`, as in `SLIDE_IMAGE [, Image] [, CONGRID=0]`. In this example, `CONGRID` is set to 1 by default. If you specify `CONGRID=0`, you can turn it back on by specifying either `/CONGRID` or `CONGRID=1`.

- Some keywords are used to obtain values that can be used upon return from the function or procedure. These keywords are listed as `KEYWORD=variable`. Any valid variable name can be used for these keywords, and the variable does not need to be defined first. Note, however, that when a keyword calls for a named variable, only a named variable can be used—sending an expression causes an error.

For example, the `WIDGET_CONTROL` procedure can return the user values of widgets in a named variable using the `GET_UVALUE` keyword. To return the user value for a widget ID (contained in the variable `mywidget`) in the variable `userval`, you would use the command:

```
WIDGET_CONTROL, mywidget, GET_UVALUE = userval
```

Upon return from the procedure, `userval` contains the user value. Note that `userval` did not have to be defined before the call to `WIDGET_CONTROL`.

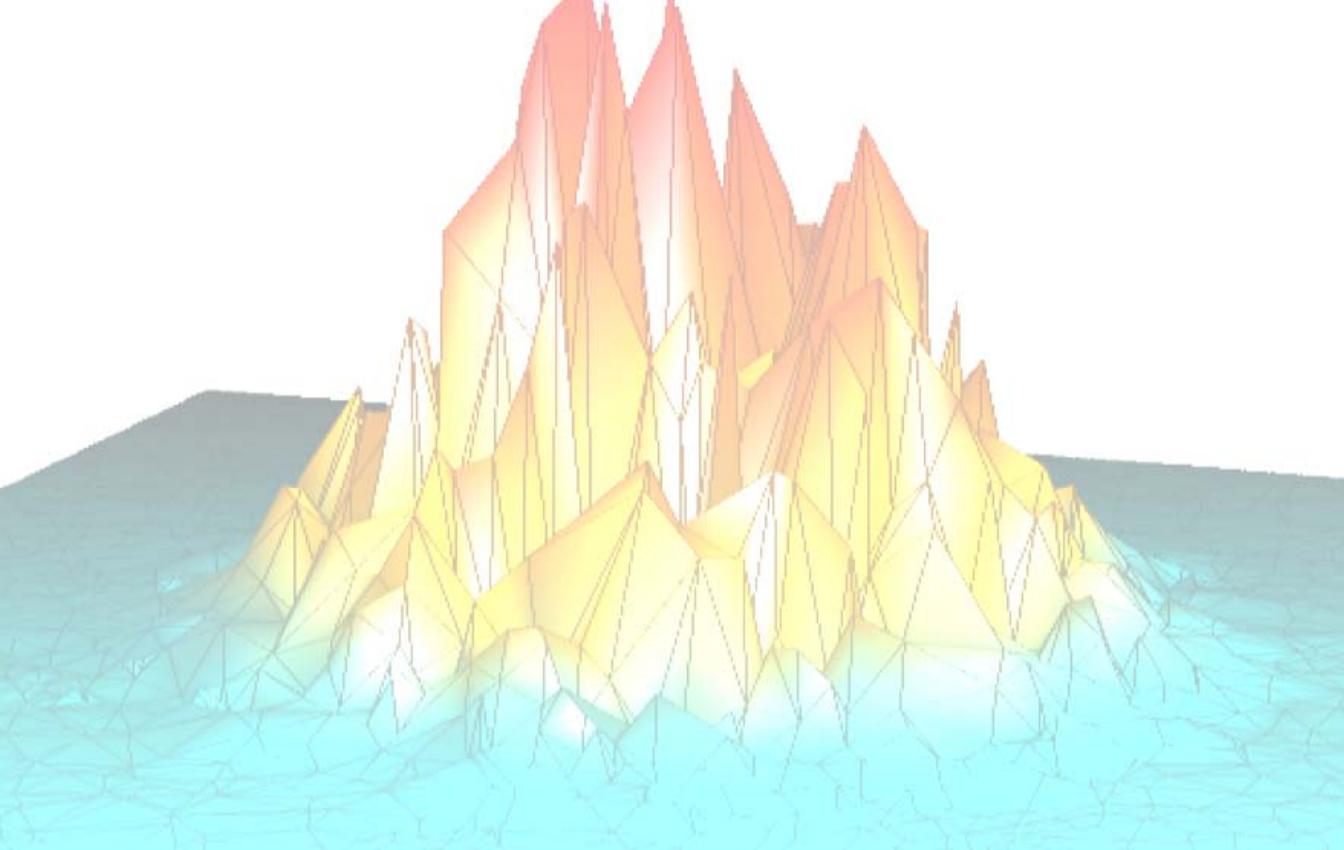
- Some routines have keywords that are mutually exclusive, meaning only one of the keywords can be present in a given statement. These keywords are grouped together, and separated by a vertical line. For example, consider the following syntax:

```
PLOT, [X,] Y [, /DATA | , /DEVICE | , /NORMAL]
```

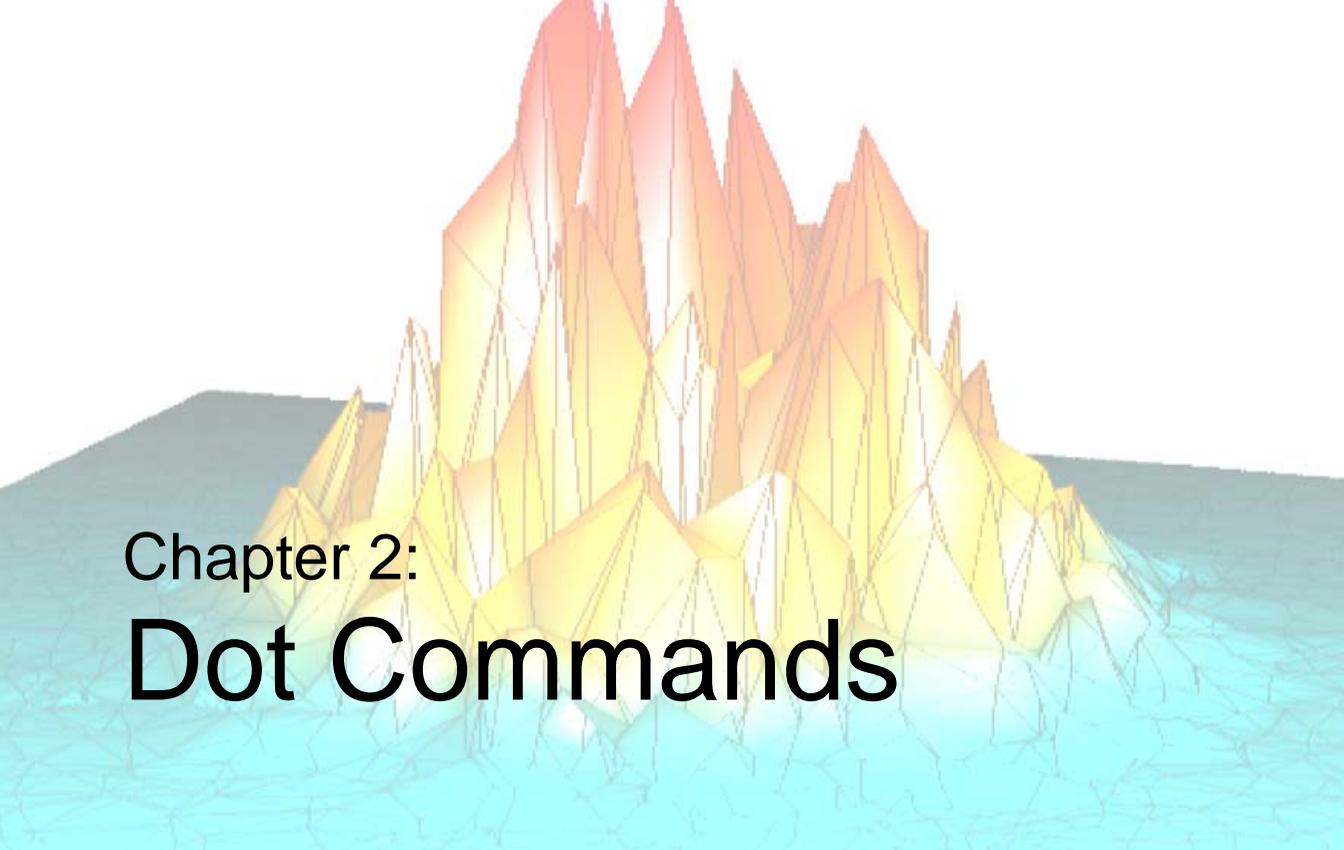
In this example, you can choose either `DATA`, `DEVICE`, or `NORMAL`, but not more than one. An example of a valid statement is:

```
PLOT, SIN(A), /DEVICE
```

- Keywords can be abbreviated to their shortest unique length. For example, the XSTYLE keyword can be abbreviated to XST because there are no other keywords in IDL that begin with XST. You cannot shorten XSTYLE to XS, however, because there are other keywords that begin with XS, such as XSIZE.



Part I: IDL Command Reference



Chapter 2: Dot Commands

.COMPILE

The `.COMPILE` command compiles and saves procedures and programs in the same manner as `.RUN`. If one or more filenames are specified, the procedures and functions contained therein are compiled *but not executed*. If you enter this command at the Command Input Line of the IDLDE and the files are not yet open, IDL opens the files within Editor windows and compiles the procedures and functions contained therein.

See [RESOLVE_ROUTINE](#) for a way to invoke the same operation from within an IDL routine, and [RESOLVE_ALL](#) for a way to automatically compile all user-written or library functions called by all currently-compiled routines.

If the `-f` flag is specified, `File` is compiled from the source stored temporarily in `TempFile` rather than on disk in `File` itself. This allows you to make changes to `File` (in an IDLDE editor window, for example), store the modified source into the temporary file (IDLDE does it automatically), compile, and test the changes without overwriting the original code stored in `File`.

Note

`.COMPILE` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

```
.COMPILE [File1, ..., Filen]
```

```
.COMPILE -f File TempFile
```

See Also

[.RNEW](#), [.RUN](#)

.CONTINUE

The `.CONTINUE` command continues execution of a program that has stopped because of an error, a stop statement, or a keyboard interrupt. IDL saves the location of the beginning of the last statement executed before an error. If it is possible to correct the error condition in the interactive mode, the offending statement can be re-executed by entering `.CONTINUE`. After `STOP` statements, `.CONTINUE` continues execution at the next statement. The `.CONTINUE` command can be abbreviated; for example, `.C`. Execution of a program interrupted by typing `Ctrl+C` also can be resumed at the point of interruption with the `.CONTINUE` command.

Note

`.CONTINUE` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.CONTINUE`

.EDIT

The .EDIT command opens files in IDL Editor windows when called from the Command Input Line of the IDLDE. This functionality is only available on the Windows and Motif platforms. Note that filenames are separated by spaces, not commas.

Note

.EDIT is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.EDIT File1 [File2 ... Filen]`

.FULL_RESET_SESSION

The `.FULL_RESET_SESSION` command does everything `.RESET_SESSION` does, plus the following:

- Removes all system routines installed via `LINKIMAGE` or a DLM.
- Removes all structure definitions installed via a DLM.
- Removes all message blocks added by DLMs or by the `DEFINE_MSGBLK` or `DEFINE_MSGBLK_FROM_FILE` routines.
- Unloads all sharable libraries loaded into IDL via `CALL_EXTERNAL`, `LINKIMAGE`, or a DLM.
- Re-initializes all DLMs to their unloaded initial state.

Note

`.FULL_RESET_SESSION` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

```
.FULL_RESET_SESSION
```

.GO

The .GO command starts execution at the beginning of a previously-compiled main program.

Note

.GO is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

.GO

.OUT

The .OUT command continues executing statements in the current program until it returns.

Note

.OUT is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

.OUT

.RESET_SESSION

The `.RESET_SESSION` command resets much of the state of an IDL session without requiring the user to exit and restart the IDL session.

`.RESET_SESSION` does the following:

- Returns current execution point to `$MAIN$` ([RECALL](#)).
- Removes all breakpoints.
- Clears the path cache (see [PATH_CACHE](#) for details).
- Closes all files except the standard 3 units, the [JOURNAL](#) file (if any), and any files in use by graphics drivers.
- Disables [SHMDEBUG](#) mode.
- Destroys/Removes the following:
 - All local variables in `$MAIN$`.
 - All widgets. Exit handlers are not called.
 - All windows and pixmaps for the current window system graphics device are closed. No other graphics state is reset.
 - All common blocks.
 - All handles
 - All user defined system variables
 - All pointer and object reference heap variables.
 - Object destructors are not called.
 - All user defined structure definitions.
 - All user defined object definitions.
 - All compiled user functions and procedures, including the main program (`$MAIN$`), if any.
 - Any memory segments created by [SHMMAP](#).

The following are not reset:

- The current values of intrinsic system variables are retained.
- The saved commands and output log are preserved.

- Graphics drivers are not reset to their full uninitialized state. However, all windows and pixmaps for the current window system device are closed.
- The following files are not closed:
 - Stdin (LUN 0)
 - Stdout (LUN -1)
 - Stderr (LUN -2)
 - The journal file (!JOURNAL) if one is open.
 - Any files in use by graphics drivers (e.g. PostScript).
- Dynamically loaded graphics drivers (LINKIMAGE) are not removed, nor are any dynamic sharable libraries containing such drivers, even if the same library was also used for another purpose such as CALL_EXTERNAL, LINKIMAGE system routines, or DLMs. See the [.FULL_RESET_SESSION](#) executive command to unload dynamic libraries.
- Message blocks are not removed. See the [.FULL_RESET_SESSION](#) executive command to remove message blocks loaded by DLMs or created using the [DEFINE_MSGBLK](#) or [DEFINE_MSGBLK_FROM_FILE](#) routines.

Note

`.RESET_SESSION` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.RESET_SESSION`

.RETURN

The `.RETURN` command continues execution of a program until encountering a `RETURN` statement. This is convenient for debugging programs since it allows the whole program to run, stopping before returning to the next-higher program level so you can examine local variables.

Also see the [RETURN](#) command.

Note

`.RETURN` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.RETURN`

.RNEW

The .RNEW command compiles and saves procedures and functions in the same manner as .RUN. In addition, all variables in the main program unit, except those in common blocks, are erased. The -T and -L filename switches have the same effect as with .RUN.

Note

.RNEW is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.RNEW [File1, ..., Filen]`

To save listing in a file: `.RNEW -L ListFile.lis File1 [, File2, ..., Filen]`

To display listing on screen: `.RNEW -T File1 [, File2, ..., Filen]`

Example

Some statements using the .RUN and .RNEW commands are shown below.

Statement	Description
<code>.RUN</code>	Accept a program from the keyboard. Retain the present variables.
<code>.RUN myfile</code>	Compile the file myfile.pro. If it is not found in the current directory, try to find it in the directory search path.
<code>.RUN -T A, B, C</code>	Compile the files a.pro, b.pro and c.pro. List the files on the terminal.

Table 2-1: Examples using .RUN and .RNEW

Statement	Description
<code>.RNEW -L myfile.lis myfile, yourfile</code>	Erase all variables and compile the files myfile.pro and yourfile.pro. Produce a listing on myfile.lis.

Table 2-1: Examples using .RUN and .RNEW

See Also

[.COMPILE](#), [.RUN](#)

.RUN

The `.RUN` command compiles procedures, functions, and/or main programs in memory. Main programs are executed immediately. The command can be followed by a list of files to be compiled. Filenames are separated by blanks, tabs, or commas.

If a file specification is included in the command, IDL searches for the file first in the current directory, then in the directories specified by the system variable `!PATH`. See [“Running IDL Programs”](#) in Chapter 9 of the *Using IDL* manual for more information on IDL’s search strategy.

If a main program unit is encountered, execution of the program will begin after all files have been read if there were no errors. The values of all of the variables are retained. If the file isn’t found, input is accepted from the keyboard until a complete program unit is entered.

Files containing IDL procedures, programs, and functions are assumed to have the file extension (suffix) `.pro`. Files created with the `SAVE` procedure are assumed to have the extension `.sav`. See [Chapter 9, “Creating and Running Programs in IDL”](#) in the *Using IDL* manual for further information.

Note

`.RUN` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.RUN [File1, ..., Filen]`

To save listing in a file: `.RUN -L ListFile.lis File1 [, File2, ..., Filen]`

To display listing on screen: `.RUN -T File1 [, File2, ..., Filen]`

Note

Subsequent calls to `.RUN` compile the procedure again.

Using `.RUN` to Make Program Listings

The command arguments `-T` for terminal listing or `-L filename` for listing to a named file can appear after the command name and before the program filenames to produce a numbered program listing directed to the terminal or to a file.

For instance, to see a listing on the screen as a result of compiling a procedure contained in a file named `analyze.pro`, use the following command:

```
.RUN -T analyze
```

To compile the same procedure and save the listing in a file named `analyze.lis`, use the following command:

```
.RUN -L analyze.lis analyze
```

In listings produced by IDL, the line number of each statement is printed at the left margin. This number is the same as that printed in IDL error statements, simplifying location of the statement causing the error.

Note

If the compiled file contains more than one procedure or function, line numbering is reset to “1” each time the end of a program segment is detected.

Each level of block nesting is indented four spaces to the right of the preceding block level to improve the legibility of the program’s structure.

See Also

[.COMPILE](#), [.RNEW](#)

.SKIP

The `.SKIP` command skips one or more statements. It is useful for moving past a program statement that caused an error. If the optional argument n is present, it gives the number of statements to skip; otherwise, a single statement is skipped.

Note that `.SKIP` does not execute or evaluate the code it is skipping. Rather, it arbitrarily alters the current program counter to the n th physical statement following the current point. This has implications that may not be obvious on initial consideration:

- `.SKIP` does not skip into a called routine.
- `.SKIP` moves to the n th physical statement following the current program location. This may not be the statement that execution would have actually have moved to if you had allowed the program to run normally.
- Arbitrarily moving the program counter in this way may leave your program in an unrunnable state, depending on resulting state of the local variables and the statements that the newly positioned program counter attempts to execute next.

In contrast, the `.STEP` executive command has none of the above drawbacks and can be used instead in many situations. The advantage of `.SKIP` over `.STEP` is that `.SKIP` can move past statements that `.STEP` cannot, such as:

- Statements with errors that cause execution to halt.
- Infinite loops, and similar logic errors.

For example, consider the following program segment:

```
..... ..
OPENR, 1, 'missing'
READF, 1, xxx, ..., ...
... ..
```

If the `OPENR` statement fails because the specified file does not exist, program execution will halt with the `OPENR` statement as the current statement. Execution can not be resumed with the executive command `.CONTINUE` because it attempts to re-execute the offending `OPENR` statement, causing the same error. The remainder of the program can be executed by entering `.SKIP`, which skips over the incorrect `OPEN` statement.

Note

`.SKIP` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.SKIP [n]`

.STEP

The `.STEP` command executes one or more statements in the current program starting at the current position, stops, and returns control to the interactive mode. This command is useful in debugging programs. The optional argument n indicates the number of statements to execute. If n is omitted, a single statement is executed.

Note

`.STEP` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.STEP [n]` or `.S [n]`

.STEPOVER

The `.STEPOVER` command executes one or more statements in the current program starting at the current position, stops, and returns control to the interactive mode. Unlike `.STEP`, if `.STEPOVER` executes a statement that calls another routine, the called routine runs until it ends before control returns to interactive mode. That is, a statement calling another routine is treated as a single statement.

The optional argument `n` indicates the number of statements to execute. If `n` is omitted, a single statement (or called routine) is executed.

Note

`.STEPOVER` is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

`.STEPOVER [n]` or `.SO [n]`

.TRACE

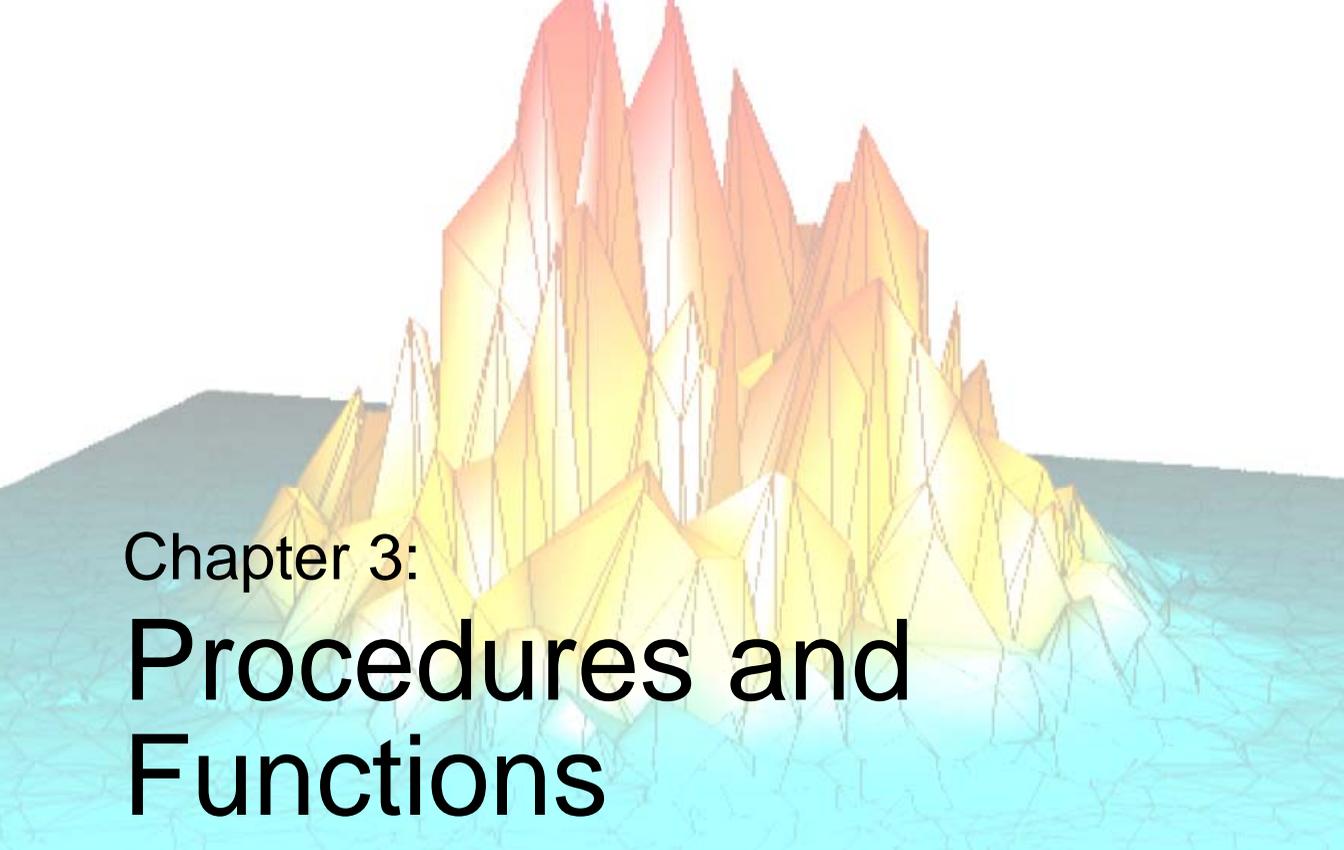
The .TRACE command continues execution of a program that has stopped because of an error, a stop statement, or a keyboard interrupt.

Note

.TRACE is an executive command. Executive commands can only be used at the IDL command prompt, not in programs.

Syntax

.TRACE



Chapter 3:

Procedures and Functions

A_CORRELATE

The A_CORRELATE function computes the autocorrelation $P_x(L)$ or autocovariance $R_x(L)$ of a sample population X as a function of the lag L .

$$P_x(L) = P_x(-L) = \frac{\sum_{k=0}^{N-L-1} (x_k - \bar{x})(x_{k+L} - \bar{x})}{\sum_{k=0}^{N-1} (x_k - \bar{x})^2}$$

$$R_x(L) = R_x(-L) = \frac{1}{N} \sum_{k=0}^{N-L-1} (x_k - \bar{x})(x_{k+L} - \bar{x})$$

where \bar{x} is the mean of the sample population $x = (x_0, x_1, x_2, \dots, x_{N-1})$.

Note

This routine is primarily designed for use in 1-D time-series analysis. The mean is subtracted before correlating. For image processing, methods based on FFT should be used instead if more than a few tens of points exist. For example:

```
Function AutoCorrelate, X
  Temp = FFT(X, -1)
  RETURN, FFT(Temp * CONJ(Temp), 1)
END
```

This routine is written in the IDL language. Its source code can be found in the file `a_correlate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = A_CORRELATE(*X*, *Lag* [, /COVARIANCE] [, /DOUBLE])

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Lag

An n -element integer vector in the interval $[-(n-2), (n-2)]$, specifying the signed distances between indexed elements of X .

Keywords

COVARIANCE

Set this keyword to compute the sample autocovariance rather than the sample autocorrelation.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```

; Define an  $n$ -element sample population:
X = [3.73, 3.67, 3.77, 3.83, 4.67, 5.87, 6.70, 6.97, 6.40, 5.57]
; Compute the autocorrelation of X for LAG = -3, 0, 1, 3, 4, 8:
lag = [-3, 0, 1, 3, 4, 8]
result = A_CORRELATE(X, lag)
PRINT, result

```

IDL prints:

```

0.0146185  1.00000  0.810879  0.0146185  -0.325279  -0.151684

```

Version History

Introduced: 4.0

See Also

[CORRELATE](#), [C_CORRELATE](#), [M_CORRELATE](#), [P_CORRELATE](#),
[R_CORRELATE](#)

ABS

The ABS function returns the absolute value of its argument.

Syntax

Result = ABS(*X*)

Return Value

Returns the absolute value of its argument.

Arguments

X

The value for which the absolute value is desired. If *X* is of complex type, ABS returns the magnitude of the complex number:

$$\sqrt{\text{Real}^2 + \text{Imaginary}^2}$$

If *X* is of complex type, the result is returned as the corresponding floating point type. For all other types, the result has the same type as *X*. If *X* is an array, the result has the same structure, with each element containing the absolute value of the corresponding element of *X*.

ABS applied to any of the unsigned integer types results in the unaltered value of *X* being returned.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To print the absolute value of -25, enter:

```
PRINT, ABS(-25)
```

IDL prints:

```
25
```

Version History

Introduced: Original

ACOS

The ACOS function returns the angle, expressed in radians, whose cosine is X (i.e., the arc-cosine). For real input, the range of ACOS is between 0 and π .

For input of a complex number, $Z = X + iY$, the complex arccosine is given by,

$$\text{acos}(Z) = \text{acos}(B) - i \log(A + \sqrt{A^2 - 1}) \quad \text{if } Y \geq 0$$

$$\text{acos}(Z) = \text{acos}(B) + i \log(A + \sqrt{A^2 - 1}) \quad \text{if } Y < 0$$

where

$$A = 0.5 \sqrt{(X + 1)^2 + Y^2} + 0.5 \sqrt{(X - 1)^2 + Y^2}$$

$$B = 0.5 \sqrt{(X + 1)^2 + Y^2} - 0.5 \sqrt{(X - 1)^2 + Y^2}$$

The separation of the two formulas at $Y = 0$ takes into account the branch-cut discontinuity along the real axis from $-\infty$ to -1 and $+1$ to $+\infty$, and ensures that $\cos(\text{acos}(Z))$ is equal to Z . For reference, see formulas 4.4.37-39 in Abramowitz, M. and Stegun, I.A., 1964: Handbook of Mathematical Functions (Washington: National Bureau of Standards).

Syntax

Result = ACOS(*X*)

Return Value

Returns the angle, expressed in radians, whose cosine is X (i.e., the arc-cosine).

Arguments

X

The cosine of the desired angle. For real input, X should be in the range -1 to $+1$. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. If X is an array, the result has the same structure, with each element containing the arc-cosine of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Find the angle whose cosine is 0.707 and print the result in degrees by entering:

```
PRINT, 180/!PI*ACOS(0.707)
IDL prints:
45.0086
```

Find the complex arccosine of $2 + i$ and print the result by entering:

```
PRINT, ACOS(COMPLEX(2,1))
IDL prints:
( 0.507356, -1.46935)
```

See the ATAN function for an example of visualizing the complex arccosine.

Version History

Introduced: Original

PHASE keyword: 5.6

See Also

[COS](#), [COSH](#), [ASIN](#), [SIN](#), [SINH](#), [ATAN](#), [TAN](#), [TANH](#)

ADAPT_HIST_EQUAL

The ADAPT_HIST_EQUAL function performs adaptive histogram equalization, a form of automatic image contrast enhancement. The algorithm is described in Pizer et. al., “Adaptive Histogram Equalization and its Variations.”, Computer Vision, Graphics and Image Processing, 39:355-368. Adaptive histogram equalization involves applying contrast enhancement based on the local region surrounding each pixel. Each pixel is mapped to an intensity proportional to its rank within the surrounding neighborhood. This method of automatic contrast enhancement has proven to be broadly applicable to a wide range of images and to have demonstrated effectiveness.

This routine is written in the IDL language. Its source code can be found in the file `adapt_hist_equal.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = ADAPT_HIST_EQUAL (Image [, CLIP=value] [, FCN=vector]
    [, NREGIONS=nregions] [, TOP=value] )
```

Return Value

The result of the function is a byte image with the same dimensions as the input image parameter.

Arguments

Image

A two-dimensional array representing the image for which adaptive histogram equalization is to be performed. This parameter is interpreted as unsigned 8-bit data, so be sure that the input values are properly scaled into the range of 0 to 255.

Keywords

CLIP

Set this keyword to a nonzero value to clip the histogram by limiting its slope to the given CLIP value, thereby limiting contrast. For example, if CLIP is set to 3, the slope of the histogram is limited to 3. By default, the slope and/or contrast is not

limited. Noise over-enhancement in nearly homogeneous regions is reduced by setting this parameter to values larger than 1.0.

FCN

Set this keyword to the desired cumulative probability distribution function in the form of a 256 element vector. If omitted, a linear ramp, which yields equal probability bins results. This function is later normalized, so magnitude is inconsequential, though should increase monotonically.

NREGIONS

Set this keyword to the size of the overlapped tiles, as a fraction of the largest dimensions of the image size. The default is 12, which makes each tile 1/12 the size of the largest image dimension.

TOP

Set this keyword to the maximum value of the scaled output array. The default is 255.

Examples

The following code snippet reads a data file in the `examples/data` subdirectory of the IDL distribution containing a cerebral angiogram, and then displays both the original image and the adaptive histogram equalized image:

```
OPENR, 1, FILEPATH('cereb.dat', $
    SUBDIRECTORY=['examples', 'data'])

;Image size = 512 x 512
a = BYTARR(512,512, /NOZERO)

;Read it
READU, 1, a
CLOSE, 1

; Reduce size of image for comparison
a = CONGRID(a, 256,256)

;Show original
TVSCL, a, 0

;Show processed
TV, ADAPT_HIST_EQUAL(a, TOP=!D.TABLE_SIZE-1), 1
```

Version History

Introduced: 5.3

See Also

[H_EQ_CT](#), [H_EQ_INT](#), [HIST_2D](#), [HIST_EQUAL](#), [HISTOGRAM](#)

ALOG

The ALOG function returns the natural logarithm of X .

For input of a complex number, $Z = X + iY$, the complex number can be rewritten as $Z = R \exp(i\theta)$, where $R = \text{abs}(Z)$ and $\theta = \text{atan}(y,x)$. The complex natural log is then given by,

$$\text{alog}(Z) = \text{alog}(R) + i\theta$$

In the above formula, the use of the two-argument arctangent separates the solutions at $Y = 0$ and takes into account the branch-cut discontinuity along the real axis from $-\infty$ to 0 , and ensures that $\exp(\text{alog}(Z))$ is equal to Z . For reference, see formulas 4.4.1-3 in Abramowitz, M. and Stegun, I.A., 1964: *Handbook of Mathematical Functions* (Washington: National Bureau of Standards).

Syntax

Result = ALOG(X)

Return Value

Returns the natural logarithm of X .

Arguments

X

The value for which the natural log is desired. For real input, X should be greater than or equal to zero. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. If X is an array, the result has the same structure, with each element containing the natural log of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS,

TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Find the natural logarithm of 2 and print the result by entering:

```
PRINT, ALOG(2)
IDL prints:
0.693147
```

Find the complex natural log of $\sqrt{2} + i\sqrt{2}$ and print the result by entering:

```
PRINT, ALOG(COMPLEX(sqrt(2), sqrt(2)))
IDL prints:
( 0.693147, 0.785398)
```

Note

The real part of the result is just $\text{ALOG}(2)$ and the imaginary part gives the angle (in radians) of the complex number relative to the real axis.

See the ATAN function for an example of visualizing the complex natural log.

Version History

Introduced: Original

See Also

[ALOG10](#), [ATAN](#)

ALOG10

The ALOG10 function returns the logarithm to the base 10 of X .

For input of a complex number, $Z = X + iY$, the complex number can be rewritten as $Z = R \exp(iq)$, where $R = \text{abs}(Z)$ and $q = \text{atan}(y,x)$. The complex log base 10 is then given by,

$$\text{alog10}(Z) = \text{alog10}(R) + i q/\text{alog}(10)$$

In the above formula, the use of the two-argument arctangent separates the solutions at $Y = 0$ and takes into account the branch-cut discontinuity along the real axis from $-\infty$ to 0, and ensures that $10^{\text{alog10}(Z)}$ is equal to Z . For reference, see formulas 4.4.1-3 in Abramowitz, M. and Stegun, I.A., 1964: *Handbook of Mathematical Functions* (Washington: National Bureau of Standards).

Syntax

Result = ALOG10(X)

Return Value

Returns the logarithm to the base 10 of X .

Arguments

X

The value for which the base 10 log is desired. For real input, X should be greater than or equal to zero. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. If X is an array, the result has the same structure, with each element containing the base 10 log of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS,

TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Find the base 10 logarithm of 100 and print the result by entering:

```
PRINT, ALOG10(100)
IDL prints:
2.00000
```

See the ATAN function for an example of visualizing the complex logarithm.

Version History

Introduced: Original

See Also

[ALOG](#), [ATAN](#)

AMOEBA

The AMOEBA function performs multidimensional minimization of a function $Func(x)$, where x is an n -dimensional vector, using the downhill simplex method of Nelder and Mead, 1965, *Computer Journal*, Vol 7, pp 308-313.

The downhill simplex method is not as efficient as Powell's method, and usually requires more function evaluations. However, the simplex method requires only function evaluations—not derivatives—and may be more reliable than Powell's method.

This routine is written in the IDL language. Its source code can be found in the file `amoeba.pro` in the `lib` subdirectory of the IDL distribution. AMOEBA is based on the routine `amoeba` described in section 10.4 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = AMOEBA( Ftol [, FUNCTION_NAME=string]
                [, FUNCTION_VALUE=variable] [, NCALLS=value] [, NMAX=value]
                [, P0=vector, SCALE=vector | , SIMPLEX=array] )
```

Return Value

If the minimum is found, AMOEBA returns an n -element vector corresponding to the function's minimum value. If a minimum within the given tolerance is not found within the specified number of iterations, AMOEBA returns a scalar value of -1. Results are returned with the same precision (single- or double-precision floating-point) as is returned by the user-supplied function to be minimized.

Arguments

Ftol

The fractional tolerance to be achieved in the function value—that is, the fractional decrease in the function value in the terminating step. If the function you supply returns a single-precision result, *Ftol* should never be less than your machine's floating-point precision—the value contained in the EPS field of the structure returned by the `MACHAR` function. If the function you supply returns a double-precision floating-point value, *Ftol* should not be less than your machine's double-precision floating-point precision. See [MACHAR](#) for details.

Keywords

FUNCTION_NAME

Set this keyword equal to a string containing the name of the function to be minimized. If this keyword is omitted, AMOEBA assumes that an IDL function named “FUNC” is to be used.

The function to be minimized must be written as an IDL function and compiled prior to calling AMOEBA. This function must accept an n -element vector as its only parameter and return a scalar single- or double precision floating-point value as its result.

See the *Example* section below for an example function.

FUNCTION_VALUE

Set this keyword equal to a named variable that will contain an $(n+1)$ -element vector of the function values at the simplex points. The first element contains the function minimum.

NCALLS

Set this keyword equal to a named variable that will contain a count of the number of times the function was evaluated.

NMAX

Set this keyword equal to a scalar value specifying the maximum number of function evaluations allowed before terminating. The default is 5000.

P0

Set this keyword equal to an n -element single- or double-precision floating-point vector specifying the initial starting point. Note that if you specify P0, you must also specify SCALE.

For example, in a 3-dimensional problem, if the initial guess is the point [0,0,0], and you know that the function’s minimum value occurs in the interval:

$$-10 < x[0] < 10, \quad -100 < x[1] < 100, \quad -200 < x[2] < 200,$$

specify: P0=[0,0,0] and SCALE=[10, 100, 200].

Alternately, you can omit P0 and SCALE and specify SIMPLEX.

SCALE

Set this keyword equal to a scalar or n -element vector containing the problem's characteristic length scale for each dimension. SCALE is used with P0 to form an initial $(n+1)$ point simplex. If all dimensions have the same scale, set SCALE equal to a scalar.

If SCALE is specified as a scalar, the function's minimum lies within a distance of SCALE from P0. If SCALE is an N -dimensional vector, the function's minimum lies within the $Ndim+1$ simplex with the vertices P0, $P0 + [1,0,\dots,0] * SCALE$, $P0 + [0,1,0,\dots,0] * SCALE$, ..., and $P0+[0,0,\dots,1] * SCALE$.

SIMPLEX

Set this keyword equal to an n by $n+1$ single- or double-precision floating-point array containing the starting simplex. After AMOEBA has returned, the SIMPLEX array contains the simplex enclosing the function minimum. The first point in the array, SIMPLEX[*], corresponds to the function's minimum. This keyword is ignored if the P0 and SCALE keywords are set.

Examples

Use AMOEBA to find the slope and intercept of a straight line that fits a given set of points, minimizing the maximum error. The function to be minimized (FUNC, in this case) returns the maximum error, given $p[0] = \text{intercept}$, and $p[1] = \text{slope}$.

```

; First define the function FUNC:
FUNCTION FUNC, P
COMMON FUNC_XY, X, Y
RETURN, MAX(ABS(Y - (P[0] + P[1] * X)))
END

; Put the data points into a common block so they are accessible to
; the function:
COMMON FUNC_XY, X, Y

; Define the data points:
X = FINDGEN(17)*5
Y = [ 12.0, 24.3, 39.6, 51.0, 66.5, 78.4, 92.7, 107.8, $
      120.0, 135.5, 147.5, 161.0, 175.4, 187.4, 202.5, 215.4, 229.9]

; Call the function. Set the fractional tolerance to 1 part in
; 10^5, the initial guess to [0,0], and specify that the minimum
; should be found within a distance of 100 of that point:
R = AMOEBA(1.0e-5, SCALE=1.0e2, P0 = [0, 0], FUNCTION_VALUE=fval)

```

```
; Check for convergence:
IF N_ELEMENTS(R) EQ 1 THEN MESSAGE, 'AMOEBA failed to converge'

; Print results:
PRINT, 'Intercept, Slope:', r, $
      'Function value (max error): ', fval[0]
```

IDL prints:

```
Intercept, Slope:      11.4100      2.72800
Function value:       1.33000
```

Version History

Introduced: 5.0

See Also

[DFPMIN](#), [POWELL](#), [SIMPLEX](#)

ANNOTATE

The ANNOTATE procedure starts an IDL widget program that allows you to interactively annotate images and plots with text and drawings. Drawing objects include lines, arrows, polygons, rectangles, circles, and ellipses. Annotation files can be saved and restored, and annotated displays can be written to TIFF or PostScript files. The Annotation widget will work on any IDL graphics window or draw widget.

This routine is written in the IDL language. Its source code can be found in the file `annotate.pro` in the `lib` subdirectory of the IDL distribution.

Using the Annotation Widget

Before calling the Annotation widget, plot or display your data in an IDL graphics window or draw widget. Unless you specify otherwise (using the `DRAWABLE` or `WINDOW` keywords), annotations will be made in the current graphics window.

For information on using the Annotation widget, click on the widget's "Help" button.

Syntax

```
ANNOTATE [, COLOR_INDICES=array] [, DRAWABLE=widget_id | ,  
         WINDOW=index] [, LOAD_FILE=filename] [, /TEK_COLORS]
```

Arguments

None.

Keywords

COLOR_INDICES

An array of color indices from which the user can choose colors. For example, to allow the user to choose 10 colors, spread evenly over the available indices, set the keyword as follows:

```
COLOR_INDICES = INDGEN(10) * (!D.N_COLORS-1) / 9
```

If neither `TEK_COLORS` or `COLOR_INDICES` are specified, the default is to load 10 colors, evenly distributed over those available.

DRAWABLE

The widget ID of the draw widget for the annotations. Do not set both `DRAWABLE` and `WINDOW`. If neither `WINDOW` or `DRAWABLE` are specified, the current window is used.

LOAD_FILE

The name of an annotation format file to load after initialization.

TEK_COLORS

Set this keyword and the Tektronix color table is loaded starting at color index `TEK_COLORS(0)`, with `TEK_COLORS(1)` color indices. The Tektronix color table contains up to 32 distinct colors suitable for graphics. If neither `TEK_COLORS` or `COLOR_INDICES` are specified, the default is to load 10 colors, evenly distributed over those available.

WINDOW

The window index number of the window to receive the annotations. Do not set both `DRAWABLE` and `WINDOW`. If neither `WINDOW` or `DRAWABLE` are specified, the current window is used.

Examples

```
; Output an image in the current window:
TVSCL, HANNING(300,200)
; Annotate it:
ANNOTATE
```

Version History

Introduced: Pre 4.0

See Also

[PLOTS, XYOUTS](#)

APP_USER_DIR

The APP_USER_DIR function provides access to the IDL *application user directory*. The application user directory is a location where IDL, and applications written in IDL, can store user-specific data that will persist between IDL sessions. For example, the IDL iTools store user-specified preferences, styles, and macros in the application user directory.

The application user directory is created automatically by IDL as a subdirectory (named `.idl`) of the user's *home directory*. (For additional information on the location of the user's home directory on different platforms, see “HOME” in [Chapter 1](#), “Environment Variables Used by IDL” in the *Using IDL* manual.)

To prevent unrelated applications from encountering each other's files, the `.idl` directory is organized into subdirectories with names specified by the application author. For example, files used by applications created by RSI are stored in subdirectories of the `.idl` directory named `rsi` and `idl`. Inside each author's subdirectory, files are organized by application and (optionally) by a combination of attributes of the IDL version that creates the directory.

Note

The organization of the `.idl` directory is opaque to both users and authors of IDL applications. Application authors are expected to manage the directory exclusively using the APP_USER_DIR function, and application users generally do not need to manually modify the files it contains.

APP_USER_DIR simplifies cross-platform application development by providing a well-defined location for IDL applications to store their resource files, regardless of the platform or version of IDL. The uniform organization it enforces is also a benefit for IDL users, since it makes it easier for them to understand the meaning and importance of the files found in their `.idl` directory.

Directories created by APP_USER_DIR have the following features:

- Each author of IDL applications has a unique area underneath the `.idl` directory. Different authors do not share space with each other, or with RSI.
- By default, an application directory is shared by all instances of the application. The application author can use RESTRICT keywords to specify that the directory only be used by a specified version of the application, or when running on a version of IDL with specified attributes such as release, operating system, hardware platform, *etc.* See “RESTRICT Keywords” on page 110 for additional details.

- A README file is automatically generated for each author and application directory, following a standard format. The application author is required to supply the body text for the application README file. (Text for the author README file is optional, but recommended.) The text is expected to explain the purpose of the directory, and supply contact information for the author. Each README file has a version associated with it, allowing APP_USER_DIR to automatically update the file as necessary.

APP_USER_DIR performs the following operations, in order:

1. If the specified author directory does not exist, it is created.
2. If the specified author directory does not have a standard README file, it is created. If there is an existing README file with a version number that is lower than the specified number, the file is replaced.
3. If the specified application subdirectory does not exist within the author directory, it is created.
4. If the specified application directory does not have a standard README file, it is created. If there is an existing README file with a version number that is lower than the specified number, the file is replaced.
5. The path to the application directory is returned as the value of the APP_USER_DIR function.

Other than the README file, the contents of an application directory are entirely up to the application author. The APP_USER_DIR function does not manage the content, only its location. The FILEPATH function can be used to construct filenames within the application directory, and IDL's many input/output and file manipulation routines can be used to create and manage application specific content within it.

Syntax

```
Result = APP_USER_DIR(AuthorDirname, AuthorDesc, AppDirname, AppDesc,
  AppReadmeText, AppReadmeVersion
  [, AUTHOR_README_TEXT=string array]
  [, AUTHOR_README_VERSION=integer]
  [, RESTRICT_APPVERSION=string] [, /RESTRICT_ARCH]
  [, /RESTRICT_FAMILY | /RESTRICT_OS]
  [, /RESTRICT_FILE_OFFSET_BITS] [, /RESTRICT_IDL_RELEASE]
  [, /RESTRICT_MEMORY_BITS] )
```

Return Value

Returns a string containing the path to the directory to be used by the calling application.

Note

The directory and all associated README files are created if they do not exist. If the README files exist, they will be updated if the values of *AppReadmeVersion* and the `AUTHOR_README_VERSION` keyword are greater than those of the existing README files.

Arguments

AuthorDirname

A string specifying the name of the *author directory* to be used by the calling application. The author directory names `rsi` and `idl` are reserved for use by IDL and IDL applications created by RSI. Other authors should be careful to pick a unique name unlikely to conflict with others. (See “[Advice for Library Authors](#)” in Chapter 5 of the *Building IDL Applications* manual for advice on selecting a unique prefix for a library; the same advice applies when selecting an author directory name.)

In choosing *AuthorDirname*, be aware of the following:

1. *AuthorDirname* is case insensitive, regardless of the operating system platform. `APP_USER_DIR` will convert the name to lower case before creating the directory or returning its path string.
2. You should attempt to keep the name short, to minimize the chance that the full directory name is longer than the maximum supported length on the current platform.
3. The following characters are automatically converted to underscores (`_`) when creating the directory name: `-` `)` `(` `.` `:` `>` `<` as well as *space* and *tab*.

AuthorDesc

A string containing a one-line description of the author in human-readable form. This string is used for the header of the README file within the author directory. For example, the *AuthorDesc* for IDL applications written by RSI is “Research Systems, Inc.”

AppDirname

A string specifying the name of the application directory to be used by the calling application. For example, the IDL *iTools* application uses the application directory name “*itools*”.

Note

`APP_USER_DIR` modifies the *AppDirname* string you specify in the same way it modifies the *AuthorDirname* string. See [AuthorDirname](#), above.

AppDesc

A string giving a brief description of the application in human-readable form. This string is used for the header of the README file within the application directory. For example, the *AppDesc* for the IDL *iTools* application is “IDL Intelligent Tools (*iTools*)”.

AppReadmeText

A string or string array containing the text used as the body of the README file within the application directory. The README text is for the benefit of IDL users who are trying to understand what the directory is for, and is expected to supply the following information:

- The identity of the author, and author contact information.
- A brief description of what the application is, and what it does.
- Any special rules governing if and when the user is allowed to delete the directory and its contents.

AppReadmeVersion

An integer specifying the version number to be associated with the README file. *AppReadmeVersion* must be greater than zero.

Initially, *AppReadmeVersion* should be set to one. Every time you modify the text supplied as the *AppReadmeText* argument, you should increment the value of *AppReadmeVersion* by one. `APP_USER_DIR` uses this information to ensure that existing README files are automatically updated to the latest version.

Note

The application README file version is different from, and unrelated to, the application version (as specified via the `RESTRICT_APPVERSION` keyword). The README version applies only to the contents of the README file. You might

want to change the text in a README file without changing the version of the application; perhaps to clarify existing information or update your contact information.

Keywords

AUTHOR_README_TEXT

Set the `AUTHOR_README_TEXT` keyword equal to a string or string array to supply README text for the author directory. Authors with multiple IDL applications should consider providing application-independent contact information here.

Note

`APP_USER_DIR` requires you to supply text for the body of the application directory README file. You are not required to supply text for the author directory README file, but supplying such text is strongly recommended.

If you include this keyword, use the `AUTHOR_README_VERSION` keyword to supply a version number for the README file.

AUTHOR_README_VERSION

Set this keyword equal to an integer that specifies the version number for the author README file. The specified value must be greater than zero.

Initially, `AUTHOR_README_VERSION` should be set equal to one. Every time you modify the text supplied as the `AUTHOR_README_TEXT` keyword, you should increment the value of `AUTHOR_README_VERSION` by one. `APP_USER_DIR` uses this information to ensure that existing author README files are automatically updated to the latest version. If `AUTHOR_README_VERSION` is not supplied, a default version of 1 is used. However, RSI recommends that you explicitly specify this keyword whenever `AUTHOR_README_TEXT` is used.

Note

The author README file version is different from, and unrelated to, the application version (as specified via the `RESTRICT_APPVERSION` keyword). The README version applies only to the contents of the README file. You might want to change the text in a README file without changing the version of the application; perhaps to clarify existing information or update your contact information.

RESTRICT Keywords

By default, APP_USER_DIR creates an application user directory that will be shared by *all* instances of the application. You can use RESTRICT keywords to specify that the directory only be used by a specified version of the application, or when running on a version of IDL with specified attributes such as release, operating system, hardware platform, *etc.* The RESTRICT keywords allow you to ensure that instances of your application running in different IDL, hardware, and operating system environments store their user data separately, when appropriate.

In deciding whether to use the RESTRICT keywords for your application, you should consider the following issues:

Network Installations

In many environments, user home directories are kept on network accessible devices, and are mounted on many different systems. These systems may be running different IDL versions, different operating systems, and different hardware.

The advantages of the networked home directory are easier system administration, and the fact that no matter which computer you use, your files are always available without the need to copy them between systems. APP_USER_DIR is designed to work in such an environment. Concerning yourself with such heterogeneous network environments may seem like overkill if you are using a system with its own dedicated home directory, but you should remember that your users may be using a highly networked environment, and as such your application needs to take these issues into consideration.

Application Version Compatibility

Experience shows that many application authors do not plan adequately for cross-version compatibility. This lack of planning may cause few problems at first, but as new versions of the application are released, incompatibilities surface. Consider the following when designing your application to simplify later upgrades:

1. Can you imagine that a future version of your application might want to structure its data in a different way, or that changes for a new version might confuse older versions that also use the same data? If so, you should specify the RESTRICT_APPVERSION keyword to give your application a version. If in doubt, specify a version of '1.0'. You will never be required to change the version number, but you will have the flexibility to do so if your application user directories include application version information.
2. Carefully consider how platform-specific your application is, and apply any of the other RESTRICT keywords that are necessary.

3. If versioning is important to your application, you will eventually face the situation in which your users are upgrading to a new version. In such cases, you may want to offer them the option of migrating their configuration data from the older version. The `APP_USER_DIR_QUERY` function can be used to locate the application user directories for these older versions. Note that your application data must be designed in such a way that such migrations are possible. In particular, any versioning information needed to select the best migration candidate must be present.

RESTRICT_APPVERSION

Set this keyword to a string specifying the application version. Different application user directories will be created for different values of this keyword.

Note

The application version can be any arbitrary string. In choosing your version string, be aware that the same rules and considerations described for the *AppDirname* argument apply to `RESTRICT_APPVERSION`.

RESTRICT_ARCH

Set this keyword to specify that different application user directories be created when your application runs on systems with different hardware architectures, as reported by the `!VERSION.ARCH` system variable field.

RESTRICT_FAMILY

Set this keyword to specify that different application user directories be created when your application runs on systems running different operating system families, as reported by the `!VERSION.OS_FAMILY` system variable field.

Note

If `RESTRICT_OS` is also specified, `RESTRICT_FAMILY` is ignored and has no effect.

RESTRICT_FILE_OFFSET_BITS

Set this keyword to specify that different application user directories be created when your application runs on versions of IDL that use a different number of file offset bits, as reported by the `!VERSION.FILE_OFFSET_BITS` system variable field.

RESTRICT_IDL_RELEASE

Set this keyword to specify that different application user directories be created when your application runs under different versions of IDL, as specified by the `!VERSION.RELEASE` system variable field.

RESTRICT_MEMORY_BITS

Set this keyword to specify that different application user directories be created when your application runs on versions of IDL that use a different number of memory address bits, as reported by the `!VERSION.MEMORY_BITS` system variable field.

RESTRICT_OS

Set this keyword to specify that different application user directories be created when your application runs on systems running different operating systems, as reported by the `!VERSION.OS` system variable field.

Examples

The following function demonstrates how `APP_USER_DIR` can be used efficiently to create an application user directory for an IDL application supporting the “Amazing” Grill System (AGS), a product of the fictional Acme Widgets, Inc. We assume that each AGS application user directory should only be shared between instances of the same version of the application that are running under the same operating system family. The use of the `RESTRICT` keywords causes the AGS software to create a different application user directory for each unique combination of these two attributes. The rest of the application can call this function as many times as desired. It ensures that the necessary directory exists on the first call, and then simply returns the path string on subsequent calls:

```
FUNCTION acme_grill_config_dir

COMMON acme_grill_common, config_dir

IF (N_ELEMENTS(config_dir) NE 1) THEN BEGIN

; Increment if author_readme_text is changed
author_readme_version = 1

author_readme_text = $
    ['This is the user configuration directory for', $
    'IDL based products from Acme Widgets, Inc:', $
    '', $
    '    Acme Widgets, Inc.', $
```

```

        '    1234 Amazing Way', $
        '    Grill Valley, NV, 12345', $
        '    USA', $
        '', $
        'Thank you for using products from Acme Widgets, Inc.' ]

; Increment if app_readme_text is changed
app_readme_version = 1

app_readme_text = $
['This is the configuration directory for the', $
 'Acme "Amazing" Grill System. It is used to', $
 'remember grill settings in between grill', $
 'invocations.', $
 '', $
 'It is safe to remove this directory, as it', $
 'will be recreated on demand. Note that all', $
 'settings (e.g. smoke injection depth, juicitron', $
 'angle, etc.) will revert to their default settings.', $
 '', $
 'Thank you for using the Acme "Amazing" Grill System.']]

config_dir = APP_USER_DIR('acme', 'Acme Widgets, Inc.', $
 'acme_grill', 'The Acme "Amazing" Grill System', $
 app_readme_text, app_readme_version, $
 AUTHOR_README_TEXT=author_readme_text, $
 AUTHOR_README_VERSION=author_readme_version, $
 RESTRICT_APPVERSION='1.0', /RESTRICT_FAMILY)

ENDIF

RETURN, config_dir

END

```

Version History

Introduced: 6.1

See Also

[APP_USER_DIR_QUERY](#), [FILE_MKDIR](#), [FILE_TEST](#), [FILEPATH](#), [OPEN](#),
[!VERSION](#) system variable

APP_USER_DIR_QUERY

The APP_USER_DIR_QUERY function is used to locate existing *application user directories* previously created by the APP_USER_DIR function. An IDL application can use APP_USER_DIR_QUERY to locate the directories that were used by other (presumably older) versions of itself. A newly installed version can use this information to migrate application settings or data from another version in order to preserve the user's customizations.

To use APP_USER_DIR_QUERY, your application should call it with the same values of the *AuthorDirname* and *AppDirname* arguments, and the exact set of RESTRICT keywords, used to call APP_USER_DIR. In this configuration, APP_USER_DIR_QUERY will return the same application user directory returned by APP_USER_DIR. To search for other *related* application user directories, use the QUERY keywords to specify which of the attributes specified in the call to APP_USER_DIR should be allowed to take on any value.

As an example, suppose your application creates user application directories that depend on the version of the application and on the operating system in use. You might want to locate all directories created for a specific application version, regardless of the operating system in use when the directory was created.

Tip

The application user directory for the current system is always one of the values returned by APP_USER_DIR_QUERY. In the common example of an application that wants to migrate data from a different version, the path for the current system is not desired. Use the EXCLUDE_CURRENT keyword to suppress it from the results.

Syntax

```
Result = APP_USER_DIR_QUERY(AuthorDirname, AppDirname
    [, COUNT=variable] [, /EXCLUDE_CURRENT][, /QUERY_APPVERSION]
    [, /QUERY_ARCH] [, /QUERY_FAMILY] [, /QUERY_FILE_OFFSET_BITS]
    [, /QUERY_IDL_RELEASE] [, /QUERY_MEMORY_BITS] [, /QUERY_OS]
    [RESTRICT Keywords] )
```

Return Value

Returns a string array containing all application user directories that match the query criteria. If no matching application user directories exist, a null scalar string is returned.

Arguments

AuthorDirname

A string specifying the name of the *author directory* to be used by the calling application. For `APP_USER_DIR_QUERY` to work correctly, *AuthorDirname* must be set to the same value used in the call to `APP_USER_DIR`. See [APP_USER_DIR](#) for details.

AppDirname

A string specifying the name of the application directory to be used by the calling application. For `APP_USER_DIR_QUERY` to work correctly, *AppDirname* must be set to the same value used in the call to `APP_USER_DIR`. See [APP_USER_DIR](#) for details.

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of matching application user directories found. If no matching directories are found, the specified variable will contain the value 0 (zero).

EXCLUDE_CURRENT

By default, `APP_USER_DIR_QUERY` includes the application user directory that matches the current system (the directory that would be returned by `APP_USER_DIR`) in *Result*. Set this keyword to exclude the directory returned by `APP_USER_DIR` from the *Result*.

RESTRICT Keywords

`APP_USER_DIR_QUERY` accepts the same RESTRICT keywords documented for [APP_USER_DIR](#). For `APP_USER_DIR_QUERY` to work correctly, you must

specify the same `RESTRICT` keywords as were specified to `APP_USER_DIR`. See the documentation for `APP_USER_DIR` for full details.

There is a subtle difference in the meaning of the `RESTRICT` keywords between `APP_USER_DIR` and `APP_USER_DIR_QUERY`. In the case of `APP_USER_DIR`, specifying one of the `RESTRICT` keywords causes IDL to create a directory whose name incorporates information about the value of the specified attribute *on the current platform*. In the case of `APP_USER_DIR_QUERY`, specifying one of the `RESTRICT` keywords tells IDL that the directory name must contain information about the specified attribute; if the corresponding `QUERY` keyword is also specified, directory names that contain *any* value for the specified attribute will be matched.

QUERY Keywords

A minimally correct call to `APP_USER_DIR_QUERY` consists of specifying the same *AuthorDirname* argument, *AppDirname* argument, and `RESTRICT` keywords as the corresponding call to `APP_USER_DIR`. Such a call will return a single string containing the path returned by `APP_USER_DIR`. Using `APP_USER_DIR_QUERY` in this manner is not particularly interesting; in order to locate *other* application user directories in addition to the one that applies to the current platform, you must add one or more `QUERY` keywords.

Each `QUERY` keyword corresponds to one of the `RESTRICT` keywords. The presence of a `QUERY` keyword tells `APP_USER_DIR_QUERY` to search for application user directories that satisfy all of the `RESTRICT` keywords while allowing any value for the attribute specified by the `QUERY` keyword.

QUERY_APPVERSION

Set this keyword to match application user directories that have any value for the application version. This keyword is quietly ignored if the `RESTRICT_APPVERSION` keyword is not set.

QUERY_ARCH

Set this keyword to match application user directories that have any value for the hardware architecture. This keyword is quietly ignored if the `RESTRICT_ARCH` keyword is not set.

QUERY_FAMILY

Set this keyword to match application user directories that have any value for the operating system family. This keyword is quietly ignored if the `RESTRICT_FAMILY` keyword is not set.

QUERY_FILE_OFFSET_BITS

Set this keyword to match application user directories that have any value for the number of bits used for file offsets. This keyword is quietly ignored if the `RESTRICT_FILE_OFFSET_BITS` keyword is not set.

QUERY_IDL_RELEASE

Set this keyword to match application user directories that have any value for the IDL version string. This keyword is quietly ignored if the `RESTRICT_IDL_RELEASE` keyword is not set.

QUERY_MEMORY_BITS

Set this keyword to match application user directories that have any value for the number of bits used for memory addresses. This keyword is quietly ignored if the `RESTRICT_MEMORY_BITS` keyword is not set.

QUERY_OS

Set this keyword to match application user directories that have any value for the operating system. This keyword is quietly ignored if the `RESTRICT_OS` keyword is not set.

Example

In the discussion of the `APP_USER_DIR` function, we give the example of an IDL application supporting the “Amazing” Grill System, a product of the fictional Acme Widgets, Inc. The application user directory created for that application is dependent on two attributes:

1. Application version (`RESTRICT_APPVERSION`)
2. Operating system family (`RESTRICT_FAMILY`)

When a new version of IDL runs this application for the first time, the application code will find that its newly created application directory is empty. In such a situation, the application might wish to offer the user a choice between starting over with the application defaults or migrating settings and data from an older version of the application.

The following call to `APP_USER_DIR_QUERY` can be used to locate application user directories that are for other versions of the application, but for the same operating system family. The `QUERY_APPVERSION` keyword causes the function

to match all application versions, while the `EXCLUDE_CURRENT` keyword tells it to exclude the application user directory for the current system:

```
old_appdirs = APP_USER_DIR_QUERY('acme', 'acme_grill', $  
    RESTRICT_APPVERSION='1.0', /RESTRICT_FAMILY, $  
    /QUERY_APPVERSION, /EXCLUDE_CURRENT)
```

Note that `APP_USER_DIR_QUERY` may return more than one directory. It is up to the application to decide which one to use. Applications that intend to migrate information between versions should include information in their application user directories that will facilitate such migration.

Version History

Introduced: 6.1

See Also

[APP_USER_DIR](#), [FILE_SEARCH](#)

ARG_PRESENT

The ARG_PRESENT function is useful in user-written procedures that need to know if the lifetime of a value they are creating extends beyond the current routine's lifetime. This can be important for two reasons:

1. To avoid expensive computations that the caller is not interested in.
2. To prevent heap variable leakage that would result if the routine creates pointers or object references and assigns them to arguments that are *not* passed back to the caller.

Syntax

Result = ARG_PRESENT(*Variable*)

Return Value

Returns a nonzero value if the following conditions are met:

- The argument to ARG_PRESENT was passed as a plain or keyword argument to the current routine by its caller, and
- The argument to ARG_PRESENT is a named variable into which a value will be copied when the current routine exits.

In other words, ARG_PRESENT returns TRUE if the value of the specified variable will be passed back to the caller.

Arguments

Variable

The variable to be tested.

Example

Suppose that you are writing an IDL procedure that has the following procedure definition line:

```
PRO myproc, RET_PTR = ret_ptr
```

The intent of the RET_PTR keyword is to pass back a pointer to a new pointer heap variable. The following command could be used to avoid creating (and possibly losing) a pointer if no named variable is provided by the caller:

```
IF ARG_PRESENT(ret_ptr) THEN BEGIN
```

The commands that follow would only be executed if `ret_ptr` is supplied and will be copied into a variable in the scope of the calling routine.

Version History

Introduced: 5.0

See Also

[KEYWORD_SET](#), [N_ELEMENTS](#), [N_PARAMS](#)

ARRAY_EQUAL

The `ARRAY_EQUAL` function is a fast way to compare data for equality in situations where the index of the elements that differ are not of interest. This operation is much faster than using `TOTAL(A NE B)`, because it stops the comparison as soon as the first inequality is found, an intermediate array is not created, and only one pass is made through the data. For best speed, ensure that the operands are of the same data type.

Arrays may be compared to scalars, in which case each element is compared to the scalar. For two arrays to be equal, they must have the same number of elements. If the types of the operands differ, the type of the least precise is converted to that of the most precise, unless the `NO_TYPECONV` keyword is specified to prevent it. This function works on all numeric types, strings, pointer references, and object references. In the case of pointer and object references, `ARRAY_EQUAL` compares the *references* (which are long integers), not the heap variables to which the references point.

Syntax

```
Result = ARRAY_EQUAL( Op1 , Op2 [, /NO_TYPECONV ] )
```

Return Value

Returns 1 (true) if, and only if, all elements of *Op1* are equal to *Op2*; returns 0 (false) at the first instance of inequality.

Arguments

Op1, Op2

The variables to be compared.

Keywords

NO_TYPECONV

By default, `ARRAY_EQUAL` converts operands of different types to a common type before performing the equality comparison. Set `NO_TYPECONV` to disallow this implicit type conversion. If `NO_TYPECONV` is specified, operands of different types are never considered to be equal, even if their numeric values are the same.

Examples

```
; Return True (1) if all elements of a are equal to a 0 byte:  
IF ARRAY_EQUAL(a, 0b) THEN ...  
; Return True (1) if all elements of a are equal all elements of b:  
IF ARRAY_EQUAL(a, b) THEN ...
```

Version History

Introduced: 5.4

ARRAY_INDICES

The `ARRAY_INDICES` function converts one-dimensional subscripts of an array into corresponding multi-dimensional subscripts.

This routine is written in the IDL language. Its source code can be found in the file `array_indices.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = `ARRAY_INDICES(Array, Index)`

Return Value

If *Index* is a scalar, returns a vector containing *m* dimensional subscripts, where *m* is the number of dimensions of *Array*.

If *Index* is a vector containing *n* elements, returns an (*m* x *n*) array, with each row containing the multi-dimensional subscripts corresponding to that index.

Arguments

Array

An array of any type.

Index

A scalar or vector containing the one-dimensional subscripts to be converted.

Keywords

None.

Examples

Example 1

This example finds the location of the maximum value of a random 10 by 10 array:

```
seed = 111
array = RANDOMU(seed, 10, 10)
mx = MAX(array, location)
ind = ARRAY_INDICES(array, location)
PRINT, ind, array[ind[0],ind[1]], $
    FORMAT = '(%"Value at [%d, %d] is %f")'
```

IDL prints:

```
Value at [3, 6] is 0.973381
```

Example 2

This example routine locates the highest point in the example Maroon Bells data set and places a flag at that point.

Enter the following code in the IDL editor:

```
PRO ExARRAY_INDICES

; Import Maroon Bells data.
file = FILEPATH('surface.dat', $
    SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [350, 450], $
    DATA_TYPE = 2, ENDIAN='little')

; Display data.
ISURFACE, data

; Calculate the value and one-dimensional
; array location of the highest point.
maxValue = MAX(data, maxPoint)

; Using ARRAY_INDICES to convert the one-
; dimensional array location to a two-
; dimensional array location.
maxLocation = ARRAY_INDICES(data, maxPoint)

; Print the results.
PRINT, 'Highest Point Location: ', maxLocation
PRINT, 'Highest Point Value: ', maxValue

; Create flag for the highest point.
```

```

x = maxLocation[0]
y = maxLocation[1]
z = maxValue
xFlag = [x, x, x + 50., x]
yFlag = [y, y, y + 50., y]
zFlag = [z, z + 1000., z + 750., z + 500.]

; Display flag at the highest point.
IPLOT, xFlag, yFlag, zFlag, /OVERPLOT

END

```

Save the code as `EXARRAY_INDICES.pro`, compile it and run it. The following figure displays the output of this example:

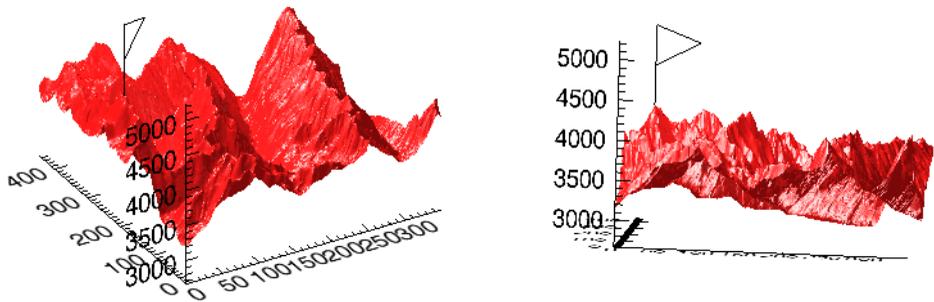


Figure 3-1: Maroon Bells Surface Plot with Flag at Highest Point Before Rotation (Left) and After Rotation (Right)

For a better view of the flag, use the Rotate tool to rotate the surface.

Version History

Introduced: 6.0

See Also

[MAX](#), [MIN](#), [WHERE](#)

ARROW

The ARROW procedure draws one or more vectors with arrow heads.

This routine is written in the IDL language. Its source code can be found in the file `arrow.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
ARROW, X0, Y0, X1, Y1 [, /DATA | , /NORMALIZED] [, HSIZE=length]  
    [, COLOR=index] [, HTHICK=value] [, /SOLID] [, THICK=value]
```

Arguments

X0, Y0

Arrays or scalars containing the coordinates of the tail end of the vector or vectors. Coordinates are in DEVICE coordinates unless otherwise specified.

X1, Y1

Arrays or scalars containing the coordinates of the arrowhead end of the vector or vectors. *X1* and *Y1* must have the save number of elements as *X0* and *Y0*.

Keywords

DATA

Set this keyword if vector coordinates are DATA coordinates.

NORMALIZED

Set this keyword if vector coordinates are NORMALIZED coordinates.

HSIZE

Use this keyword to set the length of the lines used to draw the arrowhead. The default is 1/64th the width of the display (!D.X_SIZE / 64.). If the HSIZE is positive, the value is assumed to be in device coordinate units. If HSIZE is negative, the arrowhead length is set to the vector length * ABS(HSIZE). The lines are separated by 60 degrees to make the arrowhead.

COLOR

The color of the arrow. The default is the highest color index.

HTHICK

The thickness of the arrowheads. The default is 1.0.

SOLID

Set this keyword to make a solid arrow, using polygon fills, looks better for thick arrows.

THICK

The thickness of the body. The default is 1.0.

Examples

Draw an arrow from (100,150) to (300,350) in DEVICE units:

```
ARROW, 100, 150, 300, 350
```

Draw a sine wave with arrows from the line $Y = 0$ to $\text{SIN}(X/4)$:

```
X = FINDGEN(50)
Y = SIN(x/4)
PLOT, X, Y
ARROW, X, REPLICATE(0,50), X, Y, /DATA
```

Version History

Introduced: Pre 4.0

See Also

[ANNOTATE](#), [PLOTS](#), [VELOVECT](#)

ASCII_TEMPLATE

The ASCII_TEMPLATE function presents a graphical user interface (GUI) which generates a template defining an ASCII file format. Templates are IDL structure variables that may be used when reading ASCII files with the READ_ASCII routine. See [READ_ASCII](#) for details on reading ASCII files.

This routine is written in the IDL language. Its source code can be found in the file `ascii_template.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = ASCII_TEMPLATE( [Filename] [, BROWSE_LINES=lines]  
                        [, CANCEL=variable] [, GROUP=widget_id] )
```

Return Value

Returns a template defining an ASCII file format.

Arguments

Filename

A string containing the name of a file to base the template on. If *Filename* is not specified, a dialog allows you to choose a file.

Keywords

BROWSE_LINES

Set this keyword equal to the number of lines that will be read in at a time when the “Browse” button is selected. The default is 50 lines.

CANCEL

Set this keyword to a named variable that will contain the byte value 1 if the user clicked the “Cancel” button, or 0 otherwise.

GROUP

The widget ID of an existing widget that serves as “group leader” for the ASCII_TEMPLATE graphical user interface. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

The ASCII_TEMPLATE Interface

When the ASCII_TEMPLATE function is invoked, the following dialog is displayed:

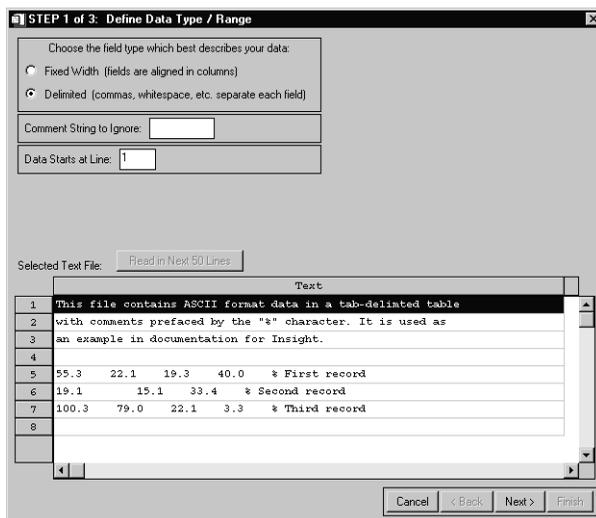


Figure 3-2: ASCII Template - Define Data Type / Range

Note

If no filename is supplied in the call to the ASCII_TEMPLATE function, a file selection dialog is displayed prior to the first ASCII_TEMPLATE screen.

The first page displays a representative sample of lines from the data file with their numbers on the left. Select the field type that best describes the data. Click the **Next** button on the bottom-right corner of the screen to move to the next page.

The second page displays the number of fields per line which is listed as three and the white space is selected for the data delimiter. Click the **Next** button on the bottom right corner of the screen to move to the next page.

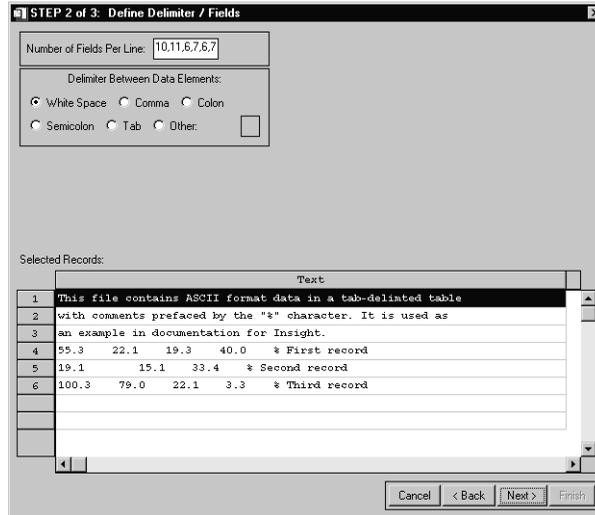


Figure 3-3: ASCII Template - Define Delimiter / Fields

The third page displays the columns in the data set which can be named and their data type specified. Name the fields by typing in the name text at the upper right of the form. Click the **Finish** button on the bottom-right corner of the screen.

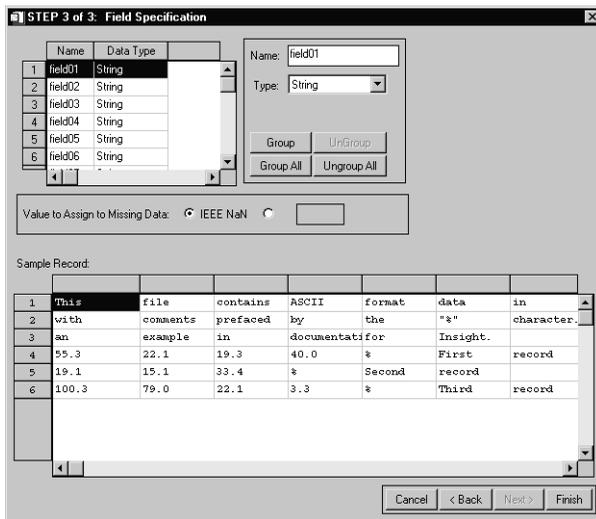


Figure 3-4: ASCII Template - Field Specification

Examples

Use the following command to generate a template structure from the file “myFile”:

```
myTemplate = ASCII_TEMPLATE(myFile)
```

Version History

Introduced: 5.0

See Also

[READ_ASCII](#), [BINARY_TEMPLATE](#)

ASIN

The ASIN function returns the angle, expressed in radians, whose sine is X (i.e., the arc-sine).

For real input, the range of ASIN is between $-\pi/2$ and $\pi/2$.

For input of a complex number, $Z = X + iY$, the complex arcsine is given by,

$$\text{asin}(Z) = \text{asin}(B) + i \text{alog}(A + \sqrt{A^2 - 1}) \quad \text{if } Y \geq 0$$

$$\text{asin}(Z) = \text{asin}(B) - i \text{alog}(A + \sqrt{A^2 - 1}) \quad \text{if } Y < 0$$

where

$$A = 0.5 \sqrt{(X + 1)^2 + Y^2} + 0.5 \sqrt{(X - 1)^2 + Y^2}$$

$$B = 0.5 \sqrt{(X + 1)^2 + Y^2} - 0.5 \sqrt{(X - 1)^2 + Y^2}$$

The separation of the two formulas at $Y = 0$ takes into account the branch-cut discontinuity along the real axis from $-\infty$ to -1 and $+1$ to $+\infty$, and ensures that $\sin(\text{asin}(Z))$ is equal to Z . For reference, see formulas 4.4.37-39 in Abramowitz, M. and Stegun, I.A., 1964: *Handbook of Mathematical Functions* (Washington: National Bureau of Standards).

Syntax

Result = ASIN(X)

Return Value

Returns the angle, expressed in radians, whose sine is X (i.e., the arc-sine).

Arguments

X

The sine of the desired angle. For real input, X should be in the range -1 to $+1$. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. If X is an array, the result has the same structure, with each element containing the arcsine of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Find the angle whose sine is 0.707 and print the result in degrees by entering:

```
PRINT, 180/!PI*ASIN(0.707)
IDL prints:
44.9913
```

Find the complex arcsine of $2 + i$ and print the result by entering:

```
PRINT, ASIN(COMPLEX(2,1))
IDL prints:
(      1.06344,      1.46935)
```

See the ATAN function for an example of visualizing the complex arcsine.

Version History

Introduced: Original

See Also

[ACOS](#), [COS](#), [COSH](#), [SIN](#), [SINH](#), [ATAN](#), [TAN](#), [TANH](#)

ASSOC

The ASSOC function associates an array structure with a file. It provides a basic method of random access input/output in IDL.

Note

Unformatted data files generated by FORTRAN programs under UNIX contain an extra long word before and after each logical record in the file. ASSOC does not interpret these extra bytes but considers them to be part of the data. This is true even if the F77_UNFORMATTED keyword is specified in the **OPEN** statement.

Therefore, ASSOC should not be used with such files. Instead, such files should be processed using **READU** and **WRITEU**. An example of using IDL to read such data is given in “Using Unformatted Input/Output” in Chapter 11 of the *Building IDL Applications* manual.

Note

Associated file variables cannot be used for output with files opened using the **COMPRESS** keyword to **OPEN**. This is due to the fact that it is not possible to move the current file position backwards in a compressed file that is currently open for writing. ASSOC is allowed with compressed files opened for input only. However, such operations may be slow due to the large amount of work required to change the file position in a compressed file.

Effective use of ASSOC requires the ability to rapidly position the file to arbitrary positions. In general, files that require random access may not be good candidates for compression. If this is necessary however, such files can be processed using **READU** and **WRITEU**.

Syntax

Result = ASSOC(*Unit*, *Array_Structure* [, *Offset*] [, **PACKED**])

Return Value

Returns a value that when assigned to a variable, stores the association between an array structure and a file in an *associated variable*. This variable provides a means of mapping a file into vectors or arrays of a specified type and size.

Arguments

Unit

The IDL file unit to associate with *Array_Structure*.

Array_Structure

An expression of the data type and structure to be associated with *Unit* are taken from *Array_Structure*. The actual value of *Array_Structure* is not used.

Offset

The offset in the file to the start of the data in the file, in bytes.

Keywords

PACKED

When ASSOC is applied to structures, the default action is to map the actual definition of the structure for the current machine, including any holes required to properly align the fields. (IDL uses the same rules for laying out structures as the C language). If the PACKED keyword is specified, I/O using the resulting variable instead works in the same manner as READU and WRITEU, and data is moved one field at a time and there are no alignment gaps between the fields.

Examples

Suppose that the file `images.dat` holds 5 images as 256-element by 256-element arrays of bytes. Open the file for reading and create an associated variable by entering:

```
OPENR, 1, 'images.dat' ;Open the file as file unit 1.  
A = ASSOC(1, BYTARR(256, 256)) ;Make an associated variable.
```

Now `A[0]` corresponds to the first image in the file, `A[1]` is the second element, etc. To display the first image in the file, you could enter:

```
TV, A[0]
```

The data for the first image is read and then displayed. Note that the data associated with `A[0]` is not held in memory. It is read in every time there is a reference to `A[0]`. To store the image in the memory-resident array `B`, you could enter:

```
B = A[0]
```

Note

It is also possible to refer to individual elements within an associated array directly, using multiple subscripts. See “[Multiple Subscripts With Associated File Variables](#)” in Chapter 11 of the *Building IDL Applications* manual for details and examples.

Version History

Introduced: Original

See Also

[OPEN, READU](#), “[Associated Input/Output](#)” in Chapter 11 of the *Building IDL Applications* manual.

ATAN

The ATAN function returns the angle, expressed in radians, whose tangent is X (i.e., the arc-tangent). If two parameters are supplied, the angle whose tangent is equal to Y/X is returned.

For real input, the range of ATAN is between $-\pi/2$ and $\pi/2$ for the single argument case, and between $-\pi$ and π if two arguments are given.

In the single argument case with a complex number, $Z = X + iY$, the complex arctangent is given by,

$$\text{atan}(Z) = 0.5 \text{atan}(2x, 1 - x^2 - y^2) + 0.25 i \text{alog}((x^2 + (y+1)^2)/(x^2 + (y-1)^2))$$

In the above formula, the use of the two-argument arctangent separates the solutions at $X = 0$ and takes into account the branch-cut discontinuity along the imaginary axis from $-i\infty$ to $-i$ and $+i$ to $+i\infty$, and ensures that $\tan(\text{atan}(Z))$ is equal to Z . For reference, see formulas 4.4.37-39 in Abramowitz, M. and Stegun, I.A., 1964: *Handbook of Mathematical Functions* (Washington: National Bureau of Standards).

In the two argument case with two complex numbers Zy and Zx , the complex arctangent is given by,

$$\text{atan}(Zy, Zx) = -i \text{alog}((Zx + iZy)/\text{sqrt}(Zx^2 + Zy^2))$$

In the two argument case (either real or complex), if both arguments are zero then the result is undefined.

Syntax

Result = ATAN(X [, /PHASE])

or

Result = ATAN(Y, X)

Return Value

Returns the angle, expressed in radians, whose tangent is X (i.e., the arc-tangent). If two parameters are supplied, the angle whose tangent is equal to Y/X is returned.

Arguments

X

The tangent of the desired angle. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. If X is an array, the result has the same structure, with each element containing the arctangent of the corresponding element of X.

Y

An optional argument. If this argument is supplied, ATAN returns the angle whose tangent is equal to Y/X. If both arguments are arrays, the function matches up the corresponding elements of X and Y, returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other arguments is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

Keywords

PHASE

If this keyword is set, and the argument is a complex number Z, then the complex phase angle is computed as $\text{ATAN}(\text{Imaginary}(Z), \text{Real_part}(Z))$. If this keyword is not set then the complex arctangent is computed as described above. If the argument is not complex, or if two arguments are present, then this keyword is ignored.

Tip

Using the PHASE keyword is equivalent to computing $\text{ATAN}(\text{Imaginary}(Z), \text{Real_part}(Z))$, but uses less memory and is faster.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Example

Find the angle whose tangent is 0.5 and print the result in degrees by entering:

```
PRINT, 180/!PI*ATAN(0.5)
IDL prints:
26.5651
```

Find the angle whose tangent is 0.5, taking into account that the tangent came from the ratio -0.25/-0.5:

```
PRINT, 180/!PI*ATAN(-0.25, -0.5)
IDL prints:
-153.435
```

Find the complex arccosine of $2 + i$ and print the result by entering:

```
PRINT, ATAN(COMPLEX(2,1))
IDL prints:
(      1.17810,      0.173287)
```

Create a visualization of the complex arctangent:

```
; Create a grid of complex numbers.
n = 100
x = (FINDGEN(n)-(n-1)/2.0)/(n/4)
z = DCOMPLEX(REBIN(x,n,n), REBIN(TRANSPOSE(x),n,n))

; Try any of these transcendental functions:
;   ACOS, COS, COSH, ASIN, SIN, SINH,
;   ATAN, TAN, TANH, ALOG, EXP
fn = ATAN(z)
oReal = OBJ_NEW('IDLgrSurface', FLOAT(fn), x, x, $
  COLOR=[255, 180, 0], STYLE=2)
oImag = OBJ_NEW('IDLgrSurface', IMAGINARY(fn), x, x, $
  COLOR=[0, 150, 255], STYLE=2)

; Add graphics objects to a model and rotate to nice view.
oModel = OBJ_NEW('IDLgrModel')
oModel->Add, oReal
oModel->Add, oImag
oModel->ROTATE, [0,0,1], 25
oModel->ROTATE, [1,0,0], -30
```

```
    ; Display using XOBJVIEW.  
;Block input so we can destroy objects after.  
    XOBJVIEW, oModel, /BLOCK, SCALE=1, $  
        TITLE='Complex transcendental function', $  
XSIZE=700, YSIZE=700  
    OBJ_DESTROY, oModel
```

Version History

Introduced: Original

See Also

[ACOS](#), [COS](#), [COSH](#), [SIN](#), [ASIN](#), [SINH](#), [TAN](#), [TANH](#)

AXIS

The `AXIS` procedure draws an axis of the specified type and scale at a given position. The new scale is saved for use by subsequent overplots if the `SAVE` keyword parameter is set. By default, `AXIS` draws an X axis. The `XAXIS`, `YAXIS`, and `ZAXIS` keywords can be used to select a specific axis type and position.

Syntax

```
AXIS [, X [, Y [, Z]]] [, /SAVE] [, XAXIS={0 | 1} | YAXIS={0 | 1} | ZAXIS={0 | 1 | 2 | 3}] [, /XLOG] [, /YNOZERO] [, /YLOG]
```

Graphics Keywords: [, CHARSIZE=*value*] [, CHARTHICK=*integer*] [, COLOR=*value*] [, /DATA | , /DEVICE | , /NORMAL] [, FONT=*integer*] [, /NODATA] [, /NOERASE] [, SUBTITLE=*string*] [, /T3D] [, TICKLEN=*value*] [, {X | Y | Z}CHARSIZE=*value*] [, {X | Y | Z}GRIDSTYLE=*integer*{0 to 5}] [, {X | Y | Z}MARGIN=[*left, right*] [, {X | Y | Z}MINOR=*integer*] [, {X | Y | Z}RANGE=[*min, max*] [, {X | Y | Z}STYLE=*value*] [, {X | Y | Z}THICK=*value*] [, {X | Y | Z}TICKFORMAT=*string*] [, {X | Y | Z}TICKINTERVAL=*value*] [, {X | Y | Z}TICKLAYOUT=*scalar*] [, {X | Y | Z}TICKLEN=*value*] [, {X | Y | Z}TICKNAME=*string_array*] [, {X | Y | Z}TICKS=*integer*] [, {X | Y | Z}TICKUNITS=*string*] [, {X | Y | Z}TICKV=*array*] [, {X | Y | Z}TICK_GET=*variable*] [, {X | Y | Z}TITLE=*string*] [, ZVALUE=*value*{0 to 1}]

Arguments

X, Y, and Z

Scalars giving the starting coordinates of the new axis. If no coordinates are specified, the axis is drawn in its default position as given by the `[XYZ]AXIS` keyword. When

drawing an X axis, the X coordinate is ignored, similarly the Y and Z arguments are ignored when drawing their respective axes (i.e., new axes will always point in the correct direction).

Keywords

SAVE

Set this keyword to indicate that the scaling to and from data coordinates established by the call to `AXIS` is to be saved in the appropriate axis system variable, `!X`, `!Y`, or `!Z`. If this keyword is not present, the scaling is not changed.

XAXIS

Set this keyword to draw an X axis. If the X argument *is not* present, setting `XAXIS` equal to 0 draws an axis under the plot window with the tick marks pointing up, and setting `XAXIS` equal to one draws an axis above the plot window with the tick marks pointing down. If the X argument *is* present, the X axis is positioned accordingly, and setting `XAXIS` equal to 0 or 1 causes the tick marks to point up or down, respectively.

XLOG

Set this keyword to specify a logarithmic X axis

YAXIS

Set this keyword to draw a Y axis. If the Y argument *is not* present, setting `YAXIS` equal to 0 draws an axis on the left side of the plot window with the tick marks pointing right, and setting `YAXIS` equal to one draws an axis on the right side of the plot window with the tick marks pointing left. If the Y argument *is* present, the Y axis is positioned accordingly, and setting `YAXIS` equal to 0 or 1 causes the tick marks to point right or left, respectively.

Note

The `YAXIS` keyword must be specified in order use any Y* graphics keywords. See the note under [“Graphics Keywords Accepted”](#) on page 143 for more information.

YLOG

Set this keyword to specify a logarithmic Y axis.

YNOZERO

Set this keyword to inhibit setting the minimum Y axis value to zero when the Y data are all positive and non-zero, and no explicit minimum Y value is specified (using YRANGE, or !Y.RANGE). By default, the Y axis spans the range of 0 to the maximum value of Y, in the case of positive Y data. Set bit 4 in !Y.STYLE to make this option the default.

ZAXIS

Set this keyword to draw a Z axis. If the Z argument is *not* present, setting ZAXIS has the following meanings:

- 0 = lower (front) right, with tickmarks pointing left
- 1 = lower (front) left, with tickmarks pointing right
- 2 = upper (back) left, with tickmarks pointing right
- 3 = upper (back) right, with tickmarks pointing left

If the Z argument *is* present, the Z axis is positioned accordingly, and setting ZAXIS equal to 0 or 1 causes the tick marks to point left or right, respectively.

Note that AXIS uses the 3D plotting transformation stored in the system variable field !P.T.

Note

The ZAXIS keyword must be specified in order use any Z* graphics keywords. See the note under [Graphics Keywords Accepted](#) for more information.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above.

Note

In order for the Y* or Z* graphics keywords to work with the AXIS procedure, the corresponding YAXIS or ZAXIS keyword must be specified. For example, the following code will *not* draw a title for the Y axis:

```
AXIS, YTITLE = 'Y-axis Title'
```

To use the YTITLE graphics keyword, you must specify the YAXIS keyword to AXIS:

```
AXIS, YAXIS = 0, YTITLE = 'Y-axis Title'
```

Because the `AXIS` procedure draws an X axis by default, it is not necessary to specify the `XAXIS` keyword in order to use the `X*` graphics keywords.

[CHARSIZE](#), [CHARTHICK](#), [COLOR](#), [DATA](#), [DEVICE](#), [FONT](#), [NODATA](#), [NOERASE](#), [NORMAL](#), [SUBTITLE](#), [T3D](#), [TICKLEN](#), [\[XYZ\]CHARSIZE](#), [\[XYZ\]GRIDSTYLE](#), [\[XYZ\]MARGIN](#), [\[XYZ\]MINOR](#), [\[XYZ\]RANGE](#), [\[XYZ\]STYLE](#), [\[XYZ\]THICK](#), [\[XYZ\]TICKFORMAT](#), [\[XYZ\]TICKINTERVAL](#), [\[XYZ\]TICKLAYOUT](#), [\[XYZ\]TICKLEN](#), [\[XYZ\]TICKNAME](#), [\[XYZ\]TICKS](#), [\[XYZ\]TICKUNITS](#), [\[XYZ\]TICKV](#), [\[XYZ\]TICK_GET](#), [\[XYZ\]TITLE](#), [ZVALUE](#).

Examples

The following example shows how the `AXIS` procedure can be used with normal or polar plots to draw axes through the origin, dividing the plot window into four quadrants:

```
; Make the plot, polar in this example, and suppress the X and Y
; axes using the XSTYLE and YSTYLE keywords:
PLOT, /POLAR, XSTYLE=4, YSTYLE=4, TITLE='Polar Plot', r, theta

; Draw an X axis, through data Y coordinate of 0. Because the XAXIS
; keyword parameter has a value of 0, the tick marks point down:
AXIS,0,0,XAX=0,/DATA

; Similarly, draw the Y axis through data X = 0. The tick marks
; point left:
AXIS,0,0,0,YAX=0,/DATA
```

Version History

Introduced: Original

See Also

[LABEL_DATE](#), [PLOT](#)

BAR_PLOT

The `BAR_PLOT` procedure creates a bar graph. This routine is written in the IDL language. Its source code can be found in the file `bar_plot.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
BAR_PLOT, Values [, BACKGROUND=color_index]
          [, BARNAMES=string_array] [, BAROFFSET=scalar] [, BARSPEC=scalar]
          [, BARWIDTH=value] [, BASELINES=vector] [, BASERANGE=scalar{0.0 to
          1.0}] [, COLORS=vector] [, /OUTLINE] [, /OVERPLOT] [, /ROTATE]
          [, TITLE=string] [, XTITLE=string] [, YTITLE=string]
```

Arguments

Values

A vector containing the values to be represented by the bars. Each element in *Values* corresponds to a single bar in the output.

Keywords

BACKGROUND

A scalar that specifies the color index to be used for the background color. By default, the normal IDL background color is used.

BARNAMES

A string array, containing one string label per bar. If the bars are vertical, the labels are placed beneath them. If horizontal (rotated) bars are specified, the labels are placed to the left of the bars.

BAROFFSET

A scalar that specifies the offset to be applied to the first bar, in units of “nominal bar width”. This keyword allows, for example, different groups of bars to be overplotted on the same graph. If not specified, the default offset is equal to `BARSPEC`.

BARSPACE

A scalar that specifies, in units of “nominal bar width”, the spacing between bars. For example, if `BARSPACE` is 1.0, then all bars will have one bar-width of space between them. If not specified, the bars are spaced apart by 20% of the bar width.

BARWIDTH

A floating-point value that specifies the width of the bars in units of “nominal bar width”. The nominal bar width is computed so that all the bars (and the space between them, set by default to 20% of the width of the bars) will fill the available space (optionally controlled with the `BASERANGE` keyword).

BASELINES

A vector, the same size as *Values*, that contains the base value associated with each bar. If not specified, a base value of zero is used for all bars.

BASERANGE

A floating-point scalar in the range 0.0 to 1.0, that determines the fraction of the total available plotting area (in the direction perpendicular to the bars) to be used. If not specified, the full available area is used.

COLORS

A vector, the same size as *Values*, containing the color index to be used for each bar. If not specified, the colors are selected based on spacing the color indices as widely as possible within the range of available colors (specified by `!D.N_COLORS`).

OUTLINE

If set, this keyword specifies that an outline should be drawn around each bar.

OVERPLOT

If set, this keyword specifies that the bar plot should be overplotted on an existing graph.

ROTATE

If set, this keyword indicates that horizontal rather than vertical bars should be drawn. The bases of horizontal bars are on the left, “Y” axis and the bars extend to the right.

TITLE

A string containing the main title for the bar plot.

XTITLE

A string containing the title for the X axis.

YTITLE

A string containing the title for the Y axis.

Examples

By using the overplotting capability, it is relatively easy to create stacked bar charts, or different groups of bars on the same graph.

The following example creates a two-dimensional array of 5 columns and 8 rows, and creates a plot with 5 bars, each of which is a “stacked” composite of 8 sections.

```

;Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Load color table:
LOADCT, 5

;Make axes black:
!P.COLOR=0

;Create 5-column by 8-row array:
array = INDGEN(5,8)

;Create a 2D array, equal in size to array, that has identical
;color index values across each row to ensure that the same item is
;represented by the same color in all bars:
colors = INTARR(5,8)
FOR I = 0, 7 DO colors[* ,I]=(20*I)+20

;With arrays and colors defined, create stacked bars (note that
;the number of rows and columns is arbitrary):

;Scale range to accommodate the total bar lengths:
!Y.RANGE = [0, MAX(array)]
nrows = N_ELEMENTS(array[0,*])
base = INTARR(nrows)

```

```

FOR I = 0, nrows-1 DO BEGIN
    BAR_PLOT, array[*,I], COLORS=colors[*,I], BACKGROUND=255, $
        BASELINES=base, BARWIDTH=0.75, BARSPACE=0.25, OVER=(I GT 0)
    base = array[*,I]
ENDFOR

```

;To plot each row of array as a clustered group of bars within the ;same graph, use the `BASERANGE` keyword to restrict the available ;plotting region for each set of bars, where `NCOLS` is the number of ;columns in array. (In this example, each group uses the same set ;of colors, but this could easily be changed.):

```

ncols = N_ELEMENTS(array[*,0])
FOR I = 0, nrows-1 DO BEGIN
    BAR_PLOT, array[*,I], COLORS=colors[*,I], BACKGROUND=255, $
        BARWIDTH=0.75, BARSPACE=0.25, BAROFFSET=I*(1.4*ncols), $
        OVER=(I GT 0), BASERANGE=0.12
ENDFOR

```

Version History

Introduced: Pre 4.0

See Also

[PLOT](#), [PSYM](#) Graphics Keyword

BEGIN...END

The BEGIN...END statement defines a block of statements. A block of statements is a group of statements that is treated as a single statement. Blocks are necessary when more than one statement is the subject of a conditional or repetitive statement. For more information on using BEGIN...END and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

BEGIN

statements

END | ENDIF | ENDELSE | ENDFOR | ENDREP | ENDWHILE

The END identifier used to terminate the block should correspond to the type of statement in which BEGIN is used. The following table lists the correct END identifiers to use with each type of statement.

Statement	END Identifier	Example
ELSE BEGIN	ENDELSE	IF (0) THEN A=1 ELSE BEGIN A=2 ENDELSE
FOR <i>variable=init, limit</i> DO BEGIN	ENDFOR	FOR i=1,5 DO BEGIN PRINT, array[i] ENDFOR
IF <i>expression</i> THEN BEGIN	ENDIF	IF (0) THEN BEGIN A=1 ENDIF
REPEAT BEGIN	ENDREP	REPEAT BEGIN A = A * 2 ENDREP UNTIL A GT B
WHILE <i>expression</i> DO BEGIN	ENDWHILE	WHILE ~ EOF(1) DO BEGIN READF, 1, A, B, C ENDWHILE
<i>LABEL</i> : BEGIN	END	LABEL1: BEGIN PRINT, A END
<i>case_expression</i> : BEGIN	END	CASE name OF 'Moe': BEGIN PRINT, 'Stooge' END ENDCASE
<i>switch_expression</i> : BEGIN	END	SWITCH name OF 'Moe': BEGIN PRINT, 'Stooge' END ENDSWITCH

Table 3-1: Types of END Identifiers

Note

CASE and SWITCH also have their own END identifiers. CASE should always be ended with ENDCASE, and SWITCH should always be ended with ENDSWITCH.

Version History

Introduced: Original

BESELI

The BESELI function returns the I Bessel function of order N for the argument X . The BESELI function is adapted from “SPECFUN - A Portable FORTRAN Package of Special Functions and Test Drivers”, W. J. Cody, Algorithm 715, *ACM Transactions on Mathematical Software*, Vol 19, No. 1, March 1993.

Syntax

Result = BESELI(X , N [, /DOUBLE] [, ITER=*variable*])

Return Value

If both arguments are scalars, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of X and N , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the smallest input array.

Note

If the function does not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN.

Arguments

X

A scalar or array specifying the values for which the Bessel function is required. Values for X must be in the range -709 to 709.

Note

If X is negative then N must be an integer (either positive or negative).

N

A scalar or array specifying the order of the Bessel function to calculate. Values for N can be integers or real numbers. If N is negative then it must be an integer.

Keywords

DOUBLE

Set this keyword equal to one to return a double-precision result, or to zero to return a single-precision result. The computations will always be done using double precision. The default is to return a single-precision result if both inputs are single precision, and to return a double-precision result in all other cases.

ITER

Set this keyword equal to a named variable that will contain the number of iterations performed. If the routine converged, the stored value will be equal to the order N . If X or N are arrays, ITER will contain a scalar representing the maximum number of iterations.

Note

If the routine did not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN, and ITER will contain the largest order that *would have converged* for that X value.

Examples

The following example plots the I and K Bessel functions for orders 0, 1 and 2:

```
X = FINDGEN(40)/10

;Plot I and K Bessel Functions:
PLOT, X, BESELI(X, 0), MAX_VALUE=4, $
    TITLE = 'I and K Bessel Functions'
OPLOT, X, BESELI(X, 1)
OPLOT, X, BESELI(X, 2)
OPLOT, X, BESELK(X, 0), LINESSTYLE=2
OPLOT, X, BESELK(X, 1), LINESSTYLE=2
OPLOT, X, BESELK(X, 2), LINESSTYLE=2

;Annotate plot:
xcoords = [.18, .45, .95, 1.4, 1.8, 2.4]
ycoords = [2.1, 2.1, 2.1, 1.8, 1.6, 1.4]
labels = ['!8K!X!D0', '!8K!X!D1', '!8K!X!D2', '!8I!X!D0',
        '!8I!X!D1', '!8I!X!D2']
XYOUTS, xcoords, ycoords, labels, /DATA
```

This results in the following plot:

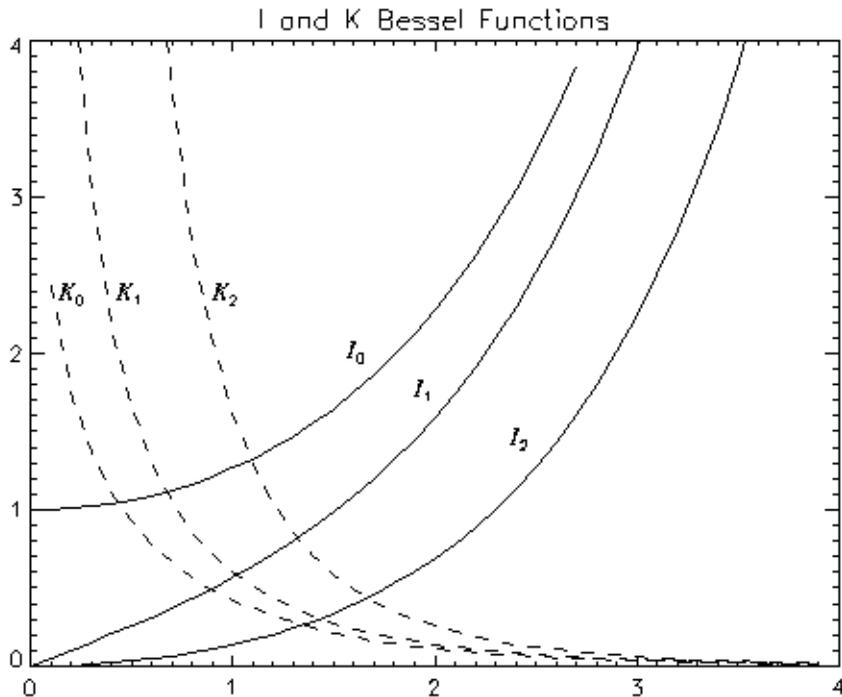


Figure 3-5: *I and K Bessel Functions.*

For an example calculating the accuracy of the Bessel function, see “[Example 2](#)” for the [BESELJ](#) routine.

Version History

Introduced: Original

DOUBLE and ITER keywords: 5.6

See Also

[BESELJ](#), [BESELK](#), [BESELY](#)

BESELJ

The BESELJ function returns the J Bessel function of order N for the argument X . The BESELJ function is adapted from “SPECFUN - A Portable FORTRAN Package of Special Functions and Test Drivers”, W. J. Cody, Algorithm 715, *ACM Transactions on Mathematical Software*, Vol 19, No. 1, March 1993.

Syntax

Result = BESELJ(X , N [, /DOUBLE] [, ITER=*variable*])

Return Value

If both arguments are scalars, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of X and N , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the smallest input array.

If X is double-precision, the result is double-precision, otherwise the result is single-precision.

Note

If the function does not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN.

Arguments

X

A scalar or array specifying the values for which the Bessel function is required. Values for X must be in the range -10^8 to 10^8 .

Note

If X is negative then N must be an integer (either positive or negative).

N

A scalar or array specifying the order of the Bessel function to calculate. Values for N can be integers or real numbers. If N is negative then it must be an integer.

Keywords

DOUBLE

Set this keyword equal to one to return a double-precision result, or to zero to return a single-precision result. The computations will always be done using double precision. The default is to return a single-precision result if both inputs are single precision, and to return a double-precision result in all other cases.

ITER

Set this keyword equal to a named variable that will contain the number of iterations performed. If the routine converged, the stored value will be equal to the order N . If X or N are arrays, ITER will contain a scalar representing the maximum number of iterations.

Note

If the routine did not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN, and ITER will contain the largest order that *would have converged* for that X value.

Examples

Example 1

The following example plots the J and Y Bessel functions for orders 0, 1, and 2:

```
X = FINDGEN(100)/10

;Plot J and Y Bessel Functions:
PLOT, X, BESELJ(X, 0), TITLE = 'J and Y Bessel Functions'
OPLOT, X, BESELJ(X, 1)
OPLOT, X, BESELJ(X, 2)
OPLOT, X, BESELY(X, 0), LINESSTYLE=2
OPLOT, X, BESELY(X, 1), LINESSTYLE=2
OPLOT, X, BESELY(X, 2), LINESSTYLE=2

;Annotate plot:
xcoords = [1, 1.66, 3, .7, 1.7, 2.65]
ycoords = [.8, .62, .52, -.42, -.42, -.42]
labels = ['!8J!X!D0', '!8J!X!D1', '!8J!X!D2', '!8Y!X!D0',
          '!8Y!X!D1', '!8Y!X!D2']
XYOUTS, xcoords, ycoords, labels, /DATA
```

This results in the following plot:

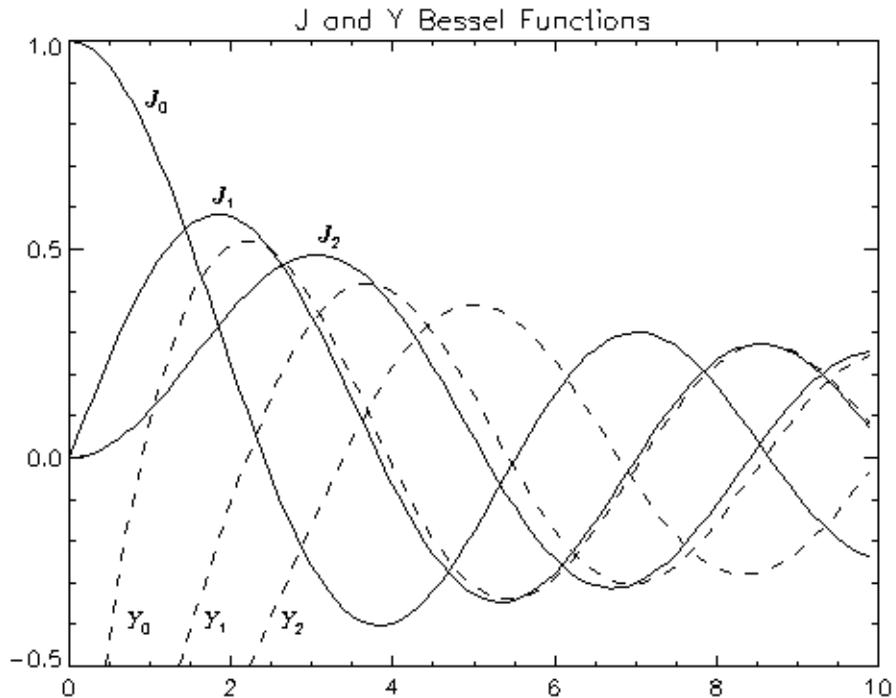


Figure 3-6: The J and Y Bessel Functions.

Example 2

Different order Bessel functions have *recurrence relationships* to each other. These relationships can be used to determine how accurately IDL is computing the Bessel functions. In the following example, the recurrence relationships for each order are set to zero and the left side of the equations are plotted. The plots show how close the left side of the equations are to zero, and therefore, how accurate IDL's computation of the Bessel functions are.

This example uses the following recurrence relationship:

$$x(J_{n-1}(x) + J_{n+1}(x)) - 2nJ_n(x) = 0$$

where $J(x)$ is the Bessel function of the first kind of order $n-1$, n , or $n+1$. (Similar recurrence relationships could be used for the other forms of the Bessel function.) Results are plotted for n equal to 1 through 6.

```

PRO AnalyzingBESELJ

; Derive x values.
x = (DINDGEN(1000) + 1.)/100.

; Initialize display window.
WINDOW, 0, TITLE = 'Bessel Functions'

; Display the first 8 orders of the Bessel function of
; the first kind.
PLOT, x, BESELJ(x, 0), /XSTYLE, /YSTYLE, $
    XTITLE = 'x', YTITLE = 'f(x)', $
    TITLE = 'Bessel Functions of the First Kind'
OOPLOT, x, BESELJ(x, 1), LINESSTYLE = 1
OOPLOT, x, BESELJ(x, 2), LINESSTYLE = 2
OOPLOT, x, BESELJ(x, 3), LINESSTYLE = 3
OOPLOT, x, BESELJ(x, 4), LINESSTYLE = 4
OOPLOT, x, BESELJ(x, 5), LINESSTYLE = 5
OOPLOT, x, BESELJ(x, 6), LINESSTYLE = 0
OOPLOT, x, BESELJ(x, 7), LINESSTYLE = 1

; Initialize display window for recurrence relations.
WINDOW, 1, XSIZE = 896, YSIZE = 512, $
    TITLE = 'Testing the Recurrence Relations'
!P.MULTI = [0, 2, 3, 0, 0]

; Initialize title variable.
nString = ['0', '1', '2', '3', '4', '5', '6', '7']

; Display recurrence relationships for order 1 to 6.
; NOTE: the results of these relationships should be
; very close to zero.
FOR n = 1, 6 DO BEGIN
    equation = x*(BESELJ(x, (n - 1)) + $
        BESELJ(x, (n + 1))) - 2.*FLOAT(n)*BESELJ(x, n)
    PLOT, x, equation, /XSTYLE, /YSTYLE, CHARSIZE = 1.5, $
        TITLE = 'n = ' + nString[n] + ': Orders of ' + $
            nString[n - 1] + ', ' + nString[n] + ', and ' + $
            nString[n + 1]
    PRINT, 'n = ' + nString[n] + ': '
    PRINT, 'minimum = ', MIN(equation)
    PRINT, 'maximum = ', MAX(equation)
ENDFOR

```

```

; Return display window back to its default setting, one
; display per window.
!P.MULTI = 0

END

```

The results for this example are shown in the following figure.

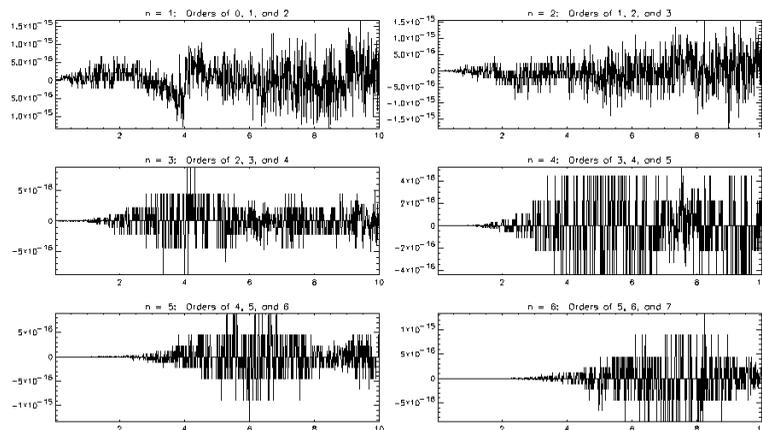


Figure 3-7: Recurrence Relationship for $J(x)$

All of these plots show that this Bessel function is calculated accurately within machine tolerance.

Version History

Introduced: Original

DOUBLE and ITER keywords: 5.6

See Also

[BESELI](#), [BESELK](#), [BESELY](#)

BESELK

The BESELK function returns the K Bessel function of order N for the argument X . The BESELK function is adapted from “SPECFUN - A Portable FORTRAN Package of Special Functions and Test Drivers”, W. J. Cody, Algorithm 715, *ACM Transactions on Mathematical Software*, Vol 19, No. 1, March 1993.

Syntax

Result = BESELK(X , N [, /DOUBLE] [, ITER=*variable*])

Return Value

If both arguments are scalars, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of X and N , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the smallest input array.

If X is double-precision, the result is double-precision, otherwise the result is single-precision.

Note

If the function does not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN.

Arguments

X

A scalar or array specifying the values for which the Bessel function is required. Values for X must be greater than or equal to zero.

N

A scalar or array specifying the order of the Bessel function to calculate. Values for N can be integers or real numbers. If N is negative then it must be an integer.

Keywords

DOUBLE

Set this keyword equal to one to return a double-precision result, or to zero to return a single-precision result. The computations will always be done using double precision. The default is to return a single-precision result if both inputs are single precision, and to return a double-precision result in all other cases.

ITER

Set this keyword equal to a named variable that will contain the number of iterations performed. If the routine converged, the stored value will be equal to the order N . If X or N are arrays, ITER will contain a scalar representing the maximum number of iterations.

Note

If the routine did not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN, and ITER will contain the largest order that *would have converged* for that X value.

Examples

The following example plots the I and K Bessel functions for orders 0, 1 and 2:

```
X = FINDGEN(40)/10

;Plot I and K Bessel Functions:
PLOT, X, BESELI(X, 0), MAX_VALUE=4, $
    TITLE = 'I and K Bessel Functions'
OPLOT, X, BESELI(X, 1)
OPLOT, X, BESELI(X, 2)
OPLOT, X, BESELK(X, 0), LINESSTYLE=2
OPLOT, X, BESELK(X, 1), LINESSTYLE=2
OPLOT, X, BESELK(X, 2), LINESSTYLE=2

;Annotate plot:
xcoords = [.18, .45, .95, 1.4, 1.8, 2.4]
ycoords = [2.1, 2.1, 2.1, 1.8, 1.6, 1.4]
labels = ['!8K!X!D0', '!8K!X!D1', '!8K!X!D2', '!8I!X!D0',
    '!8I!X!D1', '!8I!X!D2']
XYOUTS, xcoords, ycoords, labels, /DATA
```

This results in the following plot:

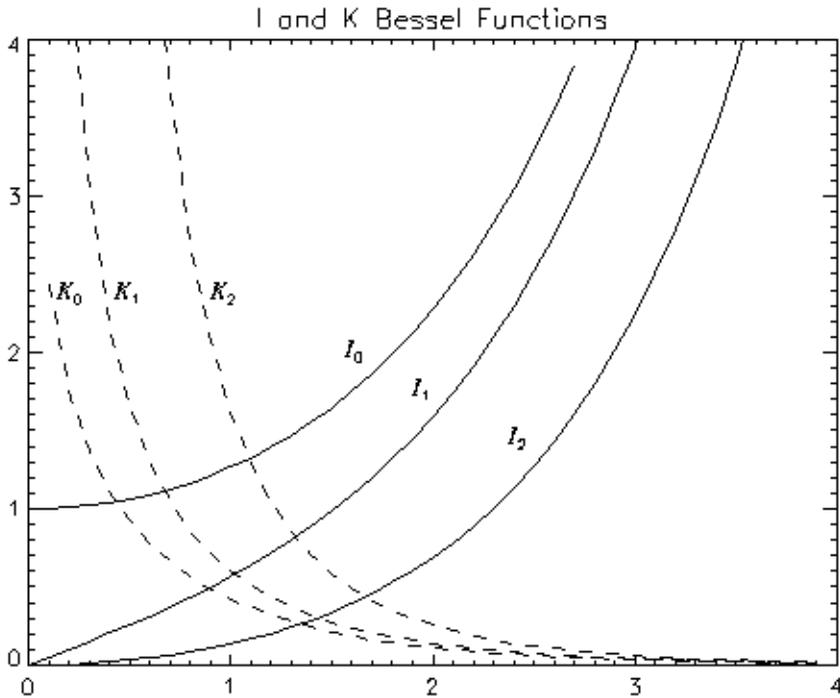


Figure 3-8: I and K Bessel Functions.

For an example calculating the accuracy of the Bessel function, see “[Example 2](#)” for the [BESELJ](#) routine.

Version History

Introduced: 5.4

DOUBLE and ITER keywords: 5.6

See Also

[BESELI](#), [BESELJ](#), [BESELY](#)

BESELY

The BESELY function returns the Y Bessel function of order N for the argument X . The BESELY function is adapted from “SPECFUN - A Portable FORTRAN Package of Special Functions and Test Drivers”, W. J. Cody, Algorithm 715, *ACM Transactions on Mathematical Software*, Vol 19, No. 1, March 1993.

Syntax

Result = BESELY(X , N [, /DOUBLE] [, ITER=*variable*])

Return Value

If both arguments are scalars, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of X and N , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the smallest input array.

If X is double-precision, the result is double-precision, otherwise the result is single-precision.

Note

If the function does not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN.

Arguments

X

A scalar or array specifying the values for which the Bessel function is required. Values for X must be in the range 0 to 10^8 .

N

A scalar or array specifying the order of the Bessel function to calculate. Values for N can be integers or real numbers. If N is negative then it must be an integer.

Keywords

DOUBLE

Set this keyword equal to one to return a double-precision result, or to zero to return a single-precision result. The computations will always be done using double precision. The default is to return a single-precision result if both inputs are single precision, and to return a double-precision result in all other cases.

ITER

Set this keyword equal to a named variable that will contain the number of iterations performed. If the routine converged, the stored value will be equal to the order N . If X or N are arrays, ITER will contain a scalar representing the maximum number of iterations.

Note

If the routine did not converge for an element of X , the corresponding element of the *Result* array will be set to the IEEE floating-point value NaN, and ITER will contain the largest order that *would have converged* for that X value.

Examples

The following example plots the J and Y Bessel functions for orders 0, 1, and 2:

```
X = FINDGEN(100)/10

;Plot J and Y Bessel Functions:
PLOT, X, BESELJ(X, 0), TITLE = 'J and Y Bessel Functions'
OPLOT, X, BESELJ(X, 1)
OPLOT, X, BESELJ(X, 2)
OPLOT, X, BESELY(X, 0), LINESSTYLE=2
OPLOT, X, BESELY(X, 1), LINESSTYLE=2
OPLOT, X, BESELY(X, 2), LINESSTYLE=2

;Annotate plot:
xcoords = [1, 1.66, 3, .7, 1.7, 2.65]
ycoords = [.8, .62, .52, -.42, -.42, -.42]
labels = ['!8J!X!D0', '!8J!X!D1', '!8J!X!D2', '!8Y!X!D0',
          '!8Y!X!D1', '!8Y!X!D2']
XYOUTS, xcoords, ycoords, labels, /DATA
```

This results in the following plot:

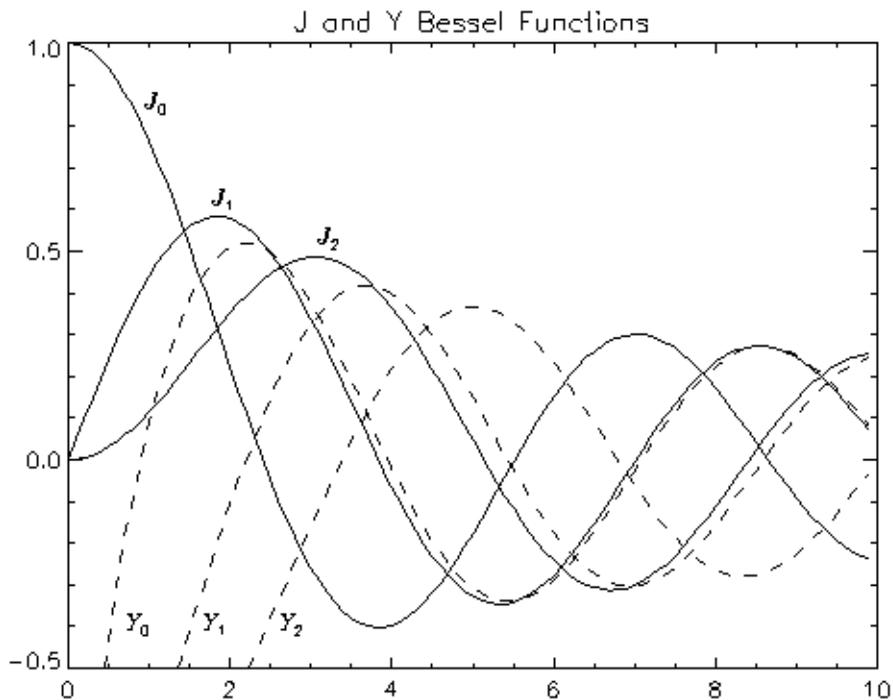


Figure 3-9: The J and Y Bessel Functions.

For an example calculating the accuracy of the Bessel function, see “[Example 2](#)” for the [BESELJ](#) routine.

Version History

Introduced: Original

DOUBLE and ITER keywords: 5.6

See Also

[BESELI](#), [BESELJ](#), [BESELK](#)

BETA

The BETA function returns the value of the beta function $B(Z, W)$. This routine is written in the IDL language. Its source code can be found in the file `beta.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = BETA( Z, W [, /DOUBLE] )
```

Return Value

If both arguments are scalar, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of Z and W , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

If both of the arguments are double-precision or if the `DOUBLE` keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

Z, W

The point at which the beta function is to be evaluated. Z and W can be scalar or array. Z or W may be complex.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the `!CPU` system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established

by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

To evaluate the beta function at the point (1.0, 1.1) and print the result:

```
PRINT, BETA(1.0, 1.1)
```

IDL prints:

```
0.909091
```

The exact solution is:

$$((1.00 * .95135077) / (1.10 * .95135077)) = 0.909091.$$

Version History

Introduced: 4.0.1

Z and W arguments accept complex input: 5.6

See Also

[GAMMA](#), [IBETA](#), [IGAMMA](#), [LNGAMMA](#)

BILINEAR

The BILINEAR function uses a bilinear interpolation algorithm to compute the value of a data array at each of a set of subscript values.

This routine is written in the IDL language. Its source code can be found in the file `bilinear.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = BILINEAR(*P*, *IX*, *JY* [, MISSING=*value*])

Return Value

This function returns a two-dimensional interpolated array of the same type as the input array.

Arguments

P

A two-dimensional data array.

IX and JY

Arrays containing the X and Y “virtual subscripts” of *P* for which to interpolate values. *IX* and *JY* can be either of the following:

- One-dimensional, *n*-element floating-point arrays of subscripts to look up in *P*. One-dimensional arrays will be converted to two-dimensional arrays in such a way that *IX* contains *n* identical rows and *JY* contains *n* identical columns.
- Two-dimensional, *n*-element floating-point arrays that uniquely specify the X subscripts (the *IX* array) and the Y subscripts (the *JY* array) of the points to be computed from the input array *P*.

In either case, *IX* must satisfy the expressions

$$0 \leq \text{MIN}(\text{IX}) < \text{N0} \quad \text{and} \quad 0 < \text{MAX}(\text{IX}) \leq \text{N0}$$

where *N0* is the total number of columns in the array *P*. *JY* must satisfy the expressions

$$0 \leq \text{MIN}(\text{JY}) < \text{M0} \quad \text{and} \quad 0 < \text{MAX}(\text{JY}) \leq \text{M0}$$

where $M0$ is the total number of rows in the array P .

It is better to use two-dimensional arrays for IX and JY because the algorithm is somewhat faster. If IX and JY are specified as one-dimensional, the returned two-dimensional arrays IX and JY can be re-used on subsequent calls to take advantage of the faster 2D algorithm.

Keywords

MISSING

The value to return for elements outside the bounds of P . If this keyword is not specified, interpolated positions that fall outside the bounds of the array P - that is, elements of the IX or JY arguments that are either less than zero or greater than the largest subscript in the corresponding dimension of P - are set equal to the value of the nearest element of P .

Examples

Create a 3 x 3 floating point array P :

```
P = FINDGEN(3,3)
```

Suppose we wish to find the value of a point half way between the first and second elements of the first row of P . Create the subscript arrays IX and JY :

```
IX = 0.5 ;Define the X subscript.
JY = 0.0 ;Define the Y subscript.
Z = BILINEAR(P, IX, JY) ;Interpolate.
PRINT, Z ;Print the value at the point IX,JY within P.
```

IDL prints:

```
0.500000
```

Suppose we wish to find the values of a 2 x 2 array of points in P . Create the subscript arrays IX and JY :

```
IX = [[0.5, 1.9], [1.1, 2.2]] ;Define the X subscripts.
JY = [[0.1, 0.9], [1.2, 1.8]] ;Define the Y subscripts.
Z = BILINEAR(P, IX, JY) ;Interpolate.
PRINT, Z ;Print the array of values.
```

IDL prints:

```
0.800000    4.60000
4.70000     7.40000
```

Version History

Introduced: Original

MISSING keyword added: 6.1

See Also

[INTERPOL](#), [INTERPOLATE](#), [KRIG2D](#)

BIN_DATE

The BIN_DATE function converts a standard form ASCII date/time string to a binary string.

This routine is written in the IDL language. Its source code can be found in the file `bin_date.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = BIN_DATE(*Ascii_Time*)

Return Value

The function returns a six-element integer array where:

- Element 0 is the year (e.g., 1994)
- Element 1 is the month (1-12)
- Element 2 is the day (1-31)
- Element 3 is the hour (0-23)
- Element 4 is minutes (0-59)
- Element 5 is seconds (0-59)

Arguments

Ascii_Time

A string containing the date/time to convert in standard ASCII format. If this argument is omitted, the current date/time is used. Standard form is a 24 character string:

DOW MON DD HH:MM:SS YYYY

where DOW is the day of the week, MON is the month, DD is the day of month, HH:MM:SS is the time in hours, minutes, second, and YYYY is the year.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[CALDAT](#), [JULDAY](#), [SYSTIME](#)

BINARY_TEMPLATE

The `BINARY_TEMPLATE` function presents a graphical user interface which allows the user to interactively generate a template structure for use with `READ_BINARY`.

The graphical user interface allows the user to define one or more fields in the binary file. The file may be big, little, or native byte ordering.

Individual fields can be edited by the user to define the dimensionality and type of data to be read. Where necessary, fields can be defined in terms of other previously defined fields using IDL expressions. Fields can also be designated as “Verify”. When a file is read using a template with “Verify” fields, those fields will be checked against a user defined value supplied via the template.

Note

Greater than (“>”) and less than (“<”) symbols can appear in the “New Field” and the “Modify Field” dialogs where the offset value is displayed. The presence of either symbol indicates that the supplied offset value is “relative” from the end of the previous field or from the initial position in the file. Greater than means offset forward. Less than means offset backward. “>0” and “<0” are synonymous and mean “offset zero bytes”. You can delete these special symbols (thereby indicating that their corresponding offset value is not “relative”) by typing over them in the “New Field” or “Modify Field” dialogs.

Syntax

```
Result = BINARY_TEMPLATE ( [Filename] [, CANCEL=variable]  
[, GROUP=widget_id] [, N_ROWS=rows] [, TEMPLATE=variable] )
```

Return Value

This function returns an anonymous structure that contains the template. If the user cancels out of the graphical user interface and no initial template was supplied, it returns zero.

Arguments

Filename

A scalar string containing the name of a binary file which may be used to test the template. As the user interacts with the `BINARY_TEMPLATE` graphical user

interface, the user's input will be tested for correctness against the binary data in the file. If *filename* is not specified, a dialog allows the user to choose the file.

Keywords

CANCEL

Set this keyword to a named variable that will contain the byte value 1 if the user clicked the "Cancel" button, or 0 otherwise.

GROUP

The widget ID of an existing widget that serves as "group leader" for the `BINARY_TEMPLATE` interface. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

N_ROWS

Set this keyword to the number of rows to be visible in the `BINARY_TEMPLATE`'s table of fields.

Note

The `N_ROWS` keyword is analogous to the `WIDGET_TABLE` and the `Y_SCROLL_SIZE` keywords.

TEMPLATE

Set this keyword to structure variable containing an initial template (usually from a previous call to `BINARY_TEMPLATE`). This template structure will be used to fill in the initial fields in the new `BINARY_TEMPLATE`. If `TEMPLATE` is specified and the user cancels out of the dialog, the specified template will be returned as the Result.

The `BINARY_TEMPLATE` Interface

When the `BINARY_TEMPLATE` function is invoked, the following dialog is displayed:

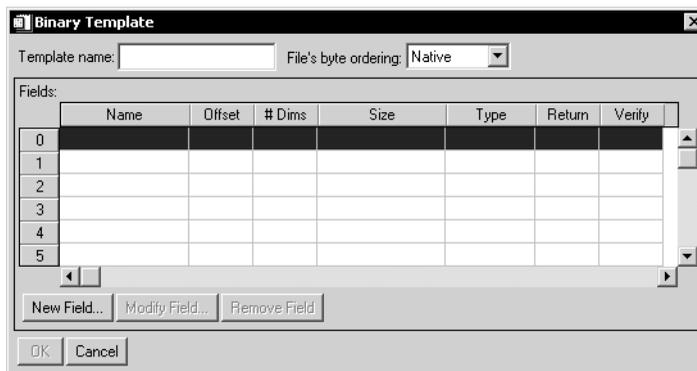


Figure 3-10: Binary Template

The **Template Name** is optional, and can be any string.

The byte order in the file is selected using the using the **File's byte ordering:** pull-down menu. The choices are:

- **Native** — The type of storage method that is native to the machine you are currently running. Little Endian for Intel microprocessor-based machines and Big Endian for Motorola microprocessor-based machines. No byte swapping will be performed.
- **Little Endian** — A method of storing numbers so that the least significant byte appears first in the number. For example, given the hexadecimal number A02B, the little endian method specifies the number to be stored as 2BA0. Specify this if the original file was created on a machine that uses an Intel microprocessor.
- **Big Endian** — A method of storing numbers so that the most significant byte appears first in the number. For example, given the hexadecimal number A02B, the big endian method specifies the number to be stored as A02B. Specify this if the original file was created on a machine that uses a Motorola microprocessor.

Fields are read in the order in which they are listed in the main dialog for `BINARY_TEMPLATE`, with offsets being added to the current file position pointer before each field is read. If a field has already been defined, clicking in the **Return** column will toggle the value of the field between Yes and No. Fields that are not marked for return can be used for calculations by other fields in the template. At least one field must be marked Yes for return in order for the `BINARY_TEMPLATE` function to return a template. Click **New Field...** to enter the description of a new template field. The New Field dialog appears:

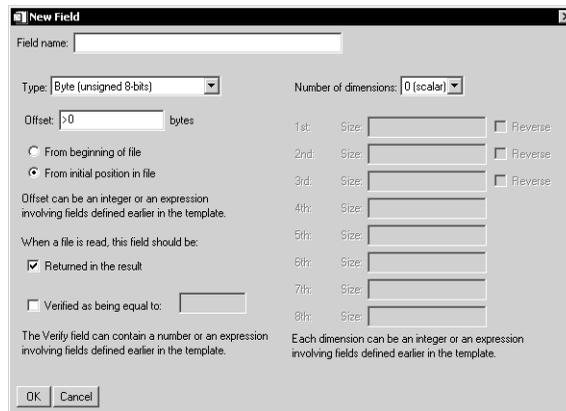


Figure 3-11: Binary Template - New Field

The **Field Name** can be any string.

The **Type** of each Template-specified field is selected from a droplist that offers the following IDL types: byte, integer, long, float, double, complex, dcomplex, uint, ulong, long64 and ulong64. Strings are read as an array of bytes for later conversion to type `STRING`.

Offsets can be specified using integer values, field names, or any valid IDL expression.

- An absolute integer offset specifies a fixed location (in bytes) from the beginning of the file (or the initial file position for an externally opened file).
- A relative integer offset specifies a position relative to the current file position pointer after the previous field (if any) is read. Relative offsets are shown in the `BINARY_TEMPLATE` user interface with a preceding `>` or `<` character, to indicate a positive (`>`) or negative (`<`) byte offset.

- Expressions can include the names of fields that will be read *before* the current field — that is, the field number of the referenced field must be lower than the field number of the field being defined.

The **Verify** field can contain an integer, field name, or any valid IDL expression. Only scalar fields can be verified. `READ_BINARY` reports an error if a verification fails.

The **Number of Dimensions** of a field can be set via a droplist of values 0 (scalar) to 8 (which is the maximum number of dimensions that an IDL variable can have.) The size of each dimension can be an integer, field name, or any valid IDL expression. Any of the first three dimensions of array data can also be specified to be reversed in order.

Note

If `BINARY_TEMPLATE` is called by a program that is running in the IDL Virtual Machine, the **Offsets**, **Verify**, and **Size** fields can contain integers or field names, but *not* an IDL expression.

Click **OK** to create the new field definition, and repeat to define all necessary fields.

The `BINARY_TEMPLATE` function returns a structure variable containing the template. The template variable can be saved and used as the value of the `TEMPLATE` keyword to the `READ_BINARY` function:

```
template = BINARY_TEMPLATE(file.dat)
Result = READ_BINARY('file.dat', TEMPLATE=template)
```

where `file.dat` is a binary data file to be read. The template variable can also be reused as the value of the `TEMPLATE` keyword to `BINARY_TEMPLATE`.

Version History

Introduced: 5.3

See Also

[READ_BINARY](#), [ASCII_TEMPLATE](#)

BINDGEN

The BINDGEN function creates a byte array with the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = \text{BINDGEN}(D_1 [, \dots, D_8])$$

Return Value

This function returns a byte array with the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments or array elements are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create a four-element by four-element byte array, and store the result in the variable A, enter:

```
A = BINDGEN(4,4)
```

Each element in **A** holds the value of its one-dimensional subscript. That is, if you enter the command:

```
PRINT, A
```

IDL prints the result:

```
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15
```

Version History

Introduced: Original

See Also

[CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#), [LINDGEN](#), [SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

BINOMIAL

The BINOMIAL function computes the probability that in a cumulative binomial (Bernoulli) distribution, a random variable X is greater than or equal to a user-specified value V , given N independent performances and a probability of occurrence or success P in a single performance:

$$\text{Probability}(X \geq V) = \sum_{x=V}^N \frac{N!}{x!(N-x)!} P^x (1-P)^{(N-x)}$$

This routine is written in the IDL language. Its source code can be found in the file `binomial.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = BINOMIAL(*V*, *N*, *P* [, /DOUBLE] [, /GAUSSIAN])

Return Value

This function returns a single- or double-precision floating point scalar or array that contains the value of the probability.

Arguments

V

A non-negative integer specifying the minimum number of times the event occurs in N independent performances.

N

A non-negative integer specifying the number of performances.

P

A non-negative single- or double-precision floating-point scalar or array, in the interval $[0.0, 1.0]$, that specifies the probability of occurrence or success of a single independent performance.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

GAUSSIAN

Set this keyword to use the Gaussian approximation, by using the normalized variable $Z = (V - NP)/\text{SQRT}(NP(1 - P))$.

Note

The Gaussian approximation is useful when N is large and neither P nor $(1-P)$ is close to zero, where the binomial summation may overflow. If GAUSSIAN is not explicitly set, and the binomial summation overflows, then BINOMIAL will automatically switch to using the Gaussian approximation.

Examples

Compute the probability of obtaining at least two 6s in rolling a die four times. The result should be 0.131944.

```
result = BINOMIAL(2, 4, 1.0/6.0)
```

Compute the probability of obtaining exactly two 6s in rolling a die four times. The result should be 0.115741.

```
result = BINOMIAL(2, 4, 1./6.) - BINOMIAL(3, 4, 1./6.)
```

Compute the probability of obtaining three or fewer 6s in rolling a die four times. The result should be 0.999228.

```
result = BINOMIAL(0, 4, 1./6.) - BINOMIAL(4, 4, 1./6.)
```

Version History

Introduced: Pre 4.0

See Also

[CHISQR_PDF](#), [F_PDF](#), [GAUSS_PDF](#), [T_PDF](#)

BLAS_AXPY

The `BLAS_AXPY` procedure updates an existing array by adding a multiple of another array. It can also be used to update one or more one-dimensional subvectors of an array according to the following vector operation:

$$Y = aX + Y$$

where a is a scale factor and X is an input vector.

`BLAS_AXPY` can be faster and use less memory than the usual IDL array notation (e.g. `Y=Y+A*X`) for updating existing arrays.

Note

`BLAS_AXPY` is much faster when operating on entire arrays and rows, than when used on columns or higher dimensions.

Syntax

```
BLAS_AXPY, Y, A, X [, D1, Loc1 [, D2, Range]]
```

Arguments

Y

The array to be updated. Y can be of any numeric type. `BLAS_AXPY` does not change the size and type of Y .

A

The scaling factor to be multiplied with X . A may be any scalar or one-element array that IDL can convert to the type of X . `BLAS_AXPY` does not change A .

X

The array to be scaled and added to array Y , or the vector to be scaled and added to subvectors of Y .

D1

An optional parameter indicating which dimension of Y is to be updated.

Loc1

A variable with the same number of elements as the number of dimensions of *Y*. The *Loc1* and *D1* arguments together determine which one-dimensional subvector (or subvectors, if *D1* and *Range* are provided) of *Y* is to be updated.

D2

An optional parameter, indicating in which dimension of *Y* a group of one-dimensional subvectors are to be updated. *D2* should be different from *D1*.

Range

A variable containing *D2* indices indicating where to put one-dimensional updates of *Y*.

Keywords

None.

Examples

The following examples show how to use the `BLAS_AXPY` procedure to add a multiple of an array, add a constant, and a group of subvectors.

```
seed = 5L
```

Create a multidimensional array:

```
A = FINDGEN(4, 5, 2)
```

Print A:

```
PRINT, A
```

IDL prints:

```

0.000000    1.00000    2.00000    3.00000
 4.00000    5.00000    6.00000    7.00000
 8.00000    9.00000   10.0000    11.0000
12.0000    13.0000    14.0000    15.0000
16.0000    17.0000    18.0000    19.0000

20.0000    21.0000    22.0000    23.0000
24.0000    25.0000    26.0000    27.0000
28.0000    29.0000    30.0000    31.0000
32.0000    33.0000    34.0000    35.0000
36.0000    37.0000    38.0000    39.0000

```

Create a random update:

```
B = RANDOMU(seed, 4, 5, 2)
```

Print B

```
PRINT, B
```

IDL prints:

0.172861	0.680409	0.917078	0.917510
0.766779	0.648501	0.334211	0.505953
0.652182	0.158174	0.912751	0.257593
0.810990	0.267308	0.188872	0.237323
0.312265	0.551604	0.944883	0.673464
0.613302	0.0874299	0.782052	0.374534
0.0799968	0.581460	0.433864	0.459824
0.634644	0.182057	0.832474	0.235194
0.432587	0.453664	0.738821	0.355747
0.933211	0.388659	0.269595	0.796325

Add a multiple of B to A (i.e., $A = A + 4.5*B$):

```
BLAS_AXPY, A, 4.5, B
```

Print A:

```
PRINT, A
```

IDL prints:

0.777872	4.06184	6.12685	7.12880
7.45051	7.91825	7.50395	9.27679
10.9348	9.71178	14.1074	12.1592
15.6495	14.2029	14.8499	16.0680
17.4052	19.4822	22.2520	22.0306
22.7599	21.3934	25.5192	24.6854
24.3600	27.6166	27.9524	29.0692
30.8559	29.8193	33.7461	32.0584
33.9466	35.0415	37.3247	36.6009
40.1994	38.7490	39.2132	42.5835

Add a constant to a subvector of A (i.e. $A[* , 3, 1] = A[* , 3, 1] + 4.3$):

```
BLAS_AXPY, A, 1., REPLICATE(4.3, 4), 1, [0, 3, 1]
```

Print A:

```
PRINT, A
```

IDL prints:

0.777872	4.06184	6.12685	7.12880
7.45051	7.91825	7.50395	9.27679
10.9348	9.71178	14.1074	12.1592
15.6495	14.2029	14.8499	16.0680
17.4052	19.4822	22.2520	22.0306
22.7599	21.3934	25.5192	24.6854
24.3600	27.6166	27.9524	29.0692
30.8559	29.8193	33.7461	32.0584
38.2466	39.3415	41.6247	40.9009
40.1994	38.7490	39.2132	42.5835

Create a vector update and print:

```
C = FINDGEN(5)
PRINT, C
```

IDL prints:

0.000000	1.00000	2.00000	3.00000	4.00000
----------	---------	---------	---------	---------

Add C to a group of subvectors of A (i.e. FOR i = 0, 1 DO A[1, *, i] = A[1, *, i] + C) and print:

```
BLAS_AXPY, A, 1., C, 2, [1, 0, 0], 3, LINDGEN(2)
PRINT, A
```

IDL prints:

0.777872	4.06184	6.12685	7.12880
7.45051	8.91825	7.50395	9.27679
10.9348	11.7118	14.1074	12.1592
15.6495	17.2029	14.8499	16.0680
17.4052	23.4822	22.2520	22.0306
22.7599	21.3934	25.5192	24.6854
24.3600	28.6166	27.9524	29.0692
30.8559	31.8193	33.7461	32.0584
38.2466	42.3415	41.6247	40.9009
40.1994	42.7490	39.2132	42.5835

Version History

Introduced: 5.1

See Also

[REPLICATE_INPLACE](#)

BLK_CON

The `BLK_CON` function computes a “fast convolution” of a digital signal and an impulse-response sequence. It returns the filtered signal.

This routine is written in the IDL language. Its source code can be found in the file `blk_con.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = BLK_CON( Filter, Signal [, B_LENGTH=scalar] [, /DOUBLE] )
```

Return Value

This function returns a vector with the same length as *Signal*. If either of the input arguments are double-precision or the `DOUBLE` keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

Filter

A P -element floating-point vector containing the impulse-response sequence of the digital filter.

Signal

An n -element floating-point vector containing the discrete signal samples.

Keywords

B_LENGTH

A scalar specifying the *block length* of the subdivided signal segments. If this parameter is not specified, a near-optimal value is chosen by the algorithm based upon the length P of the impulse-response sequence. If P is a value less than 11 or greater than 377, then `B_LENGTH` must be specified.

`B_LENGTH` must be greater than the filter length, P , and less than the number of signal samples.

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

Examples

```
; Create a filter of length P = 32:
filter = REPLICATE(1.0,32)    ;Set all points to 1.0
filter(2*INDGEN(16)) = 0.5    ;Set even points to 0.5

; Create a sampled signal with random noise:
signal = SIN((FINDGEN(1000)/35.0)^2.5)
noise = (RANDOMU(SEED,1000)-.5)/2.
signal = signal + noise

; Convolve the filter and signal using block convolution:
result = BLK_CON(filter, signal)
```

Version History

Introduced: Pre 4.0

See Also

[CONVOL](#)

BOX_CURSOR

The `BOX_CURSOR` procedure emulates the operation of a variable-sized box cursor (also known as a “marquee” selector).

Warning

`BOX_CURSOR` does not function properly when used within a draw widget. See the `BUTTON_EVENTS` and `MOTION_EVENTS` keywords in [WIDGET_DRAW](#).

This routine is written in the IDL language. Its source code can be found in the file `box_cursor.pro` in the `lib` subdirectory of the IDL distribution.

Using BOX_CURSOR

Once the box cursor has been realized, hold down the left mouse button to move the box by dragging. Hold down the middle mouse button to resize the box by dragging. (The corner nearest the initial mouse position is moved.) Press the right mouse button to exit the procedure and return the current box parameters.

On machines with only two mouse buttons, hold down the left and right buttons simultaneously to resize the box.

Syntax

```
BOX_CURSOR, [ X0, Y0, NX, NY [, /INIT] [, /FIXED_SIZE]] [, /MESSAGE]
```

Arguments

X0, Y0

Named variables that will contain the coordinates of the lower left corner of the box cursor.

NX, NY

Named variables that will contain the width and height of the cursor, in pixels.

Keywords

INIT

If this keyword is set, the arguments *X0*, *Y0*, *NX*, and *NY* contain the initial position and size of the box.

FIXED_SIZE

If this keyword is set, *NX* and *NY* contain the initial size of the box. This size may not be changed by the user.

MESSAGE

If this keyword is set, IDL prints a message describing operation of the cursor.

Version History

Introduced: Pre 4.0

See Also

Routines: [CURSOR](#)

Keywords to “IDL Graphics Devices” on page 4243: [CURSOR_CROSSHAIR](#), [CURSOR_IMAGE](#), [CURSOR_STANDARD](#), [CURSOR_XY](#)

BREAK

The BREAK statement provides a convenient way to immediately exit from a loop (FOR, WHILE, REPEAT), CASE, or SWITCH statement without resorting to GOTO statements.

Note

BREAK is an IDL statement. For information on using statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

BREAK

Examples

This example exits the enclosing WHILE loop when the value of i hits 5.

```
I = 0
WHILE (1) DO BEGIN
    i = i + 1
    IF (i eq 5) THEN BREAK
ENDWHILE
```

Version History

Introduced: 5.4

BREAKPOINT

The BREAKPOINT procedure allows you to insert and remove breakpoints in programs for debugging. A breakpoint causes program execution to stop after the designated statement is executed. Breakpoints are specified using the source file name and line number. For multiple-line statements (statements containing “\$”, the continuation character), specify the line number of the last line of the statement.

You can insert breakpoints in programs without editing the source file. Enter the following :

```
HELP, /BREAKPOINT
```

to display the breakpoint table which gives the index, module and source file locations of each breakpoint.

Syntax

```
BREAKPOINT [, File], Index [, AFTER=integer] [, /CLEAR]  
[, /CONDITION='expression'] [, /DISABLE] [, /ENABLE]  
[, /ON_RECOMPILE] [, /ONCE] [, /SET]
```

Arguments

File

An optional string argument that contains the name of the source file. Note that if *File* is not in the current directory, the full path name must be specified even if *File* is in one of the directories specified by !PATH.

Index

The line number at which to clear or set a breakpoint.

Keywords

AFTER

Set this keyword equal to an integer *n*. Execution will stop only after the *n*th time the breakpoint is hit. For example:

```
BREAKPOINT, /SET, 'test.pro', 8, AFTER=3
```

sets a breakpoint at the eighth line of the file `test.pro`, but only stops execution after the breakpoint has been encountered three times.

CLEAR

Set this keyword to remove a breakpoint. The breakpoint to be removed is specified either by index, or by the source file and line number. Use command `HELP, /BREAKPOINT` to display the indices of existing breakpoints. For example:

```
; Clear breakpoint with an index of 3:
BREAKPOINT, /CLEAR, 3

; Clear the breakpoint corresponding to the statement in the file
; test.pro, line number 8:
BREAKPOINT, /CLEAR, 'test.pro',8
```

CONDITION

Set this keyword to a string containing an IDL expression. When a breakpoint is encountered, the expression is evaluated. If the expression is true (if it returns a non-zero value), program execution is interrupted. The expression is evaluated in the context of the program containing the breakpoint. For example:

```
BREAKPOINT, 'myfile.pro', 6, CONDITION='i gt 2'
```

If `i` is greater than 2 at line 6 of `myfile.pro`, the program is interrupted.

DISABLE

Set this keyword to disable the specified breakpoint, if it exists. The breakpoint can be specified using the breakpoint index or file and line number:

```
; Disable breakpoint with an index of 3:
BREAKPOINT, /DISABLE, 3

; Disable the breakpoint corresponding to the statement in the file
; test.pro, line number 8:
BREAKPOINT, /DISABLE, 'test.pro',8
```

ENABLE

Set this keyword to enable the specified breakpoint if it exists. The breakpoint can be specified using the breakpoint index or file and line number:

```
; Enable breakpoint with an index of 3:
BREAKPOINT, /ENABLE, 3

; Enable the breakpoint at line 8 of the file test.pro
BREAKPOINT, /ENABLE, 'test.pro',8
```

ON_RECOMPILE

Set this keyword to specify that the breakpoint will not take effect until the next time the file containing it is compiled.

ONCE

Set this keyword to make the breakpoint temporary. If ONCE is set, the breakpoint is cleared as soon as it is hit. For example:

```
BREAKPOINT, /SET, 'file.pro', 12, AFTER=3, /ONCE
```

sets a breakpoint at line 12 of `file.pro`. Execution stops when line 12 is encountered the third time, and the breakpoint is automatically cleared.

SET

Set this keyword to set a breakpoint at the designated source file line. If this keyword is set, the first input parameter, *File* must be a string expression that contains the name of the source file. The second input parameter must be an integer that represents the source line number.

For example, to set a breakpoint at line 23 in the source file `xyz.pro`, enter:

```
BREAKPOINT, /SET, 'xyz.pro', 23
```

Version History

Introduced: Pre 4.0

BROYDEN

The BROYDEN function solves a system of n nonlinear equations (where $n \geq 2$) in n dimensions using a globally-convergent Broyden's method.

BROYDEN is based on the routine `broydn` described in section 9.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = BROYDEN( X, Vecfunc [, CHECK=variable] [, /DOUBLE] [, EPS=value]
  [, ITMAX=value] [, STEPMAX=value] [, TOLF=value] [, TOLMIN=value]
  [, TOLX=value] )
```

Return Value

This function returns an n -element vector containing the solution.

Arguments

X

An n -element vector (where $n \geq 2$) containing an initial guess at the solution of the system.

Vecfunc

A scalar string specifying the name of a user-supplied IDL function that defines the system of non-linear equations. This function must accept a vector argument X and return a vector result.

For example, suppose we wish to solve the following system:

$$\begin{bmatrix} 3x - \cos(yz) - 1/2 \\ x^2 - 81(y + 0.1)^2 + \sin(z) + 1.06 \\ e^{-xy} + 20z + \frac{10\pi - 3}{3} \end{bmatrix} = 0$$

To represent this system, we define an IDL function named BROYFUNC:

```

FUNCTION broyfunc, X
  RETURN, [3.0 * X[0] - COS(X[1]*X[2]) - 0.5,$
          X[0]^2 - 81.0*(X[1] + 0.1)^2 + SIN(X[2]) + 1.06,$
          EXP(-X[0]*X[1]) + 20.0 * X[2] + (10.0*!PI - 3.0)/3.0]
END

```

Keywords

CHECK

BROYDEN calls an internal function named `fmin()` to determine whether the routine has converged to a local rather than a global minimum (see *Numerical Recipes*, section 9.7). Use the **CHECK** keyword to specify a named variable which will be set to 1 if the routine has converged to a local minimum or to 0 if not. If the routine does converge to a local minimum, try restarting from a different initial guess to obtain the global minimum.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EPS

Set this keyword to a number close to machine accuracy, used to remove noise from each iteration. The default is 10^{-7} for single precision, and 10^{-14} for double precision.

ITMAX

Use this keyword to specify the maximum allowed number of iterations. The default is 200.

STEPMAX

Use this keyword to specify the scaled maximum step length allowed in line searches. The default value is 100.0.

TOLF

Set the convergence criterion on the function values. The default value is 1.0×10^{-4} .

TOLMIN

Set the criterion for deciding whether spurious convergence to a minimum of the function `fmin()` has occurred. The default value is 1.0×10^{-6} .

TOLX

Set the convergence criterion on X . The default value is 1.0×10^{-7} .

Examples

We can use `BROYDEN` to solve the non-linear system of equations defined by the `BROYFUNC` function above:

```
;Provide an initial guess as the algorithm's starting point:  
X = [-1.0, 1.0, 2.0]  
  
;Compute the solution:  
result = BROYDEN(X, 'BROYFUNC')  
  
;Print the result:  
PRINT, result
```

IDL prints:

```
0.500000 -1.10731e-07 -0.523599
```

The exact solution (to eight-decimal accuracy) is [0.5, 0.0, -0.52359877].

Version History

Introduced: 4.0

See Also

[FX_ROOT](#), [FZ_ROOTS](#), [NEWTON](#)

BYTARR

The BYTARR function creates a byte vector or array.

Syntax

Result = BYTARR(D_1 [, ..., D_8] [, /NOZERO])

Return Value

This function returns a byte vector or array.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, BYTARR sets every element of the result to zero. If the NOZERO keyword is set, this zeroing is not performed (array elements contain random values) and BYTARR executes faster.

Examples

To create B as a 3 by 3 by 5 byte array where each element is set to zero, enter:

```
B = BYTARR(3, 3, 5)
```

Version History

Introduced: Original

See Also

[COMPLEXARR](#), [DBLARR](#), [FLTARR](#), [INTARR](#), [LON64ARR](#), [LONARR](#),
[MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

BYTE

The BYTE function returns a result equal to *Expression* converted to byte type. If *Expression* is a string, each string is converted to a byte vector of the same length as the string. Each element of the vector is the character code of the corresponding character in the string. The BYTE function can also be used to extract data from *Expression* and place it in a byte scalar or array without modification, if more than one parameter is present. See “[Type Conversion Functions](#)” on page 58 for details.

Syntax

$$Result = \text{BYTE}(Expression[, Offset [, D_1[, \dots, D_8]]])$$

Return Value

This function returns the result of the *Expression* converted to byte type.

Arguments

Expression

The expression to be converted to type byte.

Offset

The byte offset from the beginning of *Expression*. Specifying this argument allows fields of data extracted from *Expression* to be treated as byte data without conversion.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Example

If the variable A contains the floating-point value 10.0, it can be converted to byte type and saved in the variable B by entering:

```
B = BYTE(A)
```

Version History

Introduced: Original

See Also

[COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#), [LONG64](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

BYTEORDER

The `BYTEORDER` procedure converts integers between host and network byte ordering or floating-point values between the native format and XDR (IEEE) format. This routine can also be used to swap the order of bytes within both short and long integers. If the type of byte swapping is not specified via one of the keywords below, bytes within short integers are swapped (even and odd bytes are interchanged).

The size of the parameter, in bytes, must be evenly divisible by two for short integer swaps, and by four for long integer swaps. `BYTEORDER` operates on both scalars and arrays. The parameter must be a variable, not an expression or constant, and may not contain strings. The contents of *Variable* are overwritten by the result.

Network byte ordering is “big endian”. That is, multiple byte integers are stored in memory beginning with the most significant byte.

Syntax

```

BYTEORDER, Variable1, ..., Variablen [, /DTOVAX] [, /DTOXDR] [, /FTOVAX]
  [, /FTOXDR] [, /HTONL] [, /HTONS] [, /L64SWAP] [, /LSWAP] [, /NTOHL]
  [, /NTOHS] [, /SSWAP] [, /SWAP_IF_BIG_ENDIAN]
  [, /SWAP_IF_LITTLE_ENDIAN] [, /VAXTOD] [, /VAXTOF] [, /XDRTOD]
  [, /XDRTOF]

```

Arguments

Variable_n

A named variable (not an expression or constant) that contains the data to be converted. The contents of *Variable* are overwritten by the new values.

Keywords

D_TOVAX

Set this keyword to convert native (IEEE) double-precision floating-point format to VAX D float format. See “[Note on Accessing Data in VAX Floating Point Format](#)” on page 204.

DTOXDR

Set this keyword to convert native double-precision floating-point format to XDR (IEEE) format.

FTOVAX

Set this keyword to convert native (IEEE) single-precision floating-point format to VAX F float format. See [“Note on Accessing Data in VAX Floating Point Format”](#) on page 204.

FTOXDR

Set this keyword to convert native single-precision floating-point format to XDR (IEEE) format.

HTONL

Set this keyword to perform host to network conversion, longwords.

HTONS

Set this keyword to perform host to network conversion, short integers.

L64SWAP

Set this keyword to perform a 64-bit swap (8 bytes). Swap the order of the bytes within each 64-bit word. For example, the eight bytes within a 64-bit word are changed from $(B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7)$, to $(B_7, B_6, B_5, B_4, B_3, B_2, B_1, B_0)$.

LSWAP

Set this keyword to perform a 32-bit longword swap. Swap the order of the bytes within each longword. For example, the four bytes within a longword are changed from (B_0, B_1, B_2, B_3) , to (B_3, B_2, B_1, B_0) .

NTOHL

Set this keyword to perform network to host conversion, longwords.

NTOHS

Set this keyword to perform network to host conversion, short integers.

SSWAP

Set this keyword to perform a short word swap. Swap the bytes within short integers. The even and odd numbered bytes are interchanged. This is the default action, if no other keyword is set.

SWAP_IF_BIG_ENDIAN

If this keyword is set, the `BYTEORDER` request will only be performed if the platform running IDL uses “big endian” byte ordering. On little endian machines, the `BYTEORDER` request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

If this keyword is set, the `BYTEORDER` request will only be performed if the platform running IDL uses “little endian” byte ordering. On big endian machines, the `BYTEORDER` request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

VAXTOD

Set this keyword to convert VAX D float format to native (IEEE) double-precision floating-point format. See [“Note on Accessing Data in VAX Floating Point Format”](#) on page 204.

VAXTOF

Set this keyword to convert VAX F float format to native (IEEE) single-precision floating-point format. See [“Note on Accessing Data in VAX Floating Point Format”](#) on page 204.

XDRTOD

Set this keyword to convert XDR (IEEE) format to native double-precision floating-point.

XDRTOF

Set this keyword to convert XDR (IEEE) format to native single-precision floating-point.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Obsolete Keywords

The following keywords are obsolete:

- DTOGFLOAT
- GFLOATTOD

For information on obsolete keywords, See [Appendix I, "Obsolete Features"](#).

Note on Accessing Data in VAX Floating Point Format

When converting between VAX and IEEE formats, you should be aware of the following basic numerical issues in order to get the best results. Translation of floating-point values from IDL's native IEEE format to the VAX format and back (that is, VAX to IEEE to VAX) is not a completely reversible operation, and should be avoided when possible. There are many cases where the recovered values will differ from the original values, including:

- The VAX floating-point format lacks support for the IEEE special values (*NaN* and *Infinity*). Hence, their special meaning is lost when they are converted to VAX format and cannot be recovered.
- The IEEE and VAX floating formats have intrinsic differences in precision and range, which can cause information to be lost in both directions. When converting from one format to another, IDL rounds the value to the nearest representable value in the target format.

As a practical matter, an initial conversion of existing VAX format data to IEEE cannot be avoided if the data is to be used on modern machines. However, each format conversion can add a small amount of error to the resulting values, so it is important to minimize the number of such conversions. RSI recommends using IEEE/VAX conversions only to read existing VAX format data, and strongly

recommends that all new files be created using the native IEEE format. This introduces only a single unavoidable conversion, and minimizes the resulting conversion error.

Version History

Introduced: Pre 4.0

See Also

[SWAP_ENDIAN](#)

BYTSCL

The BYTSCL function scales all values of *Array* that lie in the range ($Min \leq x \leq Max$) into the range ($0 \leq x \leq Top$). For floating-point input, each value is scaled using the formula $(Top + 0.9999) * x / (Max - Min)$. For integer input, each value is scaled using the formula $((Top + 1) * x - 1) / (Max - Min)$.

Syntax

Result = BYTSCL(*Array* [, **MAX**=value] [, **MIN**=value] [, **/NAN**] [, **TOP**=value])

Return Value

The returned result has the same structure as the original parameter and is of byte type.

Arguments

Array

The array to be scaled and converted to bytes.

Keywords

MAX

Set this keyword to the maximum value of *Array* to be considered. If MAX is not provided, *Array* is searched for its maximum value. All values greater or equal to MAX are set equal to TOP in the result.

Note

The data type of the value specified for MAX should match the data type of the input array. Since MAX is converted to the data type of the input array, specifying mismatched data types may produce undesired results.

MIN

Set this keyword to the minimum value of *Array* to be considered. If MIN is not provided, *Array* is searched for its minimum value. All values less than or equal to MIN are set equal to 0 in the result.

Note

The data type of the value specified for MIN should match the data type of the input array. Since MIN is converted to the data type of the input array, specifying mismatched data types may produce undesired results.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

TOP

Set this keyword to the maximum value of the scaled result. If TOP is not specified, 255 is used. Note that the minimum value of the scaled result is always 0.

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

BYTSCL is often used to scale images into the appropriate range for 8-bit displays. As an example, enter the following commands:

```
; Create a simple image array and display:
IM = DIST(200)
TV, IM

; Scale the image into the full range of bytes (0 to 255) and
; re-display it:
IM = BYTSCL(IM)

; Display the new image:
TV, IM
```

Version History

Introduced: Original

See Also

[BYTE](#), [TVSCL](#)

C_CORRELATE

The C_CORRELATE function computes the cross correlation $P_{xy}(L)$ or cross covariance $R_{xy}(L)$ of two sample populations X and Y as a function of the lag L

$$P_{xy}(L) = \begin{cases} \frac{\sum_{k=0}^{N-|L|-1} (x_{k+|L|} - \bar{x})(y_k - \bar{y})}{\sqrt{\left[\sum_{k=0}^{N-1} (x_k - \bar{x})^2 \right] \left[\sum_{k=0}^{N-1} (y_k - \bar{y})^2 \right]}} & \text{For } L < 0 \\ \frac{\sum_{k=0}^{N-L-1} (x_k - \bar{x})(y_{k+L} - \bar{y})}{\sqrt{\left[\sum_{k=0}^{N-1} (x_k - \bar{x})^2 \right] \left[\sum_{k=0}^{N-1} (y_k - \bar{y})^2 \right]}} & \text{For } L \geq 0 \end{cases}$$

$$R_{xy}(L) = \begin{cases} \frac{1}{N} \sum_{k=0}^{N-|L|-1} (x_{k+|L|} - \bar{x})(y_k - \bar{y}) & \text{For } L < 0 \\ \frac{1}{N} \sum_{k=0}^{N-L-1} (x_k - \bar{x})(y_{k+L} - \bar{y}) & \text{For } L \geq 0 \end{cases}$$

where \bar{x} and \bar{y} are the means of the sample populations $x = (x_0, x_1, x_2, \dots, x_{N-1})$ and $y = (y_0, y_1, y_2, \dots, y_{N-1})$, respectively.

This routine is written in the IDL language. Its source code can be found in the file `c_correlate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$\text{Result} = \text{C_CORRELATE}(X, Y, \text{Lag} [, /\text{COVARIANCE}] [, /\text{DOUBLE}])$$

Return Value

Returns the cross correlation $P_{xy}(L)$ or cross covariance $R_{xy}(L)$ of two sample populations X and Y as a function of the lag L .

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An n -element integer, single-, or double-precision floating-point vector.

Lag

A scalar or n -element integer vector in the interval $[-(n-2), (n-2)]$, specifying the signed distances between indexed elements of X .

Keywords

COVARIANCE

Set this keyword to compute the sample cross covariance rather than the sample cross correlation.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```
; Define two n-element sample populations:
X = [3.73, 3.67, 3.77, 3.83, 4.67, 5.87, 6.70, 6.97, 6.40, 5.57]
Y = [2.31, 2.76, 3.02, 3.13, 3.72, 3.88, 3.97, 4.39, 4.34, 3.95]
```

```
    ; Compute the cross correlation of X and Y for LAG = -5, 0, 1, 5,  
    ; 6, 7:  
    lag = [-5, 0, 1, 5, 6, 7]  
    result = C_CORRELATE(X, Y, lag)  
    PRINT, result
```

IDL prints:

```
-0.428246  0.914755  0.674547  -0.405140  -0.403100  -0.339685
```

Version History

Introduced: 4.0

See Also

[A_CORRELATE](#), [CORRELATE](#), [M_CORRELATE](#), [P_CORRELATE](#),
[R_CORRELATE](#)

CALDAT

The CALDAT procedure computes the month, day, year, hour, minute, or second corresponding to a given Julian date. The inverse of this procedure is JULDAY.

Note

The Julian calendar, established by Julius Caesar in the year 45 BCE, was corrected by Pope Gregory XIII in 1582, excising ten days from the calendar. The CALDAT procedure reflects the adjustment for dates after October 4, 1582. See the example below for an illustration.

This routine is written in the IDL language. Its source code can be found in the file `caldat.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

CALDAT, *Julian*, *Month* [, *Day* [, *Year* [, *Hour* [, *Minute* [, *Second*]]]]]

Arguments

Julian

A numeric value or array that specifies the Julian Day Number (which begins at noon) to be converted to a calendar date.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000, respectively.

Note

Julian Day Numbers should be maintained as double-precision floating-point data when the numbers are used to determine hours, minutes, and seconds.

Month

A named variable that, on output, contains a longword integer or longword integer array representing the number of the desired month (1 = January, ..., 12 = December).

Day

A named variable that, on output, contains a longword integer or longword integer array representing the number of the day of the month (1-31).

Year

A named variable that, on output, contains a longword integer or longword integer array representing the number of the desired year (e.g., 1994).

Hour

A named variable that, on output, contains a longword integer or longword integer array representing the number of the hour of the day (0-23).

Minute

A named variable that, on output, contains a longword integer or longword integer array representing the number of the minute of the hour (0-59).

Second

A named variable that, on output, contains a double-precision floating-point value or a double-precision floating-point array representing the number of the second of the minute (0-59).

Keywords

None.

Examples

In 1582, Pope Gregory XIII adjusted the Julian calendar to correct for its inaccuracy of slightly more than 11 minutes per year. As a result, the day following October 4, 1582 was October 15, 1582. CALDAT follows this convention, as illustrated by the following commands:

```
CALDAT, 2299160, Month1, Day1, Year1
CALDAT, 2299161, Month2, Day2, Year2
PRINT, Month1, Day1, Year1
PRINT, Month2, Day2, Year2
```

IDL prints:

```
10    4    1582
10   15    1582
```

Warning

You should be aware of this discrepancy between the original and revised Julian calendar reckonings if you calculate dates before October 15, 1582.

Be sure to distinguish between *Month* and *Minute* when assigning variable names. For example, the following code would cause the Month value to be the same as the Minute value:

```
;Find date corresponding to Julian day 2529161.36:
CALDAT, 2529161.36, M, D, Y, H, M, S
PRINT, M, D, Y, H, M, S
```

IDL prints:

```
0    4    2212    18    0    0.00000000
```

Moreover, Julian Day Numbers should be maintained as double-precision floating-point data when the numbers are used to determine hours, minutes, and seconds.

So, instead of the previous call to CALDAT, use something like:

```
CALDAT, 2529161.36D, Month, Day, Year, Hour, Minute, Second
PRINT, Month, Day, Year, Hour, Minute, Second
```

IDL prints:

```
7    4    2212    20    38    23.999989
```

You can also use arrays for the *Julian* argument:

```
CALDAT, DINDGEN(4) + 2449587.0D, m, d, y
PRINT, m, d, y
```

IDL prints:

```
            8            8            8            8
           22           23           24           25
        1994        1994        1994        1994
```

Version History

Introduced: Pre 4.0

See Also

[BIN_DATE](#), [JULDAY](#), [SYSTIME](#)

CALENDAR

The CALENDAR procedure displays a calendar for a month or an entire year on the current plotting device. This IDL routine imitates the UNIX `cal` command.

This routine is written in the IDL language. Its source code can be found in the file `calendar.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
CALENDAR [[, Month], Year]
```

Arguments

Month

The number of the month for which a calendar is desired (1 is January, 2 is February, ..., 12 is December). If called without arguments, CALENDAR draws a calendar for the current month.

Year

The number of the year for which a calendar should be drawn. If YEAR is provided without MONTH, a calendar for the entire year is drawn. If called without arguments, CALENDAR draws a calendar for the current month.

Example

```
; Display a calendar for the year 2038.  
CALENDAR, 2038  
; Display the calendar for October, 1582.  
CALENDAR, 10, 1582
```

Version History

Introduced: Original

See Also

[SYSTEMTIME](#)

CALL_EXTERNAL

The `CALL_EXTERNAL` function calls a function in an external sharable object and returns a scalar value. Parameters can be passed by reference (the default) or by value. See [Chapter 9, “CALL_EXTERNAL”](#) in the *External Development Guide* manual for examples.

`CALL_EXTERNAL` is supported under all operating systems supported by IDL, although there are system specific details of which you must be aware. This function requires no interface routines and is much simpler and easier to use than the `LINKIMAGE` procedure. However, `CALL_EXTERNAL` performs no checking of the type and number of parameters. Programming errors are likely to cause IDL to crash or to corrupt your data.

Warning

Input and output actions should be performed within IDL code, using IDL's built-in input/output facilities, or by using the internal `IDL_Message()` function. Performing input or output from external code, especially to the user console or tty (e.g. using `printf()` or equivalent functionality in other languages to send text to `stdout`) may create errors or generate unexpected results.

`CALL_EXTERNAL` supports the IDL Portable Convention, a portable calling convention that works on all platforms. This convention passes two arguments to the called routine, an argument count (`argc`) and an array of arguments (`argv`).

`CALL_EXTERNAL` also offers a feature called [Auto Glue](#) that can greatly simplify use of the `CALL_EXTERNAL` portable convention if you have the appropriate C compiler installed on your system. Auto glue automatically writes the glue function required to convert the (`argc`, `argv`) arguments to the actual function call, and then compiles and loads the glue function transparently. If you want IDL to simply write the glue function for you, but not compile it, the `WRITE_WRAPPER` keyword can be used.

The result of the `CALL_EXTERNAL` function is a scalar value returned by the external function. By default, this is a scalar long (32-bit) integer. This default can be changed by specifying one of the keywords described below that alter the result type.

Syntax

```
Result = CALL_EXTERNAL(Image, Entry [, P0, ..., PN-1] [, /ALL_VALUE]
    [, /B_VALUE | /D_VALUE | /F_VALUE | /I_VALUE | /L64_VALUE |
    , /S_VALUE | /UI_VALUE | /UL_VALUE | /UL64_VALUE] [, /CDECL]
```

```
[, RETURN_TYPE=value] [, /UNLOAD] [, VALUE=byte_array]
[, WRITE_WRAPPER=wrapper_file] )
```

Auto Glue keywords: [, /AUTO_GLUE] [, CC=string]
 [, COMPILE_DIRECTORY=string] [, EXTRA_CFLAGS=string]
 [, EXTRA_LFLAGS=string] [, /IGNORE_EXISTING_GLUE] [, LD=string]
 [, /NOCLEANUP] [, /SHOW_ALL_OUTPUT] [, /VERBOSE]

Return Value

This function calls a function in an external sharable object and returns a scalar value.

Arguments

Image

The name of the file, which must be a sharable library (UNIX), or DLL (Windows), which contains the routine to be called.

Entry

A string containing the name of the symbol in the library which is the entry point of the routine to be called.

P_0, \dots, P_{N-1}

The parameters to be passed to the external routine. All array and structure arguments are passed by reference (address). The default is to also pass scalars by reference, but the ALL_VALUE or VALUE keywords can be used to pass them by value. Care must be taken to ensure that the type, structure, and passing mechanism of the parameters passed to the external routine match what it expects. There are some restrictions on data types that can be passed by value, and the user needs to be aware of how IDL passes strings. Both issues discussed in further detail below.

Keywords

ALL_VALUE

Set this keyword to indicate that all parameters are passed by value. There are some restrictions on data types that should be considered when using this keyword, as discussed below.

B_VALUE

If set, this keyword indicates that the called function returns a byte value.

CDECL

The Microsoft Windows operating system has two distinct system defined standards that govern how routines pass arguments: `stdcall`, which is used by much of the operating system as well as languages such as Visual Basic, and `cdecl`, which is used widely for programming in the C language. These standards differ in how and when arguments are pushed and removed from the system stack. The standard used by a given function is determined when the function is compiled, and can usually be controlled by the programmer. If you call a function using the wrong standard (e.g. calling a `stdcall` function as if it were `cdecl`, or the reverse), you could get incorrect results, corrupted memory, or you could crash IDL. Unfortunately, there is no way for IDL to know which convention a given function uses; this information must be supplied by the user of `CALL_EXTERNAL`. If the `CDECL` keyword is present, IDL will use the `cdecl` convention to call the function. Otherwise, `stdcall` is used.

D_VALUE

If set, this keyword indicates that the called function returns a double-precision floating value.

F_VALUE

If set, this keyword indicates that the called function returns a single-precision floating value.

I_VALUE

If set, this keyword indicates that the called function returns an integer value.

L64_VALUE

If set, this keyword indicates that the called function returns a 64-bit integer value.

RETURN_TYPE

The type code to set the type of the result. See the description of the [SIZE](#) function for a list of the IDL type codes.

S_VALUE

If set, this keyword indicates that the called function returns a pointer to a null-terminated string.

UI_VALUE

If set, this keyword indicates that the called function returns an unsigned integer value.

UL_VALUE

If set, this keyword indicates that the called function returns an unsigned long integer value.

UL64_VALUE

If set, this keyword indicates that the called function returns an unsigned 64-bit integer value.

UNLOAD

Normally, IDL keeps *Image* loaded in memory after the call to `CALL_EXTERNAL` completes. This is done for efficiency—loading a sharable object can be a slow operation. Setting the `UNLOAD` keyword will cause IDL to unload *Image* after the call to it is complete. This is useful if you are debugging code in *Image*, as it allows you to iterate on your code without having to exit IDL between tests. It can also be a good idea if you do not intend to make any subsequent calls to routines within *Image*.

If IDL is unable to unload the sharable object, it will issue an error to that effect. In addition to any operating system reported problem that might occur, there are 2 situations in which IDL cannot perform the `UNLOAD` operation:

- If the sharable library has been used for any other purpose in addition to `CALL_EXTERNAL` (e.g. `LINKIMAGE`).

VALUE

A byte array, with as many elements as there are optional parameters, indicating the method of parameter passing. Arrays are always passed by reference. If parameter `Pi` is a scalar, it is passed by reference if `VALUE[i]` is 0; and by value if it is non-zero. There are some restrictions on data types that should be considered when using this keyword, as discussed below.

WRITE_WRAPPER

If set, `WRITE_WRAPPER` supplies the name of a file for `CALL_EXTERNAL` to create containing the C function required to convert the `(argc, argv)` interface used by the `CALL_EXTERNAL` portable calling convention to the interface of the target function. If `WRITE_WRAPPER` is specified, `CALL_EXTERNAL` writes the specified file, but does not attempt to actually call the function specified by Entry. The result from `CALL_EXTERNAL` is an integer 0 in this case, and has no special meaning. Use of `WRITE_WRAPPER` implies the `PORTABLE` keyword.

Note

This is similar to Auto Glue only in that `CALL_EXTERNAL` writes a function on your behalf. Unlike Auto Glue, `WRITE_WRAPPER` does not attempt to compile the resulting function or to use it. You might want to use `WRITE_WRAPPER` to generate IDL interfaces for an external library in cases where you intend to combine the interfaces with other code or otherwise modify it before using it with IDL.

Auto Glue Keywords

Auto Glue, discussed in the section “[Auto Glue](#)” on page 224, offers a simplified way to use the `CALL_EXTERNAL` portable calling convention. The following keywords control its use. Many of these keywords correspond to the same keywords to the `MAKE_DLL` procedure, and are covered in more detail in the documentation for that routine.

AUTO_GLUE

Set this keyword to enable the `CALL_EXTERNAL` Auto Glue feature.

CC

If present, a template string to be used in generating the C compiler command(s) to compile the automatically generated glue function. For a more complete description of this keyword, see [MAKE_DLL](#).

COMPILE_DIRECTORY

Specifies the directory to use for creating the necessary intermediate files and the final glue function sharable library. For a more complete description of this keyword, see [MAKE_DLL](#).

EXTRA_CFLAGS

If present, a string supplying extra options to the command used to execute the C compiler. For a more complete description of this keyword, see [MAKE_DLL](#).

EXTRA_LFLAGS

If present, a string supplying extra options to the command used to execute the linker. For a more complete description of this keyword, see [MAKE_DLL](#).

IGNORE_EXISTING_GLUE

Normally, if Auto Glue finds a pre-existing glue function, it will use it without attempting to build it again. Set `IGNORE_EXISTING_GLUE` to override this caching behavior and force `CALL_EXTERNAL` to rebuild the glue function sharable library.

LD

If present, a template string to be used in generating the linker command to build the glue function sharable library. For a more complete description of this keyword, see [MAKE_DLL](#).

NOCLEANUP

If set, `CALL_EXTERNAL` will not remove intermediate files generated in order to build the glue function sharable library after the library has been built. This keyword can be used to preserve information for debugging in case of error, or for additional information on how Auto Glue works. For a more complete description of this keyword, see [MAKE_DLL](#).

SHOW_ALL_OUTPUT

Auto Glue normally produces no output unless an error prevents successful building of the glue function sharable library. Set `SHOW_ALL_OUTPUT` to see all output produced by the process of building the library. For a more complete description of this keyword, see [MAKE_DLL](#).

VERBOSE

If set, `VERBOSE` causes `CALL_EXTERNAL` to issue informational messages as it carries out the task of locating, building, and executing the glue function. For a more complete description of this keyword, see [MAKE_DLL](#).

Obsolete Keywords

The following keywords are obsolete:

- DEFAULT
- PORTABLE
- VAX_FLOAT

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

String Parameters

IDL represents strings internally as IDL_STRING descriptors, which are defined in the C language as:

```
typedef struct {
    unsigned short slen;
    unsigned short stype;
    char *s;
} IDL_STRING;
```

To pass a string by reference, IDL passes the address of its IDL_STRING descriptor. To pass a string by value the string pointer (the `s` field of the descriptor) is passed. Programmers should be aware of the following when manipulating IDL strings:

- Called code should treat the information in the passed IDL_STRING descriptor and the string itself as read-only, and should not modify these values.
- The `slen` field contains the length of the string without including the NULL termination that is required at the end of all C strings.
- The `stype` field is used internally by IDL to know keep track of how the memory for the string was obtained, and should be ignored by CALL_EXTERNAL users.
- `s` is the pointer to the actual C string represented by the descriptor. If the string is NULL, IDL represents it as a NULL (0) pointer, not as a pointer to an empty null terminated string. Hence, called code that expects a string pointer should check for a NULL pointer before dereferencing it.

These issues are examined in greater detail in the IDL *External Development Guide*.

Calling Convention

`CALL_EXTERNAL` uses the IDL Portable convention for calling user-supplied routines. The IDL Portable calling convention can be simplified by using the Auto Glue extension, described below.

The portable interface convention passes all arguments as elements of an array of C void pointers (`void *`). The C language prototype for a user function called this way looks like one of the following:

```
RET_TYPE xxx(int argc, void *argv[])
```

Where `RET_TYPE` is one of the following: `UCHAR`, `short`, `IDL_UINT`, `IDL_LONG`, `IDL_ULONG`, `IDL_LONG64`, `IDL_ULONG64`, `float`, `double`, or `char *`. The return type used must agree with the type assumed by `CALL_EXTERNAL` as specified via the keywords described above.

`argc` is the number of arguments, and the vector `argv` contains the arguments themselves, one argument per element. Arguments passed by reference map directly to these (`void *`) pointers, and can be cast to the proper type and then dereferenced directly by the called function. Passing arguments by value is allowed, but since the values are passed in (`void *`) pointers, there are some limitations and restrictions on what is possible:

- Types that are larger than a pointer cannot be passed by value, and `CALL_EXTERNAL` will issue an error if this is attempted. This limitation applies only to the standard portable calling convention. Auto Glue does not have this limitation, and is able to pass such variables by value.
- Integer values can be easily passed by value. IDL widens any of the integer types to the C `int` type and they are then converted to a (`void *`) pointer using a C cast operation.
- There is no C language-defined conversion between pointers and floating point types, so IDL copies the data for the value directly into the pointer element. Although such values can be retrieved by the called routine with the correct C casting operations, this is inconvenient and error prone. It is best to pass non-integer data by reference.

Auto Glue

Auto Glue is an extension to the IDL Portable Calling Convention that makes it easier to use.

The portable calling convention requires your function to use the IDL defined (`argc`, `argv`) interface for passing arguments. However, functions not explicitly written for use with `CALL_EXTERNAL` may not have this interface. A common solution using the portable convention is for the IDL user to write a *glue* function that serves as an interface between IDL and the called function. The entire purpose of this glue function, which is usually very simple, is to convert the IDL (`argc`, `argv`) method of passing parameters to a form acceptable to the called function. Writing this wrapper function is easy for programmers who understand the C language, the system C compiler and linker, and how sharable libraries work on their target operating system. However, it is also tedious and error prone, and can be difficult for users that do not already have these skills.

Auto Glue uses the `MAKE_DLL` procedure to automate the process of using glue code to call functions via the `CALL_EXTERNAL` portable calling convention. Since it depends so closely on `MAKE_DLL`, an understanding of how `MAKE_DLL` works is necessary to fully understand Auto Glue. As with `MAKE_DLL`, Auto Glue requires that your system have a suitable C compiler installed. Please refer to the documentation for `MAKE_DLL`.

Auto Glue maintains a cache of previously built glue functions, and will reuse them on subsequent requests, even between IDL sessions. Glue function libraries can be recognized by their name, which starts with the prefix `idl_ce`, and ends with the proper suffix for a sharable library on the target system (most UNIX: `.so`, AIX: `.a`, HP-UX: `.sl`, Windows: `.dll`). `CALL_EXTERNAL` finds a suitable glue function by performing the following steps in order, stopping after the first one that works:

1. Look for a `ce_glue` subdirectory within the IDL distribution `bin` subdirectory for the current platform. (For example, on a Windows system the subdirectory would be located in `<IDL_DEFAULT>\bin\bin.x86`.) If this directory exists, it looks there for a sharable library containing the appropriate glue function.

Note

For customer security reasons, the `ce_glue` subdirectory does not exist in the IDL distribution as shipped by RSI, and IDL does not use it to create glue functions. However, if an individual site creates this directory and places glue library files within it, IDL will use them. Multiple IDL sessions on a given system can all share these same glue files, even when run by different users on a multi-user system. If you keep your IDL distribution on a network based file server shared by multiple clients, and if you provide a sufficient selection of glue files, it is possible that your users will not require a locally installed C compiler to use Auto Glue.

If you do create the `ce_glue` subdirectory on a multi-user system, we recommend that you make it along with all files contained within belong to the owner of the IDL distribution, and apply file protections that prevent non-privileged users from creating files in the directory or modifying them.

2. Look in the directory given by the `COMPILE_DIRECTORY` keyword, or if `COMPILE_DIRECTORY` is not present, in the directory given by the `!MAKE_DLL.COMPILE_DIRECTORY` system variable for the appropriate glue function.
3. If this step is reached, there is no pre-existing glue function available. `CALL_EXTERNAL` will create one in the same directory searched in the previous step by generating a C language file containing the needed glue function, and then compiling and linking it into a sharable library using the functionality of the `MAKE_DLL` procedure.
 - IDL loads the sharable library containing the glue function found in the previous step, as well as the library you specified with the `Image` argument.
 - `CALL_EXTERNAL` calls the glue function, causing your function to be called with the correct parameters.

The first time `CALL_EXTERNAL` encounters the need for a glue function that does not already exist, it will automatically build it, and then use it without any external indication that this has happened. You may notice a brief hesitation in IDL's execution as it waits for this process to occur. Once a glue function exists, IDL can load it immediately on subsequent calls (even in unrelated later IDL sessions), and no delay will occur.

Example: Using Auto Glue To Call System Library Routines

Under Sun Solaris, there is a function in the system math library called `hypot()` that computes the length of the hypotenuse of a right-angled triangle:

```
sqrt(x*x + y*y)
```

This function has the C prototype:

```
double hypot(double x, double y)
```

The following IDL function uses Auto Glue to call this routine:

```
FUNCTION HYPOT, X, Y
  ; Use the 32-bit or the 64-bit math library?
  LIBM=(!VERSION.MEMORY_BITS EQ 64) $
    ? '/usr/lib/sparcv9/libm.so' : '/usr/lib/libm.so'
  RETURN, CALL_EXTERNAL(LIBM, 'hypot', double(x), double(y), $
    /ALL_VALUE, /D_VALUE, /AUTO_GLUE)
END
```

Important Changes Since IDL 5.0

The current version of `CALL_EXTERNAL` differs from IDL versions up to and including IDL 5.0 in a few ways that are important to users moving code to the current version:

- Under Windows, `CALL_EXTERNAL` would pass IDL strings by value no matter how the `ALL_VALUE` or `VALUE` keywords were set. This was inconsistent with all the other platforms and created unnecessary confusion. IDL now uses these keywords to decide how to pass strings on all platforms. Windows users with existing code that expects strings to be passed by value without having specified it via one of these keywords will need to adjust their use of `CALL_EXTERNAL` or their code.
- Older versions of IDL would quietly pass by value arguments that are larger than a pointer without issuing an error when using the portable calling convention. Although this might work on some hardware, it is error prone and can cause IDL to crash. IDL now issues an error in this case. Programmers with existing code moving to a current version of IDL should change their code to pass such data by reference.

Examples

See [Chapter 9, “CALL_EXTERNAL”](#) in the *External Development Guide* manual.

Version History

Introduced: Pre 4.0

See Also

[LINKIMAGE](#)

CALL_FUNCTION

CALL_FUNCTION function calls the IDL function specified by the string *Name*, passing any additional parameters as its arguments.

Although not as flexible as the EXECUTE function, CALL_FUNCTION is much faster. Therefore, CALL_FUNCTION should be used in preference to EXECUTE whenever possible.

Syntax

$$Result = CALL_FUNCTION(Name [, P_1, \dots, P_n])$$

Return Value

The result of the called function (specified by the string *Name*) is passed back as the result of this routine.

Arguments

Name

A string containing the name of the function to be called. This argument can be a variable, which allows the called function to be determined at runtime.

P_i

The arguments to be passed to the function given by *Name*. These arguments are the positional and keyword arguments documented for the called function, and are passed to the called function exactly as if it had been called directly.

Keywords

None.

Examples

The following command indirectly calls the IDL function SQRT (the square root function) with an argument of 4 and stores the result in the variable R:

```
R = CALL_FUNCTION('SQRT', 4)
```

Version History

Introduced: Pre 4.0

See Also

[CALL_PROCEDURE](#), [CALL_METHOD](#), [EXECUTE](#)

CALL_METHOD

The CALL_METHOD function or procedure calls the object method specified by *Name*, passing any additional parameters as its arguments.

Note

CALL_METHOD can also be used as a function or a procedure.

Although not as flexible as the EXECUTE function, CALL_METHOD is much faster. Therefore, CALL_METHOD should be used in preference to EXECUTE whenever possible.

Syntax

Result = CALL_METHOD(*Name*, *ObjRef*, [, *P*₁, ..., *P*_{*n*}])

or

CALL_METHOD, *Name*, *ObjRef*, [, *P*₁, ..., *P*_{*n*}]

Return Value

Returns the results generated by the named function method when applicable.

Arguments

Name

A string containing the name of the method to be called. This argument can be a variable, which allows the called method to be determined at runtime.

ObjRef

A scalar object reference that will be passed to the method as the *Self* argument.

*P*_{*i*}

The arguments to be passed to the method given by *Name*. These arguments are the positional and keyword arguments documented for the called method, and are passed to the called method exactly as if it had been called directly.

Keywords

None.

Version History

Introduced: 5.1

See Also

[CALL_FUNCTION](#), [CALL_PROCEDURE](#), [EXECUTE](#)

CALL_PROCEDURE

CALL_PROCEDURE calls the procedure specified by *Name*, passing any additional parameters as its arguments.

Although not as flexible as the EXECUTE function, CALL_PROCEDURE is much faster. Therefore, CALL_PROCEDURE should be used in preference to EXECUTE whenever possible.

Syntax

```
CALL_PROCEDURE, Name [, P1, ..., Pn]
```

Arguments

Name

A string containing the name of the procedure to be called. This argument can be a variable, which allows the called procedure to be determined at runtime.

*P*_{*i*}

The arguments to be passed to the procedure given by *Name*. These arguments are the positional and keyword arguments documented for the called procedure, and are passed to the called procedure exactly as if it had been called directly.

Example

The following example shows how to call the PLOT procedure indirectly with a number of arguments. First, create a dataset for plotting by entering:

```
B = FINDGEN(100)
```

Call PLOT indirectly to create a polar plot by entering:

```
CALL_PROCEDURE, 'PLOT', B, B, /POLAR
```

A “spiral” plot should appear.

Version History

Introduced: Pre 4.0

See Also

[CALL_FUNCTION](#), [CALL_METHOD](#), [EXECUTE](#)

CASE

The CASE statement selects one, and only one, statement for execution, depending on the value of an expression. This expression is called the case selector expression. Each statement that is part of a CASE statement is preceded by an expression that is compared to the value of the selector expression. CASE executes by comparing the CASE expression with each selector expression in the order written. If a match is found, the statement is executed and control resumes directly below the CASE statement.

The ELSE clause of the CASE statement is optional. If included, it matches any selector expression, causing its code to be executed. For this reason, it is usually written as the last clause in the CASE statement. The ELSE statement is executed only if none of the preceding statement expressions match. If an ELSE clause is not included and none of the values match the selector, an error occurs and program execution stops.

The BREAK statement can be used within CASE statements to force an immediate exit from the CASE.

In this CASE statement, only one clause is selected, and that clause is the first one whose value is equal to the value of the case selector expression.

Tip

Each clause is tested in order, so it is most efficient to order the most frequently selected clauses first.

CASE is similar to the SWITCH statement. For more information on using CASE and other IDL program control statements, as well as the differences between CASE and SWITCH, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

```
CASE expression OF
    expression: statement
    ...
    expression: statement
[ ELSE: statement ]
ENDCASE
```

Examples

This example illustrates how the CASE statement, unlike SWITCH, executes only the one statement that matches the case expression:

```
x=2  
  
CASE x OF  
  1: PRINT, 'one'  
  2: PRINT, 'two'  
  3: PRINT, 'three'  
  4: PRINT, 'four'  
ENDCASE
```

IDL Prints:

```
two
```

Version History

Introduced: Original

CATCH

The CATCH procedure provides a generalized mechanism for the handling of exceptions and errors within IDL. Calling CATCH establishes an error handler for the current procedure that intercepts all errors that can be handled by IDL, excluding non-fatal warnings such as math errors.

When an error occurs, each active procedure, beginning with the offending procedure and proceeding up the call stack to the main program level, is examined for an error handler. If an error handler is found, control resumes at the statement after the call to CATCH. The index of the error is returned in the argument to CATCH. The `!ERROR_STATE` system variable is also set. If no error handlers are found, program execution stops, an error message is issued, and control reverts to the interactive mode. A call to `ON_IOERROR` in the procedure that causes an I/O error supersedes CATCH, and takes the branch to the label defined by `ON_IOERROR`.

This mechanism is similar, but not identical to, the `set jmp/long jmp` facilities in C and the `catch/throw` facilities in C++.

Error handling is discussed in more detail in [Chapter 19, “Controlling Errors”](#) in the *Building IDL Applications* manual.

Syntax

```
CATCH, [Variable] [, /CANCEL]
```

Arguments

Variable

A named variable in which the error index is returned. When an error handler is established by a call to CATCH, *Variable* is set to zero. If an error occurs, *Variable* is set to the error index, and control is transferred to the statement after the call to CATCH. The error index is also returned in the CODE field of the `!ERROR_STATE` system variable, i.e., `!ERROR_STATE.CODE`.

Keywords

CANCEL

Set this keyword to cancel the error handler for the current procedure. This cancellation does not affect other error handlers that may be established in other active procedures.

Note

If the CANCEL keyword is set, the *Variable* argument must not be present.

Examples

The following procedure illustrates the use of CATCH:

```

PRO CATCH_EXAMPLE

    ; Define variable A:
    A = FLTARR(10)

    ; Establish error handler. When errors occur, the index of the
    ; error is returned in the variable Error_status:
    CATCH, Error_status

    ;This statement begins the error handler:
    IF Error_status NE 0 THEN BEGIN
        PRINT, 'Error index: ', Error_status
        PRINT, 'Error message: ', !ERROR_STATE.MSG
        ; Handle the error by extending A:
        A=FLTARR(12)
        CATCH, /CANCEL
    ENDIF

    ; Cause an error:
    A[11]=12

    ; Even though an error occurs in the line above, program
    ; execution continues to this point because the event handler
    ; extended the definition of A so that the statement can be
    ; re-executed.
    HELP, A
END

```

Running the ABC procedure causes IDL to produce the following output and control returns to the interactive prompt:

```
Error index:          -144
Error message:
Attempt to subscript A with <INT (      11)> is out of range.
A                FLOAT      = Array[12]
```

Version History

Introduced: Pre 4.0

See Also

[!ERROR_STATE](#), [ON_ERROR](#), [ON_IOERROR](#), Chapter 19, “Controlling Errors” in the *Building IDL Applications* manual.

CD

The CD procedure is used to set and/or change the current working directory. This routine changes the working directory for the IDL session and any child processes started from IDL during that session after the directory change is made. Under UNIX, CD does not affect the working directory of the process that started IDL. The PUSH, POP, and PRINTD procedures provide a convenient interface to CD.

Syntax

```
CD [, Directory] [, CURRENT=variable]
```

Arguments

Directory

A scalar string specifying the path of the new working directory. If *Directory* is specified as a null string, the working directory is changed to the user's home directory (UNIX) or to the directory specified by !DIR (Windows). If this argument is not specified, the working directory is not changed.

Keywords

CURRENT

If CURRENT is present, it specifies a named variable into which the current working directory is stored as a scalar string. The returned directory is the working directory before the directory is changed. Thus, you can obtain the current working directory and change it in a single statement:

```
CD, new_dir, CURRENT=old_dir
```

Note

The return value of the CURRENT keyword does not include a directory separator at the end of the string.

Examples

Windows

To change drives:

```
CD, 'C:'
```

To specify a full path:

```
CD, 'C:\MyData\January'
```

To change from the C:\MyData directory to the C:\MyData\January directory:

```
CD, 'January'
```

To go back up a directory, use “..”. For example, if the current directory is C:\MyData\January, you could go up to the C:\MyData directory with the following command:

```
CD, '..'
```

If the current directory is C:\MyData\January, you could change to the C:\MyData\February directory with the following command:

```
CD, '..\February'
```

Unix

To specify a full path:

```
CD, '/home/data/'
```

To change to the january subdirectory of the current directory:

```
CD, 'january'
```

To go back up a directory, use “..”. For example, if the current directory is /home/data/january, you could go up to the /home/data/ directory with the following command:

```
CD, '..'
```

If the current directory is /home/data/january, you could change to the /home/data/february directory with the following command:

```
CD, '../february'
```

Version History

Introduced: Pre 4.0

See Also

[PUSHD](#), [POPD](#)

CDF Routines

For information, see [Chapter 2, “Common Data Format”](#) in the *IDL Scientific Data Formats* manual.

CEIL

The CEIL function returns the closest integer greater than or equal to its argument.

Syntax

$$Result = CEIL(X [, /L64])$$

Return Value

If the input value X is integer type, $Result$ has the same value and type as X . Otherwise, $Result$ is a 32-bit longword integer with the same structure as X .

Arguments

X

The value for which the ceiling function is to be evaluated. This value can be any numeric type (integer, floating, or complex).

Keywords

L64

If set, the result type is 64-bit integer regardless of the input type. This is useful for situations in which a floating point number contains a value too large to be represented in a 32-bit integer.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To print the ceiling function of 5.1, enter:

```
PRINT, CEIL(5.1)
; IDL prints:
6
```

To print the ceiling function of 3000000000.1, the result of which is too large to represent in a 32-bit integer:

```
PRINT, CEIL(3000000000.1D, /L64)
; IDL prints:
3000000001
```

Version History

Introduced: Pre 4.0

See Also

[COMPLEXROUND](#), [FLOOR](#), [ROUND](#)

CHEBYSHEV

The CHEBYSHEV function returns the forward or reverse Chebyshev polynomial expansion of a set of data. Note: Results from this function are subject to roundoff error given discontinuous data.

This routine is written in the IDL language. Its source code can be found in the file `chebyshev.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CHEBYSHEV(*D*, *N*)

Return Value

Returns the forward or reverse Chebyshev polynomial expansion of a set of data.

Arguments

D

A vector containing the values at the zeros of Chebyshev polynomial.

N

A flag that, if set to -1, returns a set of Chebyshev polynomials. If set to +1, the original data is returned.

Keywords

None.

Version History

Introduced: Original

See Also

[FFT](#), [WTN](#)

CHECK_MATH

The CHECK_MATH function returns and clears the accumulated math error status.

Syntax

Result = CHECK_MATH([, MASK=bitmask] [, /NOCLEAR] [, /PRINT])

Return Value

The returned value is the sum of the bit values (described in the following table) of the accumulated errors. Note that not all machines detect all errors.

Value	Condition
0	No errors detected since the last interactive prompt or call to CHECK_MATH
1	Integer divided by zero
2	Integer overflow
16	Floating-point divided by zero
32	Floating-point underflow
64	Floating-point overflow
128	Floating-point operand error. An illegal operand was encountered, such as a negative operand to the SQRT or ALOG functions, or an attempt to convert to integer a number whose absolute value is greater than $2^{31} - 1$

Table 3-2: Math Error Status Values

Note that each type of error is only represented once in the return value—any number of “Integer divided by zero” errors will result in a return value of 1.

The math error status is cleared (reset to zero) when CHECK_MATH is called, or when errors are reported. Math errors are reported either never, when the interpreter returns to an interactive prompt, or after execution of each IDL statement, depending on the value of the !EXCEPT system variable (see “!EXCEPT” on page 4361). See “Examples” below for further discussion.

Keywords

MASK

If present, the mask of exceptions to check. Otherwise, all exceptions are checked. Exceptions that are pending but not specified by MASK are not reported, and not cleared. Set this keyword equal to the sum of the bit values for each exception to be checked. For a list of the bit values corresponding to various exceptions, see [CHECK_MATH](#).

NOCLEAR

By default, CHECK_MATH returns the pending exceptions (as specified via the MASK keyword) and clears them from its list of pending exceptions. If NOCLEAR is set, the exceptions are not cleared and remain pending.

PRINT

Set this keyword to print an error message to the IDL command log if any accumulated math errors exist. If this keyword is not present, CHECK_MATH executes silently.

Examples

To simply check and clear the accumulated math error status using all the defaults, enter:

```
PRINT, CHECK_MATH( )
```

IDL prints the accumulated math error status code and resets to zero.

CHECK_MATH and !EXCEPT

Because the accumulated math error status is cleared when it is reported, the behavior and appropriate use of CHECK_MATH depends on the value of the system variable !EXCEPT.

- If !EXCEPT is set equal to 0, math exceptions are not reported automatically, and thus CHECK_MATH will always return the error status accumulated since the last time it was called.
- If !EXCEPT is set equal to 1, math exceptions are reported when IDL returns to the interactive command prompt. In this case, CHECK_MATH will return appropriate error codes when used *within* an IDL procedure, but will always return zero when called at the IDL prompt.

- If !EXCEPT is set equal to 2, math exceptions are reported after each IDL statement. In this case, CHECK_MATH will return appropriate error codes only when used *within an IDL statement*, and will always return zero otherwise.

For example:

```
;Set value of !EXCEPT to zero.
!EXCEPT=0

;Both of these operations cause errors.
PRINT, 1./0., 1/0
```

IDL prints:

```
Inf    1
```

The special floating-point value Inf is returned for 1./0. There is no integer analogue to the floating-point Inf.

```
;Check the accumulated error status.
PRINT, CHECK_MATH()
```

IDL prints:

```
17
```

CHECK_MATH reports floating-point and integer divide-by-zero errors.

```
;Set value of !EXCEPT to one.
!EXCEPT=1

;Both of these operations cause errors.
PRINT, 1./0., 1/0
```

IDL prints:

```
Inf    1
% Program caused arithmetic error: Integer divide by 0
% Program caused arithmetic error: Floating divide by 0
```

This time IDL also prints error messages.

```
;Check the accumulated error status.
PRINT, CHECK_MATH()
```

IDL prints:

```
0
```

The status was reset.

However, if we do not allow IDL to return to an interactive prompt before checking the math error status:

```
;Set value of !EXCEPT to one.
!EXCEPT=1

;Call to CHECK_MATH happens before returning to the
;IDL command prompt.
PRINT, 1./0., 1/0 & PRINT, CHECK_MATH()
```

IDL prints:

```
Inf      1
17
```

In this case, the math error status code (17) is printed, but because the error status has been cleared by the call to `CHECK_MATH`, no error messages are printed when IDL returns to the interactive command prompt. Finally,

```
;Set value of !EXCEPT to two.
!EXCEPT=2

;Call to CHECK_MATH happens before returning to the
;IDL command prompt.
PRINT, 1./0., 1/0 & PRINT, CHECK_MATH()
```

IDL prints:

```
Inf      1
% Program caused arithmetic error: Integer divide by 0
% Program caused arithmetic error: Floating divide by 0
% Detected at $MAIN$
0
```

Errors are printed before executing the `CHECK_MATH` function, so `CHECK_MATH` reports no errors. However, if we include the call to `CHECK_MATH` in the first `PRINT` command, we see the following:

```
;Call to CHECK_MATH is part of a single IDL statement.
PRINT, 1./0., 1/0, CHECK_MATH()
```

IDL prints:

```
Inf      1      17
```

Printing Error Messages

The following code fragment prints abbreviated names of errors that have occurred:

```

;Create a string array of error names.
ERRS = ['Divide by 0', 'Underflow', 'Overflow', $
        'Illegal Operand']

;Get math error status.
J = CHECK_MATH()
FOR I = 4, 7 DO IF ISHFT(J, -I) AND 1 THEN $

;Check to see if an error occurred and print the corresponding
;error message.
    PRINT, ERRS(I-4), ' Occurred'
```

Testing Critical Code

Example 1

Assume you have a critical section of code that is likely to produce an error. The following example shows how to check for errors, and if one is detected, how to repeat the code with different parameters.

```

; Clear error status from previous operations, and print error
; messages if an error exists:
JUNK = CHECK_MATH(/PRINT)

; Disable automatic printing of subsequent math errors:
!EXCEPT=0

;Critical section goes here.
AGAIN: ...

; Did an arithmetic error occur? If so, print error message and
; request new values:
IF CHECK_MATH() NE 0 THEN BEGIN
PRINT, 'Math error occurred in critical section.'

; Get new parameters from user:
READ, 'Enter new values.',...

; Enable automatic printing of math errors:
!EXCEPT=2

;And retry:
GOTO, AGAIN
ENDIF
```

Example 2

This example demonstrates the use of the MASK keyword to check for a specific error, and the NOCLEAR keyword to prevent exceptions from being cleared:

```
PRO EXAMPLE2_CHECKMATH

    PRINT, 1./0
    PRINT, CHECK_MATH(MASK=16, /NOCLEAR)
    PRINT, CHECK_MATH(MASK=2, /NOCLEAR)

END
```

IDL prints:

```
Inf
16
0
% Program caused arithmetic error: Floating divide by 0
```

Version History

Introduced: Original

See Also

[FINITE](#), [ISHFT](#), [MACHAR](#), ["!VALUES"](#) on page 4357, ["!EXCEPT"](#) on page 4361, ["Math Errors"](#) in Chapter 19 of the *Building IDL Applications* manual.

CHISQR_CVF

The CHISQR_CVF function computes the cutoff value V in a Chi-square distribution with Df degrees of freedom such that the probability that a random variable X is greater than V is equal to a user-supplied probability P .

This routine is written in the IDL language. Its source code can be found in the file `chisqr_cvf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CHISQR_CVF(*P*, *Df*)

Return Value

Returns computes the cutoff value V in a Chi-square distribution with Df degrees of freedom such that the probability that a random variable X is greater than V is equal to a user-supplied probability P .

Arguments

P

A non-negative single- or double-precision floating-point scalar, in the interval [0.0, 1.0], that specifies the probability of occurrence or success.

Df

A positive integer, single- or double-precision floating-point scalar that specifies the number of degrees of freedom of the Chi-square distribution.

Keywords

None.

Examples

Use the following command to compute the cutoff value in a Chi-square distribution with three degrees of freedom such that the probability that a random variable X is greater than the cutoff value is 0.100. The result should be 6.25139.

```
PRINT, CHISQR_CVF(0.100, 3)
```

IDL prints:

6.25139

Version History

Introduced: 4.0

See Also

[CHISQR_PDF](#), [F_CVF](#), [GAUSS_CVF](#), [T_CVF](#)

CHISQR_PDF

The CHISQR_PDF function computes the probability P that, in a Chi-square distribution with Df degrees of freedom, a random variable X is less than or equal to a user-specified cutoff value V .

This routine is written in the IDL language. Its source code can be found in the file `chisqr_pdf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CHISQR_PDF(*V*, *Df*)

Return Value

If both arguments are scalar, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of V and Df , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

If any of the arguments are double-precision, the result is double-precision, otherwise the result is single-precision.

Arguments

V

A scalar or array that specifies the cutoff value(s).

Df

A positive scalar or array that specifies the number of degrees of freedom of the Chi-square distribution.

Keywords

None.

Examples

Use the following command to compute the probability that a random variable X , from the Chi-square distribution with three degrees of freedom, is less than or equal to 6.25. The result should be 0.899939.

```
result = CHISQR_PDF(6.25, 3)
PRINT, result
```

IDL prints:

```
0.899939
```

Compute the probability that a random variable X from the Chi-square distribution with three degrees of freedom, is greater than 6.25. The result should be 0.100061.

```
PRINT, 1 - chisqr_pdf(6.25, 3)
```

IDL prints:

```
0.100061
```

Version History

Introduced: 4.0

See Also

[BINOMIAL](#), [CHISQR_CVF](#), [F_PDF](#), [GAUSS_PDF](#), [T_PDF](#)

CHOLDC

Given a positive-definite symmetric n by n array A , the CHOLDC procedure constructs its Cholesky decomposition $A = LL^T$, where L is a lower triangular array and L^T is the transpose of L .

CHOLDC is based on the routine `choldc` described in section 2.9 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the `LA_CHOLDC` procedure.

Syntax

CHOLDC, A , P [, /DOUBLE]

Arguments

A

An n by n array. On input, only the upper triangle of A need be given. On output, L is returned in the lower triangle of A , except for the diagonal elements, which are returned in the vector P .

P

An n -element vector containing the diagonal elements of L .

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

See “[CHOLSOL](#)” on page 257.

Version History

Introduced: 4.0

See Also

[CHOLSOL](#), [LA_CHOLDC](#)

CHOLSOL

The CHOLSOL function returns an n -element vector containing the solution to the set of linear equations $Ax = b$, where A is the positive-definite symmetric array returned by the CHOLDC procedure.

CHOLSOL is based on the routine `chols1` described in section 2.9 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the LA_CHOLSOL procedure.

Syntax

Result = CHOLSOL(*A*, *P*, *B* [, /DOUBLE])

Return Value

Returns an n -element vector containing the solution to the set of linear equations $Ax = b$, where A is the positive-definite symmetric array returned by the CHOLDC.

Arguments

A

An n by n positive-definite symmetric array, as output by CHOLDC. Only the lower triangle of A is accessed.

P

The diagonal elements of the Cholesky factor L , as computed by CHOLDC.

B

An n -element vector containing the right-hand side of the equation.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To solve a positive-definite symmetric system $Ax = b$:

```
;Define the coefficient array:
A = [[ 6.0, 15.0, 55.0], $
     [15.0, 55.0, 225.0], $
     [55.0, 225.0, 979.0]]

;Define the right-hand side vector B:
B = [9.5, 50.0, 237.0]

;Compute Cholesky decomposition of A:
CHOLDC, A, P

;Compute and print the solution:
PRINT, CHOLSOL(A, P, B)
```

IDL prints:

```
-0.499998  -1.00000  0.500000
```

The exact solution vector is [-0.5, -1.0, 0.5].

Version History

Introduced: 4.0

See Also

[CHOLDC](#), [CRAMER](#), [GS_ITER](#), [LA_CHOLSOL](#), [LU_COMPLEX](#), [LUSOL](#), [SVSOL](#), [TRISOL](#)

CINDGEN

The CINDGEN function returns a complex, single-precision, floating-point array with the specified dimensions.

Syntax

$$\text{Result} = \text{CINDGEN}(D_1[, \dots, D_8])$$

Return Value

Returns a complex, single-precision, floating-point array with the specified dimensions. Each element of the array has its real part set to the value of its one-dimensional subscript. The imaginary part is set to zero.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create C, a 4-element vector of complex values with the real parts set to the value of their subscripts, enter:

```
C = CINDGEN(4)
```

Version History

Introduced: Original

See Also

[BINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#), [LINDGEN](#),
[SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

CIR_3PNT

The CIR_3PNT procedure returns the radius and center of a circle, given 3 points on the circle. This is analogous to finding the circumradius and circumcircle of a triangle; the center of the circumcircle is the point at which the three perpendicular bisectors of the triangle formed by the points meet.

This routine is written in the IDL language. Its source code can be found in the file `cir_3pnt.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
CIR_3PNT, X, Y, R, X0, Y0
```

Arguments

X

A three-element vector containing the X-coordinates of the points.

Y

A three-element vector containing the Y-coordinates of the points.

R

A named variable that will contain the radius of the circle. The procedure returns 0.0 if the points are co-linear.

X0

A named variable that will contain the X-coordinate of the center of the circle. The procedure returns 0.0 if the points are co-linear.

Y0

A named variable that will contain the Y-coordinate of the center of the circle. The procedure returns 0.0 if the points are co-linear.

Keywords

None.

Examples

```
X = [1.0, 2.0, 3.0]
Y = [1.0, 2.0, 1.0]
CIR_3PNT, X, Y, R, X0, Y0
PRINT, 'The radius is: ', R
PRINT, 'The center of the circle is at: ', X0, Y0
```

Version History

Introduced: Pre 4.0

See Also

[PNT_LINE](#), [SPH_4PNT](#)

CLOSE

The CLOSE procedure closes the file units specified as arguments. All open files are also closed when IDL exits.

Syntax

```
CLOSE[, Unit1, ..., Unitn] [, /ALL] [, EXIT_STATUS=variable] [, /FILE]  
[, /FORCE]
```

Arguments

Unit_{*i*}

The IDL file units to close.

Keywords

ALL

Set this keyword to close all open file units. In addition, any file units that were allocated via GET_LUN are freed.

EXIT_STATUS

Set this keyword to a named variable that will contain the exit status reported by a UNIX child process started via the UNIT keyword to SPAWN. This value is the exit value reported by the process by calling EXIT, and is analogous to the value returned by \$? under most UNIX shells. If used with any other type of file, 0 is returned. EXIT_STATUS is not allowed in conjunction with the ALL or FILE keywords.

FILE

Set this keyword to close all file units from 1 to 99. File units greater than 99, which are associated with the GET_LUN and FREE_LUN procedures, are not affected.

FORCE

Overrides the IDL file output buffer and forces the file to be closed no matter what errors occur in the process.

IDL buffers file output for performance reasons. If it is not possible to properly flush this data when a file close is requested, an error is normally issued and the file

remains open. An example of this might be that your disk does not have room to write the remaining data. This default behavior prevents data from being lost. To override it and force the file to be closed no matter what errors occur in the process, specify `FORCE`.

Examples

If file units 1 and 3 are open, they can both be closed at the same time by entering the command:

```
CLOSE, 1, 3
```

Version History

Introduced: Original

See Also

[OPEN](#)

CLUST_WTS

The CLUST_WTS function computes the weights (the cluster centers) of an n -column, m -row array, where n is the number of variables and m is the number of observations or samples. CLUST_WTS uses k -means clustering. With this technique, CLUST_WTS starts with k random clusters and then iteratively moves items between clusters, minimizing variability within each cluster and maximizing variability between clusters.

Note

Because the initial clusters are chosen randomly, your results may differ slightly each time the CLUST_WTS routine is invoked, even for the same input data. For data with well-defined clusters the differences should be slight. For randomly-scattered data (no distinguishable clusters), the results may be significantly different, which may indicate that k -means clustering is not appropriate for your data.

Tip

For hierarchical tree clustering, see the [CLUSTER_TREE](#) function.

For more information on cluster analysis, see:

Everitt, Brian S. *Cluster Analysis*. New York: Halsted Press, 1993. ISBN 0-470-22043-0

Syntax

```
Result = CLUST_WTS( Array [, /DOUBLE] [, N_CLUSTERS=value]  
                  [, N_ITERATIONS=integer] [, VARIABLE_WTS=vector] )
```

Return Value

Returns an m -column, $N_CLUSTERS$ -row array of cluster centers by computing the weights (the cluster centers) of an n -column, m -row array, where n is the number of variables and m is the number of observations or samples.

Arguments

Array

An n -column, m -row array of any data type except string, single- or double-precision complex.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

N_CLUSTERS

Set this keyword equal to the number of cluster centers. The default is to compute m cluster centers.

N_ITERATIONS

Set this keyword equal to the number of iterations used when in computing the cluster centers. The default is to use 20 iterations.

VARIABLE_WTS

Set this keyword equal to an m -element vector of floating-point variable weights. The elements of this vector are used to give greater or lesser importance to each variable (each column) in determining the cluster centers. The default is to give all variables equal weighting using a value of 1.0.

Examples

See [“CLUSTER”](#) on page 267.

Version History

Introduced: 5.0

See Also

[CLUSTER](#), “[Multivariate Analysis](#)” in Chapter 22 of the *Using IDL* manual.,
[CLUSTER_TREE](#)

CLUSTER

The CLUSTER function computes the classification of an n -column, m -row array, where n is the number of variables and m is the number of observations or samples. CLUST_WTS uses k -means clustering. With this technique, CLUST_WTS starts with k random clusters and then iteratively moves items between clusters, minimizing variability within each cluster and maximizing variability between clusters.

Note

Because the initial clusters are chosen randomly, your results may differ slightly each time the CLUST_WTS routine is invoked, even for the same input data. For data with well-defined clusters the differences should be slight. For randomly-scattered data (no distinguishable clusters), the results may be significantly different, which may indicate that k -means clustering is not appropriate for your data.

Tip

For hierarchical tree clustering, see the [CLUSTER_TREE](#) function.

For more information on cluster analysis, see:

Everitt, Brian S. *Cluster Analysis*. New York: Halsted Press, 1993. ISBN 0-470-22043-0

Syntax

Result = CLUSTER(*Array*, *Weights* [, /DOUBLE] [, N_CLUSTERS=*value*])

Return Value

Results in a 1-column, m -row array of cluster number assignments that correspond to each sample.

Arguments

Array

An n -column, m -row array of type float or double.

Weights

An array of weights (the cluster centers) computed using the `CLUST_WTS` function. The dimensions of this array vary according to keyword values.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

N_CLUSTERS

Set this keyword equal to the number of clusters. The default is based upon the row dimension of the *Weights* array.

Examples

```

; Construct 3 separate clusters in a 3D space:
n = 50
c1 = RANDOMN(seed, 3, n)
c1[0:1,*] - = 3
c2 = RANDOMN(seed, 3, n)
c2[0,*] + = 3
c2[1,*] - = 3
c3 = RANDOMN(seed, 3, n)
c3[1:2,*] + = 3
array = [[c1], [c2], [c3]]

; Compute cluster weights, using three clusters:
weights = CLUST_WTS(array, N_CLUSTERS = 3)

; Compute the classification of each sample:
result = CLUSTER(array, weights, N_CLUSTERS = 3)

; Plot each cluster using a different symbol:
IPLOT, array[* , WHERE(result eq 0)], $
    LINESTYLE = 6, SYM_INDEX = 2
IPLOT, array[* , WHERE(result eq 1)], /OVERPLOT, $
    LINESTYLE = 6, SYM_INDEX = 4
IPLOT, array[* , WHERE(result eq 2)], /OVERPLOT, $
    LINESTYLE = 6, SYM_INDEX = 1

```

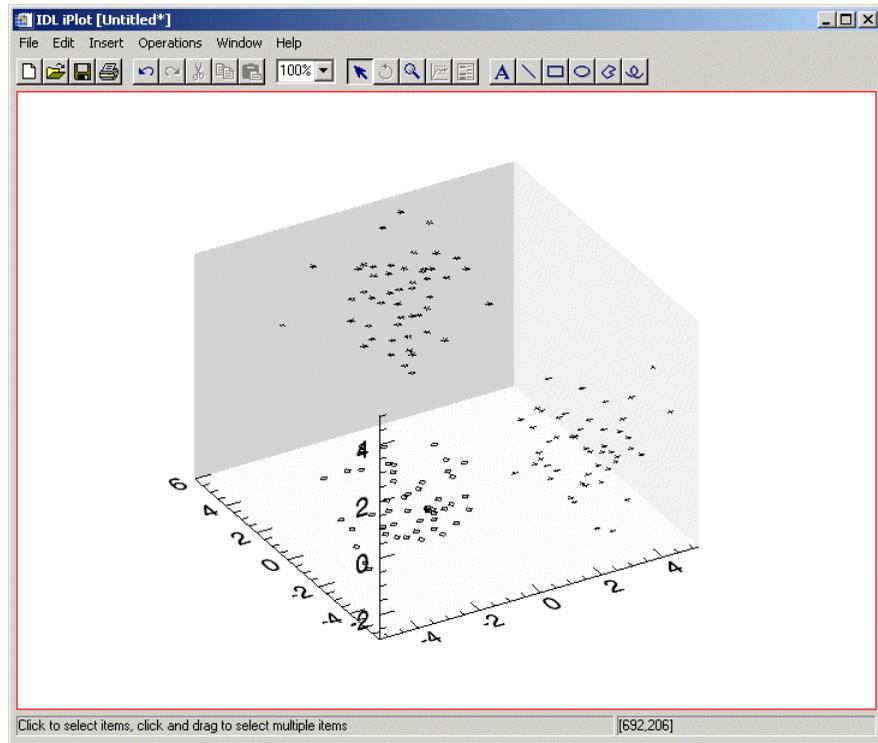


Figure 1: CLUSTER Example

Version History

Introduced: 5.0

See Also

[CLUST_WTS](#), [CLUSTER_TREE](#), [PCOMP](#), [STANDARDIZE](#), “Multivariate Analysis” in Chapter 22 of the *Using IDL* manual.

CLUSTER_TREE

The CLUSTER_TREE function computes the hierarchical clustering for a set of m items in an n -dimensional space. The CLUSTER_TREE function is designed to be used with the DENDROGRAM or DENDRO_PLOT procedures.

This routine is written in the IDL language. Its source code can be found in the file `cluster_tree.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CLUSTER_TREE(*Pairdistance*, *Linkdistance* [, LINKAGE = value])

or for LINKAGE = 3 (centroid):

Result = CLUSTER_TREE(*Pairdistance*, *Linkdistance*, LINKAGE = 3,
[, DATA = array] [, MEASURE=value] [, POWER_MEASURE=value])

Return Value

The *Result* is a 2-by- $(m-1)$ integer array containing the cluster indices. Each row of *Result* contains the indices of the two items that were clustered together. The distance between the two items is contained in the corresponding element of the *Linkdistance* output argument.

Note

The original m items are given indices 0... $m-1$, while each newly-created cluster is given a new index starting at m and incrementing.

Arguments

Pairdistance

An input array containing the pairwise distances as either a compact vector or as a symmetric matrix, usually created by the DISTANCE_MEASURE function. For the compact vector form, *Pairdistance* should be an $m*(m-1)/2$ element vector, ordered as: $[D_{0,1}, D_{0,2}, \dots, D_{0,m-1}, D_{1,2}, \dots, D_{m-2,m-1}]$, where $D_{i,j}$ denotes the distance between items i and j . For the matrix form, *Pairdistance* should be an m -by- m symmetric matrix with zeroes down the diagonal.

Linkdistance

Set this argument to a named variable in which the cluster distances will be returned as an $(m-1)$ -element single or double-precision vector. Each element of *Linkdistance* corresponds to the distance between the two items of the corresponding row in *Result*. If *Pairdistance* is double-precision then *Linkdistance* will be double-precision, otherwise *Linkdistance* will be single-precision.

Keywords

DATA

If the LINKAGE keyword is set equal to 3 (centroid), then the DATA keyword must be set to the array of original data as input to the DISTANCE_MEASURE function. The data array is necessary for computing the centroid of newly-created clusters.

Note

DATA does not need to be supplied if LINKAGE is not equal to 3.

LINKAGE

Set this keyword to an integer giving the method used for linking clusters together. Possible values are:

Value	Method Used
0	(Default) Use single linkage (nearest neighbor). The distance between two clusters is defined as the smallest distance between items in the two clusters. This method tends to string items together and is useful for non-homogeneous clusters.
1	Use complete linkage (furthest neighbor). The distance between two clusters is defined as the largest distance between items. This method is useful for homogeneous, compact, clusters but is not useful for long chain-like clusters.
2	Use weighted pairwise average. The distance between two clusters is defined as the average distance for all pairs of objects between each cluster, weighted by the number of objects in each cluster. This method works well for both homogeneous clusters and for chain-like clusters.

Table 1: LINKAGE Values and Methods

Value	Method Used
3	Use weighted centroid. The distance between two clusters is defined as the distance between the centroids of each cluster. The centroid of a cluster is the average position of all the subclusters, weighted by the number of objects in each subcluster.

Table 1: LINKAGE Values and Methods

Note

If the LINKAGE keyword is equal to 3, the distance between two clusters may be less than the distance between items within one of the clusters. In a dendrogram plot this will cause the node lines to overlap.

MEASURE

If the LINKAGE keyword is equal to 3 (centroid), set this keyword to an integer giving the distance measure (the metric) to use. Possible values are:

Value	Type
0	(Default) Euclidean distance
1	CityBlock (Manhattan) distance
2	Chebyshev distance
3	Correlative distance
4	Percent disagreement

Table 2: MEASURE Values

For consistent results, the MEASURE value should match the value used in the original call to DISTANCE_MEASURE. This keyword is ignored if LINKAGE is not equal to 3, or if POWER_MEASURE is set.

Note

See DISTANCE_MEASURE for a detailed description of the various metrics.

POWER_MEASURE

If the LINKAGE keyword is equal to 3 (centroid), set this keyword to a scalar or a two-element vector giving the parameters p and r to be used in the power distance metric. If POWER_MEASURE is a scalar then the same value is used for both p and r . For consistent results, the POWER_MEASURE value should match the value used in the original call to DISTANCE_MEASURE. This keyword is ignored if LINKAGE is not equal to 3.

Note

See DISTANCE_MEASURE for a detailed description of the power distance metric.

Example

```

; Given a set of points in two-dimensional space.
DATA = [ $
[1, 1], $
[1, 3], $
[2, 2.2], $
[4, 1.75], $
[4, 4], $
[5, 1], $
[5.5, 3]]

; Compute the Euclidean distance between each point.
DISTANCE = DISTANCE_MEASURE(data)

; Now compute the cluster analysis.
CLUSTERS = CLUSTER_TREE(distance, linkdistance)

PRINT, 'Item# Item# Distance'
PRINT, [clusters, TRANSPOSE(linkdistance)], $
      FORMAT='(I3, I7, F10.2)'
```

When this code is run, IDL prints:

```

Item#  Item#  Distance
5      3      1.25
2      1      1.28
8      0      1.56
6      4      1.80
7      10     1.95
11     9      2.05
```

Items 5 and 3 are joined to create a new cluster, which is given the item number of 7. Items 2 and 1 are joined to create a cluster with item number 8. The process continues until all items have been joined together. A graphical representation is shown below (for clarity the last cluster, between items 9 and 11, has been omitted):

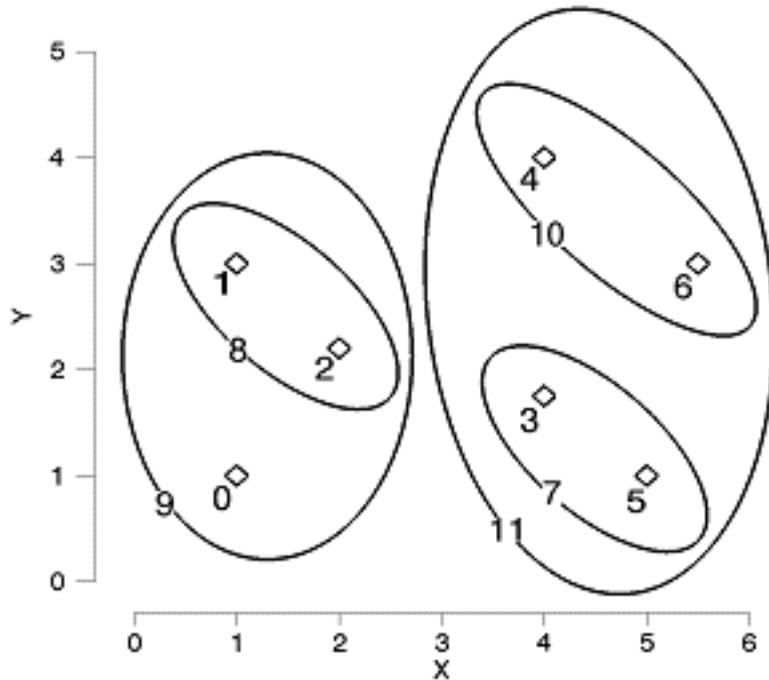


Figure 3-12: Graphical Representation of Hierarchical Clustering

Version History

Introduced: 6.1

See Also

[DENDRO_PLOT](#), [DENDROGRAM](#), [DISTANCE_MEASURE](#)

COLOR_CONVERT

The COLOR_CONVERT procedure converts colors to and from the RGB (Red Green Blue), HLS (Hue Lightness Saturation), and HSV (Hue Saturation Value) color systems. A keyword parameter indicates the type of conversion to be performed (one of the keywords must be specified). The first three parameters contain the input color triple(s) which may be scalars or arrays of the same size. The result is returned in the last three parameters, O_0 , O_1 , and O_2 . RGB values are bytes in the range 0 to 255.

Hue is measured in degrees, from 0 to 360. Saturation, Lightness, and Value are floating-point numbers in the range 0 to 1. A Hue of 0 degrees is the color red. Green is 120 degrees. Blue is 240 degrees. A reference containing a discussion of the various color systems is: Foley and Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., 1982.

Syntax

```
COLOR_CONVERT,  $I_0, I_1, I_2, O_0, O_1, O_2$  {, /HLS_RGB |, /HSV_RGB |,  
/RGB_HLS |, /RGB_HSV }
```

Arguments

I_0, I_1, I_2

The input color triple(s). These arguments may be either scalars or arrays of the same length.

O_0, O_1, O_2

The variables to receive the result. Their structure is copied from the input parameters.

Keywords

HLS_RGB

Set this keyword to convert from HLS to RGB.

HSV_RGB

Set this keyword to convert from HSV to RGB.

RGB_HLS

Set this keyword to convert from RGB to HLS.

RGB_HSV

Set this keyword to convert from RGB to HSV.

Examples

The command:

```
COLOR_CONVERT, 255, 255, 0, h, s, v, /RGB_HSV
```

converts the RGB color triple (255, 255, 0), which is the color yellow at full intensity and saturation, to the HSV system. The resulting hue in the variable h is 60.0 degrees. The saturation and value, s and v, are set to 1.0.

Version History

Introduced: Pre 4.0

See Also

[HLS](#), [HSV](#)

COLOR_QUAN

The COLOR_QUAN function quantizes a TrueColor image and returns a pseudo-color image and palette to display the image on standard pseudo-color displays. The output image and palette can have from 2 to 256 colors.

COLOR_QUAN solves the general problem of accurately displaying decomposed, TrueColor images, that contain a palette of up to 2^{24} colors, on pseudo-color displays that can only display 256 (or fewer) simultaneous colors.

Using COLOR_QUAN

One of two color quantization methods can be used:

- Method 1 is a statistical method that attempts to find the N colors that most accurately represent the original color distribution. This algorithm uses a variation of the Median Cut Algorithm, described in “Color Image Quantization for Frame Buffer Display”, from *Computer Graphics*, Volume 16, Number 3 (July, 1982), Page 297. It repeatedly subdivides the color space into smaller and smaller rectangular boxes, until the requested number of colors are obtained.

The original colors are then mapped to the nearest output color, and the original image is resampled to the new palette with optional Floyd-Steinberg color dithering. The resulting pseudo-color image and palette are usually a good approximation of the original image.

The number of colors in the output palette defaults to the number of colors supported by the currently-selected graphics output device. The number of colors can also be specified by the COLOR keyword parameter.

- Method 2, selected by setting the keyword parameter CUBE, divides the three-dimensional color space into equal-volume cubes. Each color axis is divided into CUBE segments, resulting in $CUBE^3$ volumes. The original input image is sampled to this color space using Floyd-Steinberg dithering, which distributes the quantization error to adjacent pixels.

The CUBE method has the advantage that the color tables it produces are independent of the input image, so that multiple quantized images can be viewed simultaneously. The statistical method usually provides a better-looking result and a smaller global error.

COLOR_QUAN can use the same color mapping for a series of images. See the descriptions of the GET_TRANSLATION, MAP_ALL, and TRANSLATION keywords, below.

Syntax

Result = COLOR_QUAN(*Image_R*, *Image_G*, *Image_B*, *R*, *G*, *B*)

or

Result = COLOR_QUAN(*Image*, *Dim*, *R*, *G*, *B*)

Keywords: [, **COLORS**=*integer*{2 to 256}] [, **CUBE**={2 | 3 | 4 | 5 | 6} | ,
GET_TRANSLATION=*variable* [, /MAP_ALL]] [, /DITHER]
 [, **ERROR**=*variable*] [, **TRANSLATION**=*vector*]

The input image parameter can be passed as either three, separate color-component arrays (*Image_R*, *Image_G*, *Image_B*) or as a three-dimensional array containing all three components, *Image*, and a scalar, *Dim*, indicating the dimension over which the colors are interleaved.

Return Value

Returns a pseudo-color image composed of 2 to 256 colors.

Arguments

Image_R, Image_G, Image_B

Arrays containing the red, green, and blue components of the decomposed TrueColor image. For best results, the input image(s) should be scaled to the range of 0 to 255.

Image

A three-dimensional array containing all three components of the TrueColor image.

Dim

A scalar that indicates the method of color interleaving in the *Image* parameter. A value of 1 indicates interleaving by pixel: (3, *n*, *m*). A value of 2 indicates interleaving by row: (*n*, 3, *m*). A value of 3 indicates interleaving by image: (*n*, *m*, 3).

R, G, B

Three output byte arrays containing the red, green, and blue components of the output palette.

Keywords

COLORS

The number of colors in the output palette. This value must be at least 2 and not greater than 256. The default is the number of colors supported by the current graphics output device.

CUBE

If this keyword is set, the color space is divided into $CUBE^3$ volumes, to which the input image is quantized. This result is always Floyd-Steinberg dithered. The value of CUBE can range from 2 to 6; providing from $2^3 = 8$, to $6^3 = 216$ output colors. If this keyword is set, the COLORS, DITHER, and ERROR keywords are ignored.

DITHER

Set this keyword to dither the output image. Dithering can improve the appearance of the output image, especially when using relatively few colors.

ERROR

Set this optional keyword to a named variable. A measure of the quantization error is returned. This error is proportional to the square of the Euclidean distance, in RGB space, between corresponding colors in the original and output images.

GET_TRANSLATION

Set this keyword to a named variable in which the mapping between the original RGB triples (in the TrueColor image) and the resulting pseudo-color indices is returned as a vector. Do not use this keyword if CUBE is set.

MAP_ALL

Set this keyword to establish a mapping for all possible RGB triples into pseudo-color indices. Set this keyword only if GET_TRANSLATION is also present. Note that mapping all possible colors requires more compute time and slightly degrades the quality of the resultant color matching.

TRANSLATION

Set this keyword to a vector of translation indices obtained by a previous call to COLOR_QUAN using the GET_TRANSLATION keyword. The resulting image is quantized using this vector.

Examples

The following code segment reads a TrueColor, row interleaved, image from a disk file, and displays it on the current graphics display, using a palette of 128 colors:

```

;Open an input file:
OPENR, unit, 'XXX.DAT', /GET_LUN

;Dimensions of the input image:
a = BYTARR(512, 3, 480)

;Read the image:
READU, unit, a

;Close the file:
FREE LUN, unit

;Show the quantized image. The 2 indicates that the colors are
;interleaved by row:
TV, COLOR_QUAN(a, 2, r, g, b, COLORS=128)

;Load the new palette:
TVLCT, r, g, b

```

To quantize the image into 216 equal-volume color cubes, replace the call to `COLOR_QUAN` with the following:

```
TV, COLOR_QUAN(a, 2, r, g, b, CUBE=6)
```

Version History

Introduced: Pre 4.0

See Also

[PSEUDO](#)

COLORMAP_APPLICABLE

The COLORMAP_APPLICABLE function determines whether the current visual class supports the use of a colormap, and if so, whether colormap changes affect pre-displayed Direct Graphics or if the graphics must be redrawn to pick up colormap changes.

This routine is written in the IDL language. Its source code can be found in the file `colormap_applicable.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = COLORMAP_APPLICABLE(*redrawRequired*)

Return Value

The function returns a long value of 1 if the current visual class allows modification of the color table, and 0 otherwise.

Arguments

redrawRequired

A named variable to retrieve a value indicating whether the visual class supports automatic updating of graphics. The value is 0 if the graphics are updated automatically, or 1 if the graphics must be redrawn to pick up changes to the colormap.

Keywords

None.

Examples

To determine whether to redisplay an image after a colormap change:

```
result = COLORMAP_APPLICABLE(redrawRequired)
IF ((result GT 0) AND (redrawRequired GT 0)) THEN BEGIN
    my_redraw
ENDIF
```

Version History

Introduced: 5.2

COMFIT

The COMFIT function fits the paired data $\{x_i, y_i\}$ to one of six common types of approximating models using a gradient-expansion least-squares method.

This routine is written in the IDL language. Its source code can be found in the file `comfit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = COMFIT( X, Y, A {, /EXPONENTIAL |, /GEOMETRIC |, /GOMPERTZ |,  
/HYPERBOLIC |, /LOGISTIC |, /LOGSQUARE } [, SIGMA=variable]  
[, WEIGHTS=vector] [, YFIT=variable] )
```

Return Value

Results in a vector containing the model parameters a_0, a_1, a_2 , etc.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An n -element integer, single-, or double-precision floating-point vector.

A

A vector of initial estimates for each model parameter. The length of this vector depends upon the type of model selected.

Keywords

Note

One of the following keywords specifying a type of model must be set when using COMFIT. If you do not specify a model, IDL will display a warning message when COMFIT is called.

EXPONENTIAL

Set this keyword to compute the parameters of the exponential model.

$$y = a_0 a_1^x + a_2$$

GEOMETRIC

Set this keyword to compute the parameters of the geometric model.

$$y = a_0 x^{a_1} + a_2$$

GOMPertz

Set this keyword to compute the parameters of the Gompertz model.

$$y = a_0 a_1^{a_2 x} + a_3$$

HYPERBOLIC

Set this keyword to compute the parameters of the hyperbolic model.

$$y = \frac{1}{a_0 + a_1 x}$$

LOGISTIC

Set this keyword to compute the parameters of the logistic model.

$$y = \frac{1}{a_0 a_1^x + a_2}$$

LOGSQUARE

Set this keyword to compute the parameters of the logsquare model.

$$y = a_0 + a_1 \log(x) + a_2 \log(x)^2$$

SIGMA

Set this keyword to a named variable that will contain a vector of standard deviations for the computed model parameters.

WEIGHTS

Set this keyword equal to a vector of weights for Y_i . This vector should be the same length as X and Y . The error for each term is weighted by $WEIGHTS_i$ when computing the fit. Frequently, $WEIGHTS_i = 1.0/\sigma_i^2$, where σ is the measurement error or standard deviation of Y_i (Gaussian or instrumental weighting), or $WEIGHTS = 1/Y$ (Poisson or statistical weighting). If $WEIGHTS$ is not specified, $WEIGHTS_i$ is assumed to be 1.0.

YFIT

Set this keyword to a named variable that will contain an n -element vector of y-data corresponding to the computed model parameters.

Examples

```
; Define two n-element vectors of paired data:
X = [ 2.27, 15.01, 34.74, 36.01, 43.65, 50.02, 53.84, 58.30, $
      62.12, 64.66, 71.66, 79.94, 85.67, 114.95]
Y = [ 5.16, 22.63, 34.36, 34.92, 37.98, 40.22, 41.46, 42.81, $
      43.91, 44.62, 46.44, 48.43, 49.70, 55.31]

; Define a 3-element vector of initial estimates for the logsquare
; model:
A = [1.5, 1.5, 1.5]

; Compute the model parameters of the logsquare model, A[0], A[1],
; & A[2]:
result = COMFIT(X, Y, A, /LOGSQUARE)
```

The result should be the 3-element vector: [1.42494, 7.21900, 9.18794].

Version History

Introduced: 4.0

See Also

[CURVEFIT](#), [LADFIT](#), [LINFIT](#), [LMFIT](#), [POLY_FIT](#), [SVDFIT](#)

COMMON

The COMMON statement creates or references a common block.

Note

For more information on using COMMON, see [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual.

Syntax

COMMON *Block_Name*, *Variable₁*, ..., *Variable_n*

Version History

Introduced: Original

COMPILE_OPT

The `COMPILE_OPT` statement allows you to give the IDL compiler information that changes some of the default rules for compiling the function or procedure within which the `COMPILE_OPT` statement appears.

RSI recommends the use of

```
COMPILE_OPT IDL2
```

in all new code intended for use in a reusable library. We further recommend the use of

```
COMPILE_OPT idl2, HIDDEN
```

in all such routines that are not intended to be called directly by regular users (e.g. helper routines that are part of a larger package).

Note

For information on using `COMPILE_OPT`, see [Chapter 4, “Procedures and Functions”](#) in the *Building IDL Applications* manual.

Syntax

```
COMPILE_OPT opt1 [, opt2, ..., optn]
```

Arguments

*opt*_{*n*}

This argument can be any of the following:

- **IDL2** — A shorthand way of saying:

```
COMPILE_OPT DEFINT32, STRICTARR
```

- **DEFINT32** — IDL should assume that lexical integer constants default to the 32-bit type rather than the usual default of 16-bit integers. This takes effect from the point where the `COMPILE_OPT` statement appears in the routine being compiled and remains in effect until the end of the routine. The following table illustrates how the `DEFINT32` argument changes the interpretation of integer constants:

Constant	Normal Type	DEFINT32 Type
Without type specifier:		
42	INT	LONG
'2a'x	INT	LONG
42u	UINT	ULONG
'2a'xu	UINT	ULONG
With type specifier:		
0b	BYTE	BYTE
0s	INT	INT
0l	LONG	LONG
42.0	FLOAT	FLOAT
42d	DOUBLE	DOUBLE
42us	UINT	UINT
42ul	ULONG	ULONG
42ll	LONG64	LONG64
42ull	ULONG64	ULONG64

Table 3-3: Examples of the Effect of the DEFINT32 Argument

- **HIDDEN** — This routine should not be displayed by HELP, unless the FULL keyword to HELP is used. This directive can be used to hide helper routines that regular IDL users are not interested in seeing.

A side-effect of making a routine hidden is that IDL will not print a “Compile module” message for it when it is compiled from the library to satisfy a call to it. This makes hidden routines appear built-in to the user.

- **LOGICAL_PREDICATE** — When running this routine, from the point where the COMPILE_OPT statement appears until the end of the routine, treat

any non-zero or non-NULL predicate value as “true,” and any zero or NULL predicate value as “false.”

Background

A predicate expression is an expression that is evaluated as being “true” or “false” as part of a statement that controls program execution. IDL evaluates such expressions in the following contexts:

- `IF . . . THEN . . . ELSE` statements
- `?` : inline conditional expressions
- `WHILE . . . DO` statements
- `REPEAT . . . UNTIL` statements
- when evaluating the result from an `INIT` function method to determine if a call to `OBJ_NEW` successfully created a new object

By default, IDL uses the following rules to determine whether an expression is true or false:

- **Integer** — An integer is considered true if its least significant bit is 1, and false otherwise. Hence, odd integers are true and even integers (including zero) are false. This interpretation of integer truth values is sometimes referred to as “bitwise,” reflecting the fact that the value of the least significant bit determines the result.
- **Other** — Non-integer numeric types are true if they are non-zero, and false otherwise. String and heap variables (pointers and object references) are true if they are non-NULL, and false otherwise.

The `LOGICAL_PREDICATE` option alters the way IDL evaluates predicate expressions. When `LOGICAL_PREDICATE` is set for a routine, IDL uses the following rules to determine whether an expression is true or false:

- **Numeric Types** — A number is considered true if its value is non-zero, and false otherwise.
- **Other Types** — Strings and heap variables (pointers and object references) are considered true if they are non-NULL, or false otherwise.

Note on the NOT Operator

When using the `LOGICAL_PREDICATE` compile option, you must be aware of the fact that applying the IDL `NOT` operator to integer data computes a *bitwise* negation (1’s complement), and is generally not applicable for use in logical computations. Consider the common construction:

```

WHILE (NOT EOF(lun)) DO BEGIN
...
ENDWHILE

```

The EOF function returns 0 while the file specified by LUN has data left, and returns 1 when hits the end of file. However, the expression “NOT 1” has the numeric value -2. When the LOGICAL_PREDICATE option is not in use, the WHILE statement sees -2 as false; if the LOGICAL_PREDICATE is in use, -2 is a true value and the above loop will not terminate as desired.

The proper way to write the above loop uses the ~ logical negation operator:

```

WHILE (~ EOF(lun)) DO BEGIN
...
ENDWHILE

```

It is worth noting that this version will work properly whether or not the LOGICAL_PREDICATE compile option is in use. Logical negation operations should always use the ~ operator in preference to the NOT operator, reserving NOT exclusively for bitwise computations.

- **OBSELETE** — If the user has !WARN.OBS_ROUTINES set to True, attempts to compile a call to this routine will generate warning messages that this routine is obsolete. This directive can be used to warn people that there may be better ways to perform the desired task.
- **STRICTARR** — While compiling this routine, IDL will not allow the use of parentheses to index arrays, reserving their use only for functions. Square brackets are then the only way to index arrays. Use of this directive will prevent the addition of a new function in future versions of IDL, or new libraries of IDL code from any source, from changing the meaning of your code, and is an especially good idea for library functions.

Use of STRICTARR can eliminate many uses of the FORWARD_FUNCTION definition.

Note

STRICTARR has no effect on the use of parentheses to reference structure tags using the tag index, which is not an array indexing operation. For example, no syntax error will occur when compiling the following code:

```

COMPILE_OPT STRICTARR
mystruct = {a:0, b:1}
byindex_0 = mystruct.(0)

```

For more on referencing structure tags by index, see “[Advanced Structure Usage](#)” in Chapter 9 of the *Building IDL Applications* manual.

- **STRICTARRSUBS** — When IDL subscripts one array using another array as the source of array indices, the default behavior is to clip any out-of-range indices into range and then quietly use the resulting data without error. This behavior is described in “[Array Subscripting](#)” in Chapter 7 of the *Building IDL Applications* manual. Specifying STRICTARRSUBS will instead cause IDL to treat such out-of-range array subscripts within the body of the routine containing the COMPILE_OPT statement as an error. The position of the STRICTARRSUBS option within the module is not important: All subscripting operations within the entire body of the specified routine will be treated this way.

Version History

Introduced: 5.3.

STRICTARRSUBS option added: 5.6

LOGICAL_PREDICATE option added: 6.0

COMPLEX

The COMPLEX function returns complex scalars or arrays given one or two scalars or arrays.

Syntax

Result = COMPLEX(*Real* [, *Imaginary*] [, /DOUBLE])

or

Result = COMPLEX(*Expression*, *Offset*, *D*₁ [, ..., *D*₈] [, /DOUBLE])

Return Value

Returns complex scalars or arrays given one or two scalars or arrays. If only one parameter is supplied, the imaginary part of the result is zero, otherwise it is set to the value of the *Imaginary* parameter. Parameters are first converted to single-precision floating-point. If either or both of the parameters are arrays, the result is an array, following the same rules as standard IDL operators. If three parameters are supplied, COMPLEX extracts fields of data from *Expression*.

Arguments

Real

Scalar or array to be used as the real part of the complex result.

Imaginary

Scalar or array to be used as the imaginary part of the complex result.

Expression

The expression from which data is to be extracted.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as complex data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid floating-point value and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

DOUBLE

Set this keyword to return a double-precision complex result. Setting this keyword is equivalent to using the DCOMPLEX function, and is provided as a programming convenience.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Create a complex array from two integer arrays by entering the following commands:

```
; Create the first integer array:
A = [1,2,3]

; Create the second integer array:
B = [4,5,6]
```

```

; Make A the real parts and B the imaginary parts of the new
; complex array:
C = COMPLEX(A, B)

; See how the two arrays were combined:
PRINT, C

```

IDL prints:

```

( 1.00000, 4.00000)( 2.00000, 5.00000)
( 3.00000, 6.00000)

```

The real and imaginary parts of the complex array can be extracted as follows:

```

; Print the real part of the complex array C:
PRINT, 'Real Part: ', REAL_PART(C)

; Print the imaginary part of the complex array C:
PRINT, 'Imaginary Part: ', IMAGINARY(C)

```

IDL prints:

```

Real Part:          1.00000    2.00000    3.00000
Imaginary Part:    4.00000    5.00000    6.00000

```

Version History

Introduced: Original

See Also

[BYTE](#), [CONJ](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [IMAGINARY](#), [LONG](#), [LONG64](#), [REAL_PART](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

COMPLEXARR

The COMPLEXARR function returns a complex, single-precision, floating-point vector or array.

Syntax

Result = COMPLEXARR(*D*₁[, ..., *D*₈] [, /NOZERO])

Return Value

Returns a complex, single-precision, floating-point vector or array.

Arguments

*D*_{*i*}

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, COMPLEXARR sets every element of the result to zero. If the NOZERO keyword is set, this zeroing is not performed, and COMPLEXARR executes faster.

Examples

To create an empty, 5-element by 5-element, complex array C, enter:

```
C = COMPLEXARR(5, 5)
```

Version History

Introduced: Original

See Also

[DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#), [LON64ARR](#), [LONARR](#),
[MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

COMPLEXROUND

The COMPLEXROUND function rounds real and imaginary components of a complex array.

This routine is written in the IDL language. Its source code can be found in the file `complexround.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = COMPLEXROUND(*Input*)

Return Value

Returns the result of rounding the real and imaginary components of the input array. If the array is double-precision complex, then the result is also double-precision complex.

Arguments

Input

The complex array to be rounded.

Keywords

None.

Examples

```
X = [COMPLEX(1.245, 3.88), COMPLEX(9.1, 0.3345)]  
PRINT, COMPLEXROUND(X)
```

IDL prints:

```
( 1.00000, 4.00000)( 9.00000, 0.00000)
```

Version History

Introduced: Pre 4.0

See Also

[ROUND](#)

COMPUTE_MESH_NORMALS

The COMPUTE_MESH_NORMALS function computes normal vectors for a set of polygons described by the input array.

Syntax

```
Result = COMPUTE_MESH_NORMALS(fVerts[, iConn] )
```

Return Value

Returns a $3 \times M$ array containing a unit normal for each vertex in the input array.

Arguments

fVerts

A $3 \times M$ array of vertices.

iConn

A connectivity array (see the POLYGONS keyword to IDLgrPolygon::Init). If no *iConn* array is provided, it is assumed that the vertices in *fVerts* constitute a single polygon.

Keywords

None.

Version History

Introduced: 5.1

COND

The COND function returns the condition number of a real or complex two-dimensional array A .

By default, COND uses the L_∞ norm to compute the condition number. You may use the LNORM keyword to specify the L_1 , L_2 , or L_∞ norm.

For the L_1 and L_∞ norms, the condition number is computed from $\text{NORM}(A) \cdot \text{NORM}(\text{INVERT}(A))$. If the inverse of A is invalid (due to the singularity of A or floating-point errors in the INVERT function), COND returns -1.

For the L_2 norm, the condition number is defined as the ratio of the largest singular value to the smallest. The singular values are computed using [LA_SVD](#).

This routine is written in the IDL language. Its source code can be found in the file `cond.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = COND( A [, /DOUBLE] [, LNORM={0 | 1 | 2}])
```

Return Value

Returns the condition number of an n by n real or complex array A by explicitly computing $\text{NORM}(A) \cdot \text{NORM}(A^{-1})$. If A^{-1} is invalid (due to the singularity of A or floating-point errors in the INVERT function), COND returns -1.

Arguments

A

The two-dimensional array. For LNORM = 0 or 1, the array A must be a square and can be either real or complex. For LNORM = 2, the array A may be rectangular and can only be real.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

LNORM

Set this keyword to an integer value to indicate which norm to use for the computation. The possible values of this keyword are:

Value	Description
0	Use the L_∞ norm (the maximum absolute row sum norm). This is the default.
1	Use the L_1 norm (the maximum absolute column sum norm).
2	Use the L_2 norm (the spectral norm).

Table 3-4: LNORM Keyword Values

Examples

```

; Define a complex array A:
A = [[COMPLEX(1, 0), COMPLEX(2,-2), COMPLEX(-3, 1)], $
      [COMPLEX(1,-2), COMPLEX(2, 2), COMPLEX(1, 0)], $
      [COMPLEX(1, 1), COMPLEX(0, 1), COMPLEX(1, 5)]]

; Compute the condition number of the array using internal
; double-precision arithmetic:
PRINT, COND(A, /DOUBLE)

```

IDL prints:

```
5.93773
```

Version History

Introduced: Pre 4.0

See Also

[DETERM](#), [INVERT](#), [NORM](#), [LA_SVD](#)

CONGRID

The CONGRID function shrinks or expands the size of an array by an arbitrary amount. CONGRID is similar to REBIN in that it can resize a one, two, or three dimensional array, but where REBIN requires that the new array size must be an integer multiple of the original size, CONGRID will resize an array to any arbitrary size. (REBIN is somewhat faster, however.) REBIN averages multiple points when shrinking an array, while CONGRID just resamples the array.

This routine is written in the IDL language. Its source code can be found in the file `congrid.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CONGRID( Array, X, Y, Z [, /CENTER] [, CUBIC=value{-1 to 0}]  
                [, /INTERP] [, /MINUS_ONE] )
```

Return Value

Returns the resized array. The returned array has the same number of dimensions as the original array and is of the same data type.

Arguments

Array

A 1-, 2-, or 3-dimensional array to resize. *Array* can be any type except string or structure.

X

The new X-dimension of the resized array. *X* must be an integer or a long integer, and must be greater than or equal to 2.

Y

The new Y-dimension of the resized array. If the original array has only 1 dimension, *Y* is ignored. If the original array has 2 or 3 dimensions *Y* MUST be present.

Z

The new Z-dimension of the resized array. If the original array has only 1 or 2 dimensions, Z is ignored. If the original array has 3 dimensions then Z MUST be present.

Keywords

CENTER

Set this keyword to shift the interpolation so that points in the input and output arrays are assumed to lie at the midpoint of their coordinates rather than at their lower-left corner.

CUBIC

Set this keyword to a value between -1 and 0 to use the cubic convolution interpolation method with the specified value as the interpolation parameter. Setting this keyword equal to a value greater than zero specifies a value of -1 for the interpolation parameter. Park and Schowengerdt (see reference below) suggest that a value of -0.5 significantly improves the reconstruction properties of this algorithm. This keyword has no effect when used with 3-dimensional arrays.

Cubic convolution is an interpolation method that closely approximates the theoretically optimum sinc interpolation function using cubic polynomials. According to sampling theory, details of which are beyond the scope of this document, if the original signal, f , is a band-limited signal, with no frequency component larger than ω_0 , and f is sampled with spacing less than or equal to $1/(2\omega_0)$, then f can be reconstructed by convolving with a sinc function: $\text{sinc}(x) = \sin(\pi x) / (\pi x)$.

In the one-dimensional case, four neighboring points are used, while in the two-dimensional case 16 points are used. Note that cubic convolution interpolation is significantly slower than bilinear interpolation.

For further details see:

Rifman, S.S. and McKinnon, D.M., "Evaluation of Digital Correction Techniques for ERTS Images; Final Report", Report 20634-6003-TU-00, TRW Systems, Redondo Beach, CA, July 1974.

S. Park and R. Schowengerdt, 1983 "Image Reconstruction by Parametric Cubic Convolution", Computer Vision, Graphics & Image Processing 23, 256.

INTERP

Set this keyword to force CONGRID to use linear interpolation when resizing a 1- or 2-dimensional array. CONGRID automatically uses linear interpolation if the input array is 3-dimensional. When the input array is 1- or 2-dimensional, the default is to employ nearest-neighbor sampling.

MINUS_ONE

Set this keyword to prevent CONGRID from extrapolating one row or column beyond the bounds of the input array. For example, if the input array has the dimensions (i, j) and the output array has the dimensions (x, y) , then by default the array is resampled by a factor of (i/x) in the X direction and (j/y) in the Y direction. If MINUS_ONE is set, the array will be resampled by the factors $(i-1)/(x-1)$ and $(j-1)/(y-1)$.

Examples

Given `vol` is a 3-D array with the dimensions (80, 100, 57), resize it to be a (90, 90, 80) array

```
vol = CONGRID(vol, 90, 90, 80)
```

Version History

Introduced: Original

See Also

[REBIN](#)

CONJ

The CONJ function returns the complex conjugate of X . The complex conjugate of the real-imaginary pair (x, y) is $(x, -y)$. If X is not complex, a complex-valued copy of X is used.

Syntax

Result = CONJ(X)

Return Value

Returns the complex conjugate of X .

Arguments

X

The value for which the complex conjugate is desired. If X is an array, the result has the same structure, with each element containing the complex conjugate of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Print the conjugate of the complex pair (4.0, 5.0) by entering:

```
PRINT, CONJ(COMPLEX(4.0, 5.0))
```

IDL prints:

```
( 4.00000, -5.00000)
```

Version History

Introduced: Original

See Also

[CINDGEN](#), [COMPLEX](#), [COMPLEXARR](#), [DCINDGEN](#), [DCOMPLEX](#),
[DCOMPLEXARR](#)

CONSTRAINED_MIN

The CONSTRAINED_MIN procedure solves nonlinear optimization problems of the following form:

Minimize or maximize $gp(X)$, subject to:

$$\begin{aligned} g_{lb_i} \leq g_i(X) \leq g_{ub_i} & \quad \text{for } i = 0, \dots, n_{\text{funcs}}-1, i \neq p \\ x_{lb_j} \leq x_j \leq x_{ub_j} & \quad \text{for } j = 0, \dots, n_{\text{vars}}-1 \end{aligned}$$

X is a vector of n_{vars} variables, $x_0, \dots, x_{n_{\text{vars}}-1}$, and G is a vector of n_{funcs} functions $g_0, \dots, g_{n_{\text{funcs}}-1}$, which all depend on X . Any of these functions may be nonlinear. Any of the bounds may be infinite and any of the constraints may be absent. If there are no constraints, the problem is solved as an unconstrained optimization problem. The program solves problems of this form by the Generalized Reduced Gradient Method. See References 1-4.

CONSTRAINED_MIN uses first partial derivatives of each function g_i with respect to each variable x_j . These are automatically computed by finite difference approximation (either forward or central differences).

CONSTRAINED_MIN is based on an implementation of the GRG algorithm supplied by Windward Technologies, Inc. See Reference 11.

Syntax

```
CONSTRAINED_MIN, X, Xbnd, Gbnd, Nobj, Gcomp, Inform [, EPSTOP=value]
  [, LIMSER=value] [, /MAXIMIZE] [, NSTOP=value] [, REPORT=filename]
  [, TITLE=string]
```

Arguments

X

An n_{vars} -element vector. On input, X contains initial values for the variables. On output, X contains final values of the variable settings determined by CONSTRAINED_MIN.

Xbnd

Bounds on variables. $Xbnd$ is an $n_{\text{vars}} \times 2$ element array.

- $Xbnd[j,0]$ is the lower bound for variable $x[j]$.

- $Xbnd[j,1]$ is the upper bound for variable $x[j]$.
- Use $-1.0e30$ to denote no lower bound and $1.0e30$ for no upper bound.

Gbnd

Bounds on constraint functions. $Gbnd$ is an $nfuncs \times 2$ element array.

- $Gbnd[i,0]$ is the lower bound for function $g[i]$.
- $Gbnd[i,1]$ is the upper bound for function $g[i]$.
- use $-1.0e30$ to denote no lower bound and $1.0e30$ for no upper bound.

Bounds on the objective function are ignored; set them to 0.

Nobj

Index of the objective function.

Gcomp

A scalar string specifying the name of a user-supplied IDL function. This function must accept an $nvars$ -element vector argument x of variable values and return an $nfuncs$ -element vector G of function values.

Inform

Termination status returned from `CONSTRAINED_MIN`.

<i>Inform</i> value	Message
0	Kuhn-Tucker conditions satisfied. This is the best possible indicator that an optimal point has been found.
1	Fractional change in objective less than EPSTOP for NSTOP consecutive iterations. See Keywords below. This is not as good as $Inform=0$, but still indicates the likelihood that an optimal point has been found.
2	All remedies have failed to find a better point. User should check functions and bounds for consistency and, perhaps, try other starting values.

Table 3-5: *Inform* Argument Values

Inform value	Message
3	Number of completed 1-dimensional searches exceeded LIMSER. See Keywords below. User should check functions and bounds for consistency and, perhaps, try other starting values. It might help to increase LIMSER. Use <code>LIMSER=<i>larger_value</i></code> to do this.
4	Objective function is unbounded. CONSTRAINED_MIN has observed dramatic change in the objective function over several steps. This is a good indication that the objective function is unbounded. If this is not the case, the user should check functions and bounds for consistency.
5	Feasible point not found. CONSTRAINED_MIN was not able to find a feasible point. If the problem is believed to be feasible, the user should check functions and bounds for consistency and perhaps try other starting values.
6	Degeneracy has been encountered. The point returned may be close to optimal. The user should check functions and bounds for consistency and perhaps try other starting values.
7	Noisy and nonsmooth function values. Possible singularity or error in the function evaluations.
8	Optimization process terminated by user request.
9	Maximum number of function evaluations exceeded.
-1	Fatal Error. Some condition, such as <code>nvars < 0</code> , was encountered. CONSTRAINED_MIN documented the condition in the report and terminated. In this case, the user needs to correct the input and rerun CONSTRAINED_MIN.
-2	Fatal Error. The report file could not be opened. Check the filename specified via the REPORT keyword, and make sure you have write privileges to the specified path.

Table 3-5: Inform Argument Values

<i>Inform value</i>	Message
-3	Fatal Error. Same as <i>Inform</i> = -1. In this case, the REPORT keyword was not specified. Specify the REPORT keyword and rerun CONSTRAINED_MIN, then check the report file for more detail on the error.

Table 3-5: *Inform Argument Values*

Keywords

EPSTOP

Set this keyword to specify the CONSTRAINED_MIN convergence criteria. If the fractional change in the objective function is less than EPSTOP for NSTOP consecutive iterations, the program will accept the current point as optimal. CONSTRAINED_MIN will accept the current point as optimal if the Kuhn-Tucker optimality conditions are satisfied to EPSTOP. By default, EPSTOP = 1.0e-4.

LIMSER

If the number of completed one dimensional searches exceeds LIMSER, CONSTRAINED_MIN terminates and returns inform = 3. By default: LIMSER = 10000.

MAXIMIZE

By default, the CONSTRAINED_MIN procedure performs a minimization. Set the MAXIMIZE keyword to perform a maximization instead.

NSTOP

Set this keyword to specify the CONSTRAINED_MIN convergence criteria. If the fractional change in the objective function is less than EPSTOP for NSTOP consecutive iterations, CONSTRAINED_MIN will accept the current point as optimal. By default, NSTOP = 3.

REPORT

Set this keyword to specify a name for the CONSTRAINED_MIN report file. If the specified file does not exist, it will be created. Note that if the file cannot be created, no error message will be generated. If the specified file already exists, it will be overwritten. By default, CONSTRAINED_MIN does not create a report file.

TITLE

Set this keyword to specify a title for the problem in the CONSTRAINED_MIN report.

Examples

This example has 5 variables {X0, X1, ..., X4}, bounded above and below, a quadratic objective function {G3}, and three quadratic constraints {G0, G1, G2}, with both upper and lower bounds. See the Himmelblau text [7], problem 11.

Minimize:

$$G3 = 5.3578547X2X2 + 0.8356891X0X4 + 37.293239X0 - 40792.141$$

Subject to:

$$0 < G0 = 85.334407 + 0.0056858X1X4 + 0.0006262X0X3 - 0.0022053X2X4 < 92$$

$$90 < G1 = 80.51249 + 0.0071317X1X4 + 0.0029955X0X1 + 0.0021813X2X2 < 110$$

$$20 < G2 = 9.300961 + 0.0047026X2X4 + 0.0012547X0X2 + 0.0019085X2X3 < 25$$

and,

$$78 < X0 < 102$$

$$33 < X1 < 45$$

$$27 < X2 < 45$$

$$27 < X3 < 45$$

$$27 < X4 < 45$$

This problem is solved starting from $X = \{78, 33, 27, 27, 27\}$ which is infeasible because constraint G2 is not satisfied at this point.

The constraint functions and objective function are evaluated by HMBL11:

```
; Himmelblau Problem 11
; 5 variables and 4 functions
FUNCTION HMBL11, x

g = DBLARR(4)
g[0] = 85.334407 + 0.0056858*x[1]*x[4] + 0.0006262*x[0] $
      *x[3] - 0.0022053*x[2]*x[4]
g[1] = 80.51249 + 0.0071317*x[1]*x[4] + 0.0029955*x[0] $
      *x[1] + 0.0021813*x[2]*x[2]
g[2] = 9.300961 + 0.0047026*x[2]*x[4] + 0.0012547*x[0]* $
      x[2] + 0.0019085*x[2]*x[3]
g[3] = 5.3578547*x[2]*x[2] + 0.8356891*x[0]*x[4] $
      + 37.293239*x[0] - 40792.141
RETURN, g
END
```

```

; Example problem for CONSTRAINED_MIN
; Himmelblau Problem 11
; 5 variables and 3 constraints
; Constraints and objective defined in HMBL11
xbnd  = [[78, 33, 27, 27, 27], [102, 45, 45, 45, 45]]
gbnd  = [[0, 90, 20, 0], [92, 110, 25, 0 ]]
nobj  = 3
gcomp = 'HMBL11'
title = 'IDL: Himmelblau 11'
report = 'hmb111.txt'
x      = [78, 33, 27, 27, 27]
CONSTRAINED_MIN, x, xbnd, gbnd, nobj, gcomp, inform, $
    REPORT = report, TITLE = title
g = HMBL11(x)
; Print minimized objective function for HMBL11 problem:
PRINT, g[nobj]

```

References

1. Lasdon, L.S., Waren, A.D., Jain, A., and Ratner, M., "Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming", ACM Transactions on Mathematical Software, Vol. 4, No. 1, March 1978, pp. 34-50.
2. Lasdon, L.S. and Waren, A.D., "Generalized Reduced Gradient Software for Linearly and Nonlinearly Constrained Problems", in "Design and Implementation of Optimization Software", H. Greenberg, ed., Sijthoff and Noordhoff, pubs, 1979.
3. Abadie, J. and Carpentier, J. "Generalization of the Wolfe Reduced Gradient Method to the Case of Nonlinear Constraints", in Optimization, R. Fletcher (ed.), Academic Press London; 1969, pp. 37-47.
4. Murtagh, B.A. and Saunders, M.A. "Large-scale Linearly Constrained Optimization", Mathematical Programming, Vol. 14, No. 1, January 1978, pp. 41-72.
5. Powell, M.J.D., "Restart Procedures for the Conjugate Gradient Method," Mathematical Programming, Vol. 12, No. 2, April 1977, pp. 241-255.
6. Colville, A.R., "A Comparative Study of Nonlinear Programming Codes," I.B.M. T.R. no. 320-2949 (1968).
7. Himmelblau, D.M., Applied Nonlinear Programming, McGraw-Hill Book Co., New York, 1972.
8. Fletcher, R., "A New Approach to Variable Metric Algorithms", Computer Journal, Vol. 13, 1970, pp. 317-322.

9. Smith, S. and Lasdon, L.S., Solving Large Sparse Nonlinear Programs Using GRG, ORSA Journal on Computing, Vol. 4, No. 1, Winter 1992, pp. 1-15.
10. Luenbuerger, David G., Linear and Nonlinear Programming, Second Edition, Addison-Wesley, Reading Massachusetts, 1984.
11. Windward Technologies, GRG2 Users's Guide, 1997.

Version History

Introduced: 5.1

CONTINUE

The CONTINUE statement provides a convenient way to immediately start the next iteration of the enclosing FOR, WHILE, or REPEAT loop.

Note

Do not confuse the CONTINUE statement described here with the .CONTINUE executive command. The two constructs are not related, and serve completely different purposes.

Note

CONTINUE is not allowed within CASE or SWITCH statements. This is in contrast with the C language, which does allow this.

For more information on using CONTINUE and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

CONTINUE

Examples

This example presents one way (not necessarily the best) to print the even numbers between 1 and 10.

```
FOR I = 1,10 DO BEGIN
  ; If odd, start next iteration:
  IF (I AND 1) THEN CONTINUE
  PRINT, I
ENDFOR
```

Version History

Introduced: 5.4

CONTOUR

The CONTOUR procedure draws a contour plot from data stored in a rectangular array or from a set of unstructured points. Both line contours and filled contour plots can be created. Note that outline and fill contours cannot be drawn at the same time. To create a contour plot with both filled contours and outlines, first create the filled contour plot, then add the outline contours by calling CONTOUR a second time with the OVERPLOT keyword.

Contours can be smoothed by using the MIN_CURVE_SURF function on the contour data before contouring.

Using various keywords, described below, it is possible to specify contour levels, labeling, colors, line styles, and other options. CONTOUR draws contours by searching for each contour line and then following the line until it reaches a boundary or closes.

Smoothing Contours

The MIN_CURVE_SURF function can be used to smoothly interpolate both regularly and irregularly sampled surfaces before contouring. This function replaces the older SPLINE keyword to CONTOUR, which was inaccurate and is no longer supported. MIN_CURVE_SURF interpolates the entire surface to a relatively fine grid before drawing the contours.

Syntax

```
CONTOUR, Z [, X, Y] [, C_ANNOTATION=vector_of_strings]
  [, C_CHARSIZE=value] [, C_CHARTHICK=integer] [, C_COLORS=vector]
  [, C_LABELS=vector{each element 0 or 1}] [, C_LINestyle=vector]
  [{, /FILL |, /CELL_FILL} | [, C_ORIENTATION=degrees]
  [, C_SPACING=value]] [, C_THICK=vector] [, /CLOSED] [, /DOWNHILL]
  [, /FOLLOW] [, /IRREGULAR] [, /ISOTROPIC] [, LEVELS=vector] [,
  NLEVELS=integer{1 to 60}] [, MAX_VALUE=value] [, MIN_VALUE=value]
  [, /OVERPLOT] [{, /PATH_DATA_COORDS, PATH_FILENAME=string,
  PATH_INFO=variable, PATH_XY=variable} |, TRIANGULATION=variable]
  [, /PATH_DOUBLE] [, /XLOG] [, /YLOG] [, ZAXIS={0 | 1 | 2 | 3 | 4}]
```

Graphics Keywords: Accepts all graphics keywords accepted by PLOT except for: LINSTYLE, PSYM, SYMSIZE. See “[Graphics Keywords Accepted](#)” on page 325.

Arguments

Z

A one- or two-dimensional array containing the values that make up the contour surface. If arguments *X* and *Y* are provided, the contour is plotted as a function of the (*X*, *Y*) locations specified by their contents. Otherwise, the contour is generated as a function of the two-dimensional array index of each element of *Z*.

If the `IRREGULAR` keyword is set, *X*, *Y*, *Z* are all required, and are treated as vectors. Each point has a value of $Z[i]$ and a location of ($X[i]$, $Y[i]$).

This argument is converted to double-precision floating-point before plotting. Plots created with `CONTOUR` are limited to the range and precision of double-precision floating-point values.

X

A vector or two-dimensional array specifying the *X* coordinates for the contour surface. If *X* is a vector, each element of *X* specifies the *X* coordinate for a column of *Z* (e.g., $X[0]$ specifies the *X* coordinate for $Z[0,*]$). If *X* is a two-dimensional array, each element of *X* specifies the *X* coordinate of the corresponding point in *Z* (i.e., X_{ij} specifies the *X* coordinate for Z_{ij}).

Y

A vector or two-dimensional array specifying the *Y* coordinates for the contour surface. If *Y* a vector, each element of *Y* specifies the *Y* coordinate for a row of *Z* (e.g., $Y[0]$ specifies the *Y* coordinate for $Z[* ,0]$). If *Y* is a two-dimensional array, each element of *Y* specifies the *Y* coordinate of the corresponding point in *Z* (Y_{ij} specifies the *Y* coordinate for Z_{ij}).

Keywords

C_ANNOTATION

The label to be drawn on each contour. Usually, contours are labeled with their value. This parameter, a vector of strings, allows any text to be specified. The first label is used for the first contour drawn, and so forth. If the `LEVELS` keyword is specified, the elements of `C_ANNOTATION` correspond directly to the levels specified, otherwise, they correspond to the default levels chosen by the `CONTOUR` procedure. If there are more contour levels than elements in `C_ANNOTATION`, the remaining levels are labeled with their values.

Use of this keyword implies use of the FOLLOW keyword.

Note

This keyword has no effect if the FILL or CELL_FILL keyword is set (i.e., if the contours are drawn with solid-filled or line-filled polygons).

Example

To produce a contour plot with three levels labeled “low”, “medium”, and “high”:

```
CONTOUR, Z, LEVELS = [0.0, 0.5, 1.0], $
      C_ANNOTATION = ['low', 'medium', 'high']
```

C_CHARSIZE

The size of the characters used to annotate contour labels. Normally, contour labels are drawn at 3/4 of the size used for the axis labels (specified by the CHARSIZE keyword or !P.CHARSIZE system variable. This keyword allows the contour label size to be specified directly. Use of this keyword implies use of the FOLLOW keyword.

C_CHARTHICK

The thickness of the characters used to annotate contour labels. Set this keyword equal to an integer value specifying the line thickness of the vector drawn font characters. This keyword has no effect when used with the hardware drawn fonts. The default value is 1. Use of this keyword implies use of the FOLLOW keyword.

C_COLORS

The color index used to draw each contour. This parameter is a vector, converted to integer type if necessary. If there are more contour levels than elements in C_COLORS, the elements of the color vector are cyclically repeated.

Example

If C_COLORS contains three elements, and there are seven contour levels to be drawn, the colors $c_0, c_1, c_2, c_0, c_1, c_2, c_0$ will be used for the seven levels. To call CONTOUR and set the colors to [100,150,200], use the command:

```
CONTOUR, Z, C_COLORS = [100,150,200]
```

C_LABELS

Specifies which contour levels should be labeled. By default, every other contour level is labeled. C_LABELS allows you to override this default and explicitly specify

the levels to label. This parameter is a vector, converted to integer type if necessary. If the `LEVELS` keyword is specified, the elements of `C_LABELS` correspond directly to the levels specified, otherwise, they correspond to the default levels chosen by the `CONTOUR` procedure. Setting an element of the vector to zero causes that contour label to not be labeled. A nonzero value forces labeling.

Use of this keyword implies use of the `FOLLOW` keyword.

Example

To produce a contour plot with four levels where all but the third level is labeled:

```
CONTOUR, Z, LEVELS = [0.0, 0.25, 0.75, 1.0], $
      C_LABELS = [1, 1, 0, 1]
```

C_LINestyle

The line style used to draw each contour. As with `C_COLORS`, `C_LINestyle` is a vector of line style indices. If there are more contour levels than line styles, the line styles are cyclically repeated. See “[LINestyle](#)” on page 4337 for a list of available styles.

Note

The cell drawing contouring algorithm draws all the contours in each cell, rather than following contours. Since an entire contour is not drawn as a single operation, the appearance of the more complicated linestyles will suffer. Use of the contour following method (selected with the `FOLLOW` keyword) will give better looking results in such cases.

Example

To produce a contour plot, with the contour levels directly specified in a vector `V`, with all negative contours drawn with dotted lines, and with positive levels in solid lines:

```
CONTOUR, Z, LEVELS = V, C_LINestyle = (V LT 0.0)
```

C_ORIENTATION

If the `FILL` keyword is set, this keyword can be set to the angle, in degrees counterclockwise from the horizontal, of the lines used to fill contours. If neither `C_ORIENTATION` nor `C_SPACING` are specified, the contours are solid filled.

C_SPACING

If the FILL keyword is set, this keyword can be used to control the distance, in centimeters, between the lines used to fill the contours.

C_THICK

The line used to draw each contour level. As with C_COLORS, C_THICK is a vector of line thickness values, although the values are floating point. If there are more contours than thickness elements, elements are repeated. If omitted, the overall line thickness specified by the THICK keyword parameter or !P.THICK is used for all contours.

CELL_FILL

Set this keyword to produce a filled contour plot using a “cell filling” algorithm. Use this keyword instead of FILL when you are drawing filled contours over a map, when you have missing data, or when contours that extend off the edges of the contour plot. CELL_FILL is less efficient than FILL because it makes one or more polygons for each data cell. It also gives poor results when used with patterned (line) fills, because each cell is assigned its own pattern. Otherwise, this keyword operates identically to the FILL keyword, described below.

Tip

In order for CONTOUR to fill the contours properly when using a map projection, the X and Y arrays (if supplied) must be arranged in increasing order. This ensures that the polygons generated will be in counterclockwise order, as required by the mapping graphics pipeline.

Warning

Do not draw filled contours over the poles on Cylindrical map projections. In this case, the polar points map to lines on the map, and the interpolation becomes ambiguous, causing errors in filling. One possible work-around is to limit the latitudes to the range of -89.9 degrees to + 89.9 degrees, avoiding the poles.

CLOSED

Set this keyword to a nonzero value to close contours that intersect the plot boundaries. After a contour hits a boundary, it follows the plot boundary until it connects with its other boundary intersection. Set CLOSED=0 along with PATH_INFO and/or PATH_XY to return path information for contours that are not closed.

DOWNHILL

Set this keyword to label each contour with short, perpendicular tick marks that point in the “downhill” direction, making the direction of the grade readily apparent. If this keyword is set, the contour following method is used in drawing the contours.

FILL

Set this keyword to produce a filled contour plot. The contours are filled with solid or line-filled polygons. For solid polygons, use the `C_COLOR` keyword to specify the color index of the polygons for each contour level. For line fills, use `C_ORIENTATION`, `C_SPACING`, `C_COLOR`, `C_LINestyle`, and/or `C_THICK` to specify attributes for the lines.

If the current device is not a pen plotter, each polygon is erased to the background color before the fill lines are drawn, to avoid superimposing one pattern over another.

Contours that are not closed cannot be filled because their interior and exterior are undefined. Contours created from data sets with missing data may not be closed; many map projections can also produce contours that are not closed. You should not use filled contours in these cases.

You should not use this keyword when you are drawing filled contours over a map, when you have missing data, or when contours extend off the edges of the contour plot. In these cases, you should use `CELL_FILL` instead.

Note

If the current graphics device is the Z-buffer, the algorithm used when the `FILL` keyword is specified will not work when a Z value is also specified with the graphics keyword `ZVALUE`. In this situation, use the `CELL_FILL` keyword instead of the `FILL` keyword.

FOLLOW

In IDL version 5, `CONTOUR` always uses a line-following method. The `FOLLOW` keyword remains available for compatibility with existing code, but is no longer necessary. As in previous versions of IDL, setting `FOLLOW` will cause `CONTOUR` to draw contour labels.

IRREGULAR

Set this keyword to indicate that the input data is irregularly gridded. Setting `IRREGULAR` is the same as performing an explicit triangulation. That is:

```
CONTOUR, Z, X, Y, /IRREGULAR
```

is the same as

```
TRIANGULATE, X, Y, tri ;Get triangulation
CONTOUR, Z, X, Y, TRIANGULATION=tri
```

ISOTROPIC

Set this keyword to force the scaling of the X and Y axes to be equal.

Note

The X and Y axes will be scaled isotropically and then fit within the rectangle defined by the POSITION keyword; one of the axes may be shortened. See [“POSITION”](#) on page 4339 for more information.

LEVELS

Specifies a vector containing the contour levels drawn by the CONTOUR procedure. A contour is drawn at each level in LEVELS.

Example

To draw a contour plot with levels at 1, 100, 1000, and 10000:

```
CONTOUR, Z, LEVELS = [1, 100, 1000, 10000]
```

To draw a contour plot with levels at 50, 60, ..., 90, 100:

```
CONTOUR, Z, LEVELS = FINDGEN(6) * 10 + 50
```

MAX_VALUE

Data points with values above this value are ignored (i.e., treated as missing data) when contouring. Cells containing one or more corners with values above MAX_VALUE will have no contours drawn through them. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

MIN_VALUE

Data points with values less than this value are ignored (i.e., treated as missing data) when contouring. Cells containing one or more corners with values below MIN_VALUE will have no contours drawn through them. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

NLEVELS

The number of equally spaced contour levels that are produced by CONTOUR. If the LEVELS parameter, which explicitly specifies the value of the contour levels, is present, this keyword has no effect. If neither parameter is present, approximately six levels are drawn. NLEVELS should be a positive integer.

OVERPLOT

Set this keyword to make CONTOUR “overplot”. That is, the current graphics screen is not erased, no axes are drawn and the previously established scaling remains in effect. You must explicitly specify either the values of the contour levels or the number of levels (via the NLEVELS keyword) when using this option, unless geographic mapping coordinates are in effect.

Note

When specifying overplot levels with the NLEVELS keyword, keep in mind that the levels are calculated according to the range set by the original CONTOUR call. If the overplot dataset has a different range, you might want to set your levels more explicitly with the LEVELS keyword.

PATH_DATA_COORDS

Set this keyword to cause the output contour positions to be measured in data units rather than the default normalized units. This keyword is useful only if the PATH_XY or PATH_FILENAME keywords are set.

PATH_DOUBLE

Set this keyword to indicate that the PATH_FILENAME, PATH_INFO, and PATH_XY keywords should return vertex and contour value information as double-precision floating-point values. The default is to return this information as single-precision floating-point values.

PATH_FILENAME

Specifies the name of a file to contain the contour positions. If PATH_FILENAME is present, CONTOUR does not draw the contours, but rather, opens the specified file and writes the coordinates of the contours, into it. The file consists of a series of logical records containing binary data. Each record is preceded with a header structure defining the contour as follows:

If the PATH_DOUBLE keyword is not set:

```
{CONTOUR_HEADER, TYPE:0B, HIGH:0B, LEVEL:0, NUM:0L, VALUE:0.0}
```

If the `PATH_DOUBLE` keyword is set:

```
{CONTOUR_DBL_HEADER, TYPE:0B, HIGH:0B, LEVEL:0, NUM:0L,
VALUE:0.0D}
```

The fields are:

Field	Description
TYPE	A byte that is zero if the contour is open, and one if it is closed.
HIGH	A byte that is 1 if the contour is closed and above its surroundings, and is 0 if the contour is below. This field is meaningless if the contour is not closed.
LEVEL	A short integer with value greater or equal to zero (It is an index into the LEVELS array).
NUM	The longword number of data points in the contour.
VALUE	The contour value. If the <code>PATH_DOUBLE</code> keyword is not set, this is a single-precision floating-point value; if the <code>PATH_DOUBLE</code> keyword is set, this is a double-precision floating-point value.

Table 3-6: CONTOUR Fields

Following the header in each record are NUM X-coordinate values followed by NUM Y-coordinate values. By default, these values are specified in normalized coordinates unless the `PATH_DATA_COORDS` keyword is set.

PATH_INFO

Set this keyword to a named variable that will return path information for the contours. This information can be used, along with data stored in a variable named by the `PATH_XY` keyword, to trace closed contours. To get `PATH_INFO` and `PATH_XY` with contours that are not closed, set the `CLOSED` keyword to 0. If `PATH_INFO` is present, `CONTOUR` does not draw the contours, but rather records the path information in an array of structures of the following type:

If the `PATH_DOUBLE` keyword is not set:

```
{CONTOUR_PATH_STRUCTURE, TYPE:0B, HIGH_LOW:0B, $
LEVEL:0, N:0L, OFFSET:0L, VALUE:0.0}
```

If the `PATH_DOUBLE` keyword is set:

```
{COUNTOUR_DBL_PATH_STRUCTURE, TYPE:0B, HIGH_LOW:0B, LEVEL:0,
  N: 0L, OFFSET:0L, VALUE:0.0D}
```

The fields are:

Field	Description
TYPE	A byte that is zero if the contour is open, and one if it is closed. Note - If the <code>CLOSE</code> keyword is not explicitly set equal to zero, all contours will be closed.
HIGH_LOW	A byte that is 1 if the contour is above its surroundings, and is 0 if the contour is below.
LEVEL	A short integer indicating the index of the contour level, from zero to the number of levels minus one.
N	A long integer indicating the number of XY pairs in the contour's path.
OFFSET	A long integer that is the offset into the array defined by <code>PATH_XY</code> , representing the first XY coordinate for this contour.
VALUE	The contour value. If the <code>PATH_DOUBLE</code> keyword is not set, this is a single-precision floating-point value; if the <code>PATH_DOUBLE</code> keyword is set, this is a double-precision floating-point value.

Table 3-7: PATH_INFO Fields

See the examples section below for an example using the `PATH_INFO` and `PATH_XY` keywords to return contour path information.

PATH_XY

Set this keyword to a named variable that returns the coordinates of a set of closed polygons defining the closed paths of the contours. This information can be used, along with data stored in a variable named by the `PATH_INFO` keyword, to trace closed contours. To get `PATH_XY` and `PATH_INFO` with contours that are not closed, set the `CLOSED` keyword to 0. If `PATH_XY` is present, `CONTOUR` does not draw the contours, but rather records the path coordinates in the named array. If the

PATH_DOUBLE keyword is not set, the array will contain single-precision floating point values; if the PATH_DOUBLE keyword is set, the array will contain double-precision floating point values. By default, the values in the array are specified in normalized coordinates unless the PATH_DATA_COORDS keyword is set.

See the examples section below for an example using the PATH_INFO and PATH_XY keywords to return contour path information.

TRIANGULATION

Set this keyword to a variable that contains an array of triangles returned from the TRIANGULATE procedure. Providing triangulation data allows you to contour irregularly gridded data directly, without gridding.

XLOG

Set this keyword to specify a logarithmic X axis.

YLOG

Set this keyword to specify a logarithmic Y axis.

ZAXIS

Set this keyword to an integer value to draw a Z axis for the CONTOUR plot. CONTOUR draws no Z axis by default. This keyword is of use only if a three-dimensional transformation is established. Possible values are:

- 1 - Draws Z axis from the lower right-hand corner of the plot
- 2 - Draws Z axis from the lower left-hand corner of the plot
- 3 - Draws Z axis from the upper left-hand corner of the plot
- 4 - Draws Z axis from the upper right-hand corner of the plot

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above.

BACKGROUND, CHARSIZE, CHARTHICK, CLIP, COLOR, DATA, DEVICE, FONT, NOCLIP, NODATA, NOERASE, NORMAL, POSITION, SUBTITLE, T3D, THICK, TICKLEN, TITLE, [XYZ]CHARSIZE, [XYZ]GRIDSTYLE, [XYZ]MARGIN, [XYZ]MINOR, [XYZ]RANGE, [XYZ]STYLE, [XYZ]THICK, [XYZ]TICKFORMAT, [XYZ]TICKINTERVAL, [XYZ]TICKLAYOUT,

[XYZ]TICKLEN, [XYZ]TICKNAME, [XYZ]TICKS, [XYZ]TICKUNITS,
 [XYZ]TICKV, [XYZ]TICK_GET, [XYZ]TITLE, ZVALUE.

Examples

Example 1

This example creates a contour plot with 10 contour levels where every other contour is labeled:

```
;Create a simple dataset to plot:
Z = DIST(100)

;Draw the plot:
CONTOUR, Z, NLEVELS=10, /FOLLOW, TITLE='Simple Contour Plot'
```

Example 2

This example shows the use of polygon filling and smoothing.

```
;Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Create a surface to contour (2D array of random numbers):
A = RANDOMU(seed, 5, 6)

;Smooth the dataset before contouring:
B = MIN_CURVE_SURF(A)

;Load discrete colors for contours:
TEK_COLOR

;Draw filled contours:
CONTOUR, B, /FILL, NLEVELS=5, C_COLOR=INDGEN(5)+2

;Overplot the contour lines with tickmarks:
CONTOUR, B, NLEVELS=5, /DOWNHILL, /OVERPLOT
```

Alternatively, we could draw line-filled contours by replacing the last two commands with:

```
CONTOUR, B, C_ORIENTATION=[0, 22, 45]

CONTOUR, B, /OVERPLOT, NLEVELS=5
```

Example 3

The following example saves the closed path information of a set of contours and plots the result:

```

; Create a 2D array of random numbers:
A = RANDOMU(seed, 8, 10)

; Smooth the dataset before contouring:
B = MIN_CURVE_SURF(A)

; Compute contour paths:
CONTOUR, B, PATH_XY=xy, PATH_INFO=info
FOR I = 0, (N_ELEMENTS(info) - 1 ) DO BEGIN
    S = [INDGEN(info(I).N), 0]

; Plot the closed paths:
    PLOTS, xy(*,INFO(I).OFFSET + S ), /NORM
ENDFOR

```

Example 4

This example contours irregularly-gridded data without having to call TRIGRID. First, use the TRIANGULATE procedure to get the Delaunay triangulation of your data, then pass the triangulation array to CONTOUR:

```

;Make 50 normal X, Y points:
x = RANDOMN(seed, 50)
y = RANDOMN(seed, 50)

;Make the Gaussian:
Z = EXP(-(x^2 + y^2))

;Get triangulation:
TRIANGULATE, X, Y, tri

;Draw the contours:
CONTOUR, Z, X, Y, TRIANGULATION = tri

```

Version History

Introduced: Original

See Also

[ICONTOUR](#), [IMAGE_CONT](#), [SHADE_SURF](#), [SHOW3](#), [SURFACE](#)

CONVERT_COORD

The `CONVERT_COORD` function transforms one or more sets of coordinates to and from the coordinate systems supported by IDL.

The input coordinates X and, optionally, Y and/or Z can be given in data, device, or normalized form by using the `DATA`, `DEVICE`, or `NORMAL` keywords. The default input coordinate system is `DATA`. The keywords `TO_DATA`, `TO_DEVICE`, and `TO_NORMAL` specify the output coordinate system.

If the input points are in 3D data coordinates, be sure to set the `T3D` keyword.

Note

`CONVERT_COORD` utilizes values currently stored in the `!X`, `!Y`, `!Z` and `!P` system variables to compute coordinate conversion factors.

Warning

For devices that support windows, `CONVERT_COORD` can only provide valid results if a window is open and current. Also, `CONVERT_COORD` only applies to Direct Graphics devices.

Syntax

```
Result = CONVERT_COORD( X [, Y [, Z]] [, /DATA | , /DEVICE | , /NORMAL]
    [, /DOUBLE][, /T3D] [, /TO_DATA | , /TO_DEVICE | , /TO_NORMAL] )
```

Return Value

The result of the function is a $(3, n)$ vector containing the (x, y, z) components of the n output coordinates.

Arguments

X

A vector or scalar argument providing the X components of the input coordinates. If only one argument is specified, X must be an array of either two or three vectors (i.e., $(2, *)$ or $(3, *)$). In this special case, $X[0, *]$ are taken as the X values, $X[1, *]$ are taken as the Y values, and, if present, $X[2, *]$ are taken as the Z values.

Y

An optional argument providing the Y input coordinate(s).

Z

An optional argument providing the Z input coordinate(s).

Keywords**DATA**

Set this keyword if the input coordinates are in data space (the default).

DEVICE

Set this keyword if the input coordinates are in device space.

DOUBLE

Set this keyword to indicate that the returned coordinates should be double-precision. If this keyword is not set, the default is to return single-precision coordinates (unless double-precision arguments are input, in which case the returned coordinates will be double-precision).

NORMAL

Set this keyword if the input coordinates are in normalized space.

T3D

Set this keyword if the 3D transformation !P.T is to be applied.

TO_DATA

Set this keyword if the output coordinates are to be in data space.

TO_DEVICE

Set this keyword if the output coordinates are to be in device space.

TO_NORMAL

Set this keyword if the output coordinates are to be in normalized space.

Examples

Convert, using the currently established viewing transformation, 11 points along the parametric line $x = t$, $y = 2t$, $z = t^2$, along the interval $[0, 1]$ from data coordinates to device coordinates:

```
; Establish a valid transformation matrix:
SURFACE, DIST(20), /SAVE

; Make a vector of X values:
X = FINDGEN(11)/10.

; Convert the coordinates. D will be a (3,11) element array:
D = CONVERT_COORD(X, 2*X, X^2, /T3D, /TO_DEVICE)
```

Version History

Introduced: Pre 4.0

See Also

[CV_COORD](#)

CONVOL

The CONVOL function convolves an array with a kernel, and returns the result. Convolution is a general process that can be used for various types of smoothing, signal processing, shifting, differentiation, edge detection, etc. The CENTER keyword controls the alignment of the kernel with the array and the ordering of the kernel elements. If CENTER is explicitly set to 0, convolution is performed in the strict mathematical sense, otherwise the kernel is centered over each data point.

Using CONVOL

Assume $R = \text{CONVOL}(A, K, S)$, where A is an n -element vector, K is an m -element vector ($m < n$), and S is the scale factor. If the CENTER keyword is omitted or set to 1:

$$R_t = \begin{cases} \frac{1}{S} \sum_{i=0}^{m-1} A_{t+i-m/2} K_i & \text{if } m/2 \leq t < n - m/2 \\ 0 & \text{otherwise} \end{cases}$$

where the value $m/2$ is determined by *integer division*. This means that the result of the division is the largest *integer* value less than or equal to the fractional number.

If CENTER is explicitly set to 0:

$$R_t = \begin{cases} \frac{1}{S} \sum_{i=0}^{m-1} A_{t-i} K_i & \text{if } t \geq m - 1 \\ 0 & \text{otherwise} \end{cases}$$

In the two-dimensional, zero CENTER case where A is an m by n -element array, and K is the l by l element kernel; the result R is an m by n -element array:

$$R_{t,u} = \begin{cases} \frac{1}{S} \sum_{i=0}^{l-1} \sum_{j=0}^{l-1} A_{t-i, u-j} K_{i,j} & \text{if } t \geq l - 1 \quad \text{and} \quad u \geq l - 1 \\ 0 & \text{otherwise} \end{cases}$$

The centered case is similar, except the $t-i$ and $u-j$ subscripts are replaced by $t+i-l/2$ and $u+j-l/2$.

Syntax

```
Result = CONVOL( Array, Kernel [, Scale_Factor] [, /CENTER] [, /EDGE_WRAP]
  [, /EDGE_TRUNCATE] [, MISSING=value] [, /NAN] )
```

Return Value

Returns the result of the array convolution.

Arguments

Array

An array of any basic type except string. The result of CONVOL has the same type and dimensions as *Array*.

If the *Array* parameter is of byte type, the result is clipped to the range of 0 to 255. Negative results are set to 0, and values greater than 255 are set to 255.

Kernel

An array of any type except string. If the type of *Kernel* is not the same as *Array*, a copy of *Kernel* is made and converted to the appropriate type before use. The size of the kernel dimensions must be smaller than those of *Array*.

Note

CONVOL accepts non-square kernels including one-dimensional kernels.

Scale_Factor

A scale factor that is divided into each resulting value. This argument allows the use of fractional kernel values and avoids overflow with byte or integer arguments. If omitted, a scale factor of 1 is used.

Keywords

CENTER

Set or omit this keyword to center the kernel over each array point. If CENTER is explicitly set to zero, the CONVOL function works in the conventional mathematical

sense. In many signal and image processing applications, it is useful to center a symmetric kernel over the data, thereby aligning the result with the original array.

Note that for the kernel to be centered, it must be symmetric about the point $K(\text{FLOOR}(m/2))$, where m is the number of elements in the kernel.

EDGE_WRAP

Set this keyword to make CONVOL compute the values of elements at the edge of *Array* by “wrapping” the subscripts of *Array* at the edge. For example, if CENTER is set to zero:

$$R_t = \left\{ \frac{1}{S} \left[\sum_{i=0}^{m-1} A_{((t-i) \bmod(n))} K_i \right] \right.$$

where n is the number of elements in *Array*. The mod operator in the formula above is defined as $a \bmod b = a - b * \text{floor}(a/b)$. For example, $-1 \bmod 5$ is 4. If neither EDGE_WRAP nor EDGE_TRUNCATE is set, CONVOL sets the values of elements at the edges of *Array* to zero.

EDGE_TRUNCATE

Set this keyword to make CONVOL compute the values of elements at the edge of *Array* by repeating the subscripts of *Array* at the edge. For example, if CENTER is set to zero:

$$R_t = \left\{ \frac{1}{S} \sum_{i=0}^m A_{((t-i) > 0 < (n-1))} K_i \right.$$

where n is the number of elements in *Array*. The “<” and “>” operators in the above formula return the smaller and larger of their operands, respectively. If neither EDGE_WRAP nor EDGE_TRUNCATE is set, CONVOL sets the values of elements at the edges of *Array* to zero.

MISSING

Set this keyword to the numeric value to return for elements that contain no valid points within the kernel. The default is the IEEE floating-point value NaN. This keyword is only used if the NAN keyword is set.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.) Missing data are ignored when computing the convolution for neighboring elements. In the *Result*, missing elements are replaced by the convolution of all other valid points within the kernel. If all points within the kernel are missing, then the result at that point is given by the MISSING keyword.

Note

CONVOL should never be called without the NAN keyword if the input array may possibly contain *NaN* or *Infinity* values.

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Convolve a vector of random noise and a one-dimensional triangular kernel and plot the result. Create a simple vector as the original dataset and plot it by entering:

```
A = RANDOMN(SEED, 100) & PLOT, A
```

Create a simple kernel by entering:

```
K = [1, 2, 3, 2, 1]
```

Convolve the two and overplot the result by entering:

```
OPlot, CONVOL(A, K, TOTAL(K))
```

Version History

Introduced: Original

See Also

[BLK_CON](#)

COORD2TO3

The COORD2TO3 function converts normalized X and Y screen coordinates to 3D data coordinates.

Note

A valid 3D transform must exist in !P.T or be specified by the PTI keyword. The axis scaling variables, !X.S, !Y.S and !Z.S must be valid.

This routine is written in the IDL language. Its source code can be found in the file `coord2to3.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$Result = COORD2TO3(Mx, My, Dim, D0 [, PTI])$$

Return Value

Returns a three-element vector containing 3D data coordinates given the normalized X and Y screen coordinates and one of the three data coordinates.

Arguments

Mx, My

The normalized X and Y screen coordinates.

Dim

A parameter used to specify which data coordinate is fixed. Use 0 for a fixed X data coordinate, 1 for a fixed Y data coordinate, or 2 for a fixed Z data coordinate.

D0

The value of the fixed data coordinate.

PTI

The inverse of !P.T. If this parameter is not supplied, or set to 0, COORD2TO3 computes the inverse. If this routine is to be used in a loop, the caller should supply PTI for highest efficiency.

Keywords

None.

Examples

To return the data coordinates of the mouse, fixing the data Z value at 10, enter the commands:

```
;Make sure a transformation matrix exists.  
CREATE_VIEW  
  
;Get the normalized mouse coords.  
CURSOR, X, Y, /NORM  
  
;Print the 3D coordinates.  
PRINT, COORD2TO3(X, Y, 2, 10.0)
```

See Also

[CONVERT_COORD](#), [CREATE_VIEW](#), [CV_COORD](#), [SCALE3](#), [T3D](#)

COPY_LUN

The COPY_LUN procedure copies data between two open files. It allows you to transfer a known amount of data from one file to another without needing to have the data available in an IDL variable. COPY_LUN can copy a fixed amount of data, specified in bytes or lines of text, or it can copy from the current position of the file pointer in the input file to the end of that file.

COPY_LUN copies data between open files. To copy entire files based on their names, see the [FILE_COPY](#) procedure. To read and discard a known amount of data from a file, see the [SKIP_LUN](#).

Syntax

```
COPY_LUN, FromUnit, ToUnit [, Num] [, /EOF] [, /LINES]  
      [, /TRANSFER_COUNT]
```

Arguments

FromUnit

An integer that specifies the file unit for the file from which data is to be taken (the *source* file). Data is copied from *FromUnit*, starting at the current position of the file pointer. The file pointer is advanced as data is read. The file specified by *FromUnit* must be open, and must not have been opened with the RAWIO keyword to OPEN.

ToUnit

An integer that specifies the file unit for the file to which data is to be written (the *destination* file). Data is written to *ToUnit*, starting at the current position of the file pointer. The file pointer is advanced as data is written. The file specified by *ToUnit* must be open for output (OPENW or OPENU), and must not have been opened with the RAWIO keyword to OPEN.

Num

The amount of data to transfer between the two files. This value is specified in bytes, unless the LINES keyword is specified, in which case it is taken to be the number of text lines. If *Num* is not specified, COPY_LUN acts as if the EOF keyword has been set, and copies all data in *FromUnit* (the source file) from the current position of the file pointer to the end of the file.

If *Num* is specified and the source file comes to end of file before the specified amount of data is transferred, COPY_LUN issues an end-of-file error. The EOF keyword alters this behavior.

Keywords

EOF

Set this keyword to ignore the value given by *Num* (if present) and instead transfer all data between the current position of the file pointer in *FromUnit* and the end of the file.

Note

If EOF is set, no end-of-file error is issued even if the amount of data transferred does not match the amount specified by *Num*. The TRANSFER_COUNT keyword can be used with EOF to determine how much data was transferred.

LINES

Set this keyword to indicate that the *Num* argument specifies the number of lines of text to be transferred. By default, the *Num* argument specifies the number of bytes of data to transfer.

TRANSFER_COUNT

Set this keyword equal to a named variable that will contain the amount of data transferred. If LINES is specified, this value is the number of lines of text. Otherwise, it is the number of bytes. TRANSFER_COUNT is primarily useful when the *Num* argument is not specified or the EOF keyword is present. If *Num* is specified and the EOF keyword is not present, TRANSFER_COUNT will be the same as the value specified for *Num*.

Examples

Copy the next 100000 bytes of data between two files:

```
COPY_LUN, FromUnit, ToUnit, 100000
```

Copy the next 8 lines of text between two files:

```
COPY_LUN, FromUnit, ToUnit, 8, /LINES
```

Copy the remainder of the data in one file to another, and use the TRANSFER_COUNT keyword to determine how much data was copied:

```
COPY_LUN, FromUnit, ToUnit, /EOF, TRANSFER_COUNT=n
```

Copy the remaining lines of text from one file to another, and use the `TRANSFER_COUNT` keyword to determine how many lines were transferred.

```
COPY_LUN, FromUnit, ToUnit, /EOF, /LINES, TRANSFER_COUNT=n
```

Version History

Introduced: 5.6

See Also

[CLOSE](#), [EOF](#), [FILE_COPY](#), [FILE_LINK](#), [FILE_MOVE](#), [OPEN](#), [READ/READF](#), [SKIP_LUN](#), [WRITEU](#)

CORRELATE

The CORRELATE function computes the linear Pearson correlation coefficient of two vectors or the correlation matrix of an $m \times n$ array. Alternatively, this function computes the covariance of two vectors or the covariance matrix of an $m \times n$ array.

This routine is written in the IDL language. Its source code can be found in the file `correlate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CORRELATE( X [, Y] [, /COVARIANCE] [, /DOUBLE] )
```

Return Value

If vectors of unequal lengths are specified, the longer vector is truncated to the length of the shorter vector and a single correlation coefficient is returned. If an $m \times n$ array is specified, the result will be an $m \times m$ array of linear Pearson correlation coefficients, with the element i,j corresponding to correlation of the i th and j th columns of the input array.

Arguments

X

A vector or an $m \times n$ array. X can be integer, single-, or double-precision floating-point.

Y

An integer, single-, or double-precision floating-point vector. If X is an $m \times n$ array, Y should not be supplied.

Keywords

COVARIANCE

Set this keyword to compute the sample covariance rather than the correlation coefficient.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

Define the data vectors.

```
X = [65, 63, 67, 64, 68, 62, 70, 66, 68, 67, 69, 71]
Y = [68, 66, 68, 65, 69, 66, 68, 65, 71, 67, 68, 70]
```

Compute the linear Pearson correlation coefficient of x and y. The result should be 0.702652:

```
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
0.702652
```

Compute the covariance of x and y. The result should be 3.66667.

```
PRINT, CORRELATE(X, Y, /COVARIANCE)
```

IDL prints:

```
3.66667
```

Define an array with x and y as its columns.

```
A = TRANSPOSE([[X],[Y]])
```

Compute the correlation matrix.

```
PRINT, CORRELATE(A)
```

IDL prints:

```
1.00000    0.702652
0.702652    1.00000
```

Version History

Introduced: Pre 4.0

See Also

[A_CORRELATE](#), [C_CORRELATE](#), [M_CORRELATE](#), [P_CORRELATE](#), [R_CORRELATE](#)

COS

The periodic function COS returns the trigonometric cosine of X.

Syntax

Result = COS(*X*)

Return Value

Returns the trigonometric cosine of X.

Arguments

X

The angle for which the cosine is desired, specified in radians. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. When applied to complex numbers:

$$\text{COS}(x) = (\text{EXP}(i*x) + 1/\text{EXP}(i*x))/2$$

where *i* is defined as COMPLEX(0, 1).

If X is an array, the result has the same structure, with each element containing the cosine of the corresponding element of X.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Find the cosine of 0.5 radians and print the result by entering:

```
PRINT, COS(.5)
```

IDL prints:

```
0.877583
```

Version History

Introduced: Original

See Also

[ACOS](#), [COSH](#)

COSH

The COSH function returns the hyperbolic cosine of X .

Syntax

Result = COSH(X)

Return Value

Returns the hyperbolic cosine of X .

Arguments

X

The value for which the hyperbolic cosine is desired, specified in radians. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. COSH is defined as:

$$\text{COSH}(u) = (e^u + e^{-u}) / 2$$

If X is an array, the result has the same structure, with each element containing the hyperbolic cosine of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Find the hyperbolic cosine of 0.5 radians and print the result by entering:

```
PRINT, COSH(.5)
```

IDL prints:

1.12763

Version History

Introduced: Original

See Also

[ACOS](#), [COS](#)

CPU

The CPU procedure is used to change the values stored in the read-only !CPU system variable, which in turn controls the way IDL uses the system processor or processors.

Note

Not all routines are affected by changes to the !CPU system variable. Those routines that *are* affected can override some of the values in the !CPU system variable by setting *thread pool keywords*, which change the way IDL uses the system processor(s) during a single invocation of the routine. A list of thread pool keywords appears at the end of the keywords list for each routine that is affected by the state of the !CPU system variable.

Syntax

```
CPU [, /RESET] [, RESTORE = structure] [, TPOOL_MAX_ELTS = NumMaxElts]  
    [, TPOOL_MIN_ELTS = NumMinElts] [, TPOOL_NTHREADS = NumThreads]  
    [, /VECTOR_ENABLE]
```

Arguments

None.

Keywords

RESET

Set this keyword to cause the CPU procedure to reset the values contained in the !CPU system variable to the values it had when IDL was started. See [“Restoring !CPU Values”](#) on page 349 for more information on this topic. The RESET and RESTORE keywords are mutually exclusive; only one of them can be specified in a given call to CPU.

RESTORE

Set this keyword equal to a structure of type !CPU to set the !CPU system variable with the values contained in the structure. See [“Restoring !CPU Values”](#) on page 349 for more information on this topic. The RESET and RESTORE keywords are mutually exclusive; only one of them can be specified in a given call to CPU.

TPOOL_MAX_ELTS

This keyword changes the value returned by `!CPU.TPOOL_MAX_ELTS`.

Set this keyword to a non-zero value to set the maximum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation exceeds the number contained in `!CPU.TPOOL_MAX_ELTS`, IDL will not use the thread pool for the computation. Setting this value to 0 removes any limit on maximum number of elements, and any computation with at least `!CPU.TPOOL_MIN_ELTS` will use the thread pool.

See “[Possible Drawbacks to the Use of the IDL Thread Pool](#)” in Chapter 15 of the *Building IDL Applications* manual for discussion of the circumstances under which it may be useful to specify a maximum number of elements for computations that use the thread pool.

TPOOL_MIN_ELTS

This keyword changes the value returned by `!CPU.TPOOL_MIN_ELTS`.

Set this keyword to a non-zero value to set the minimum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation is less than the number contained in `!CPU.TPOOL_MIN_ELTS`, IDL will not use the thread pool for the computation. Use this keyword to prevent IDL from using the thread pool on tasks that are too small to benefit from it.

See “[Possible Drawbacks to the Use of the IDL Thread Pool](#)” in Chapter 15 of the *Building IDL Applications* manual for discussion of the circumstances under which it may be useful to specify a minimum number of elements for computations that use the thread pool.

TPOOL_NTHREADS

This keyword changes the value returned by `!CPU.TPOOL_NTHREADS`.

Set this keyword to the number of threads IDL should use when performing computations that take advantage of the thread pool. By default, IDL will use `!CPU.HW_NCPU` threads, so that each thread will have the potential to run in parallel with the others. Set this keyword equal to 0 (zero) to ensure that `!CPU.HW_NCPU` threads will be used. Set this keyword equal to 1 (one) to disable use of the thread pool.

Note

For numerical computation, there is no benefit to using more threads than your system has CPUs. However, depending on the size of the problem and the number of other programs running on the system, there may be a performance advantage to

using fewer CPUs. See [“Possible Drawbacks to the Use of the IDL Thread Pool”](#) in Chapter 15 of the *Building IDL Applications* manual for a discussion of the circumstances under which using fewer than the maximum number of CPUs makes sense.

If the `IDL_CPU_TPOOL_NTHREADS` environment variable has been set, then `!CPU.TPOOL_NTHREADS` is initialized with this value at startup. On systems shared by multiple users, you may wish to set this environment variable so that IDL uses the specified number of threads instead of defaulting to the number of CPUs present in the underlying hardware. For more information, see [“IDL_CPU_TPOOL_NTHREADS”](#) in Chapter 1 of the *Using IDL* manual.

VECTOR_ENABLE

This keyword changes the value returned by `!CPU.VECTOR_ENABLE`.

Set this keyword to enable use of the system’s vector unit (e.g. Macintosh Altivec/Velocity Engine) if one is present. Set this keyword equal to 0 (zero) explicitly disable such use. This keyword is ignored if the current system does not support a vector unit (that is, if `!CPU.HW_VECTOR = 0`).

Restoring !CPU Values

It is sometimes necessary to alter the way IDL uses the system processor or processors for a section of code, and then to restore IDL to the settings in effect before the code section was executed. It is often possible to accomplish this using the thread pool keywords to specific routines; in some circumstances, however, this approach may be inconvenient or ineffective. For example:

- If the section of code includes many routines for which alterations are required, the repetition of the keywords is inconvenient and error prone.
- Binary and unary operators that use the thread pool do not accept keywords.
- There is no keyword equivalent to the `!CPU.VECTOR_ENABLE` field.
- Future versions of IDL may provide additional CPU options; adding keywords to your code to invoke these options could pose maintenance problems.
- You may be working with IDL routines you did not write.

In all of these cases, the answer is to use the CPU procedure to alter the value of the `!CPU` system variable to suit the needs of your code, execute the code, and then reset `!CPU` to the state it had initially. It is possible to do this by using the individual keywords to CPU, as shown below:

```

save_cpu = !CPU           ; Save the current configuration
CPU, TPOOL_NTHREADS=2, ... ; Change the current configuration
...                       ; Carry out a computation
; Restore the CPU state
CPU, TPOOL_MAX_ELTS=save_cpu.tpool_max_elts, $
    TPOOL_MIN_ELTS=save_cpu.tpool_min_elts, $
    TPOOL_NTHREADS=save_cpu.tpool_nthreads

```

There are two obvious problems with this approach: it is tedious and error prone, and if new versions of IDL add new CPU attributes, your code will require updates in order to reset them.

The `RESTORE` keyword to `CPU` solves both of these problems. When a structure of type `!CPU` is supplied using the `RESTORE` keyword, IDL adjusts the `!CPU` system variable to match the structure. Using `RESTORE`, the above example becomes:

```

save_cpu = !CPU
CPU, TPOOL_NTHREADS=2, ...
...
CPU, RESTORE=save_cpu

```

The `RESET` keyword is similar to `RESTORE`, but it resets the `!CPU` system variable to the values it had when IDL first started. The syntax is:

```

CPU, /RESET

```

The `RESET` keyword is useful in *ad hoc* analysis, when you've made changes to the `!CPU` system variable manually (perhaps as an experiment), and you wish to bring `!CPU` back to its default state for your machine without exiting and restarting IDL.

`RESET` and `RESTORE` are mutually exclusive — only one of them can be used in a given call to the `CPU` procedure. However, they are allowed in conjunction with the other keywords to `CPU`. In such cases, the explicit keywords take precedence over the values from the `RESET` or `RESTORE` keywords. For example, the following statement changes the number of threads used by the IDL thread pool to 3, and sets all other CPU parameters to the values contained in a variable named `save_cpu`:

```

CPU, TPOOL_NTHREADS=3, RESTORE=save_cpu

```

Example

Configure `!CPU` so that by default, IDL will use two threads for computations that involve more than 5000 data values.

```

CPU, TPOOL_MIN_ELTS=5000, TPOOL_NTHREADS=2

```

Version History

Introduced: 5.5

RESET and RESTORE keywords: 6.1

See Also

[!CPU](#), [Chapter 15](#), “[Multithreading in IDL](#)” in the *Building IDL Applications* manual.

CRAMER

The CRAMER function solves an n by n linear system of equations using Cramer's rule.

This routine is written in the IDL language. Its source code can be found in the file `cramer.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CRAMER(*A*, *B* [, /DOUBLE] [, ZERO=*value*])

Return Value

Returns the solution of an n by n linear system of equations using Cramer's rule.

Arguments

A

An n by n single- or double-precision floating-point array.

B

An n -element single- or double-precision floating-point vector.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ZERO

Use this keyword to set the value of the floating-point zero. A floating-point zero on the main diagonal of a triangular array results in a zero determinant. A zero determinant results in a "Singular matrix" error and stops the execution of CRAMER. For single-precision inputs, the default value is 1.0×10^{-6} . For double-precision inputs, the default value is 1.0×10^{-12} .

Examples

Define an array A and right-hand side vector B.

```
A = [[ 2.0,  1.0,  1.0], $  
      [ 4.0, -6.0,  0.0], $  
      [-2.0,  7.0,  2.0]]  
B = [3.0, 10.0, -5.0]
```

```
;Compute the solution and print.  
PRINT, CRAMER(A,B)
```

IDL prints:

```
1.00000      -1.00000      2.00000
```

Version History

Introduced: Pre 4.0

See Also

[CHOLSOL](#), [GS_ITER](#), [LU_COMPLEX](#), [LUSOL](#), [SVSOL](#), [TRISOL](#)

CREATE_CURSOR

The `CREATE_CURSOR` function creates an image array from a string array that represents a 16 by 16 cursor. The returned image array can be passed to the `REGISTER_CURSOR` procedure *Image* argument. This allows you to quickly design a cursor using a simple string array.

Syntax

```
Result = CREATE_CURSOR( StringArray [, HOTSPOT=variable]  
                        [, MASK=variable])
```

Return Value

Translates the input string array into an image that satisfies the *Image* argument of the `REGISTER_CURSOR` procedure.

Arguments

StringArray

This must be a 16 element string array of 16 characters each that represents a window cursor. The array can consist of the "#" character, space and non-space characters, and optionally, a "\$" character as follows:

- The "#" characters define the black outline of the cursor.
- Any non-space characters define the shape of the white mask. Adding a mask ensures the cursor's visibility in light and dark areas of a visualization. Use the `MASK` keyword to return the array of masked out values.
- The "\$" defines a hot spot. Use the `HOTSPOT` keyword to return the location of this character within the string array.

Keywords

HOTSPOT

Set this keyword to a variable that will contain a two-element vector indicating the location of the "\$" character in the *StringArray*. This is considered to be the mouse position of the cursor. If there is no "\$" present, then [0,0] is returned. This variable can be directly passed to the `HOTSPOT` keyword of `REGISTER_CURSOR`.

MASK

Set this keyword to return a 16 by 16 byte array containing any non-space characters, which indicates the area to be masked out. This variable can be passed to the MASK keyword of [REGISTER_CURSOR](#).

Examples

The following code creates a simple cursor. Define a string array representing the body and mask portions of the cursor. Notice the hotspot ("\$\$") in the cursor structure.

```
strArray = [ $
    |      .      |      $
    |     .#     |     $
    |    .###    |    $
    |   .####   |   $
    |  . . . # . . . |  $
    | .# . # . # . | . $
    | .## . . # . . ## | . $
    | .#####$##### | . $
    | .## . . # . . ## | . $
    |  .# . # . # . | . $
    |   . . . # . . . |   $
    |   .#####.    |   $
    |    .###.      |    $
    |     .#       |     $
    |      .       |      $
    |             |             ]

cursor_image = CREATE_CURSOR(strArray, HOTSPOT=hotspot, MASK=mask)
REGISTER_CURSOR, 'translate', IMAGE=cursor_image, $
HOTSPOT=hotspot, MASK=mask
```

After registering a cursor, you can set the current cursor using the [IDLgrWindow::SetCurrentCursor](#) method. The previous string results in the following cursor.



Figure 2: String Array Translated into a Cursor

Version History

Introduced: 6.1

CREATE_STRUCT

The CREATE_STRUCT function creates a structure given pairs of tag names and values. CREATE_STRUCT can also be used to concatenate structures.

Syntax

```
Result = CREATE_STRUCT( [Tag1, Values1, ..., Tagn, Valuesn] [, Structuresn]  
                        [, NAME=string])
```

or

```
Result = CREATE_STRUCT( [Tags, Values1, ..., Valuesn] [, Structuresn]  
                        [, NAME=string])
```

Return Value

Returns a structure composed of given pairs of tag names and values.

Arguments

Tags

The structure tag names. Tag names may be specified either as scalar strings or a single string array. If scalar strings are specified, values alternate with the tag names. If a string array is provided, values must still be specified individually. Tag names must be enclosed in quotes. Tag names may not be IDL [Reserved Words](#), and must be unique within a given structure, although the same tag name can be used in more than one structure. Tag names follow the rules of IDL identifiers: they must begin with a letter; following characters can be letters, digits, or the underscore or dollar sign characters; and case is ignored.

Note

If a tag name contains spaces, CREATE_STRUCT will replace the spaces with underscores. For example, if you specify a tag name of 'my tag', the tag will be created with the name 'my_tag'.

Values

The values for the structure fields. The number of *Values* arguments must match the number of *Tags* arguments (if tags are specified as scalar strings) or the number of elements of the *Tags* array (if tags are specified as a single array.)

Structures

One or more existing structure variables whose tags and values will be inserted into the new structure. When concatenating structures in this manner, the following rules apply:

- All tag names, whether specified via the *Tags* argument or in an existing structure variable, must be unique.
- Names of named structures included via the *Structures* arguments are not used in the newly-created structure.
- *Structures* arguments can be interspersed with groups of *Tags* and *Values* arguments in the call to `CREATE_STRUCT`. Use caution, however, to ensure that the number of *Tags* and *Values* in each group are equal, to avoid inserting a structure variable as the value of a single tag when you mean to include the structure's data as individual tags and values.

Keywords

NAME

To create a named structure, set this keyword equal to a string specifying the structure name. If this keyword is not present, an anonymous structure is created. Structure names must begin with a letter; following characters can be letters, digits, or the underscore or dollar sign characters; and case is ignored.

If `NAME` is specified and no plain arguments (tags, values, or structures) are present, then `CREATE_STRUCT` will return a structure of known type, either from IDL's internal table of already known named structures or by locating the appropriate `__define.pro` file for that structure in the current IDL search path (`!PATH`) and executing it. Hence, the following IDL statements are equivalent:

```
Result = { mystruct }
Result = CREATE_STRUCT(NAME='mystruct')
```

The `CREATE_STRUCT` version can be convenient in situations where the name of the structure is computed at runtime, and the `EXECUTE` function is not available (e.g., code running in the free IDL Virtual Machine environment, in which `EXECUTE` is disallowed).

Examples

To create the anonymous structure { A: 1, B: 'xxx' } in the variable *P*, enter:

```
p = CREATE_STRUCT('A', 1, 'B', 'xxx')
```

To add the fields “FIRST” and “LAST” to the structure, enter the following:

```
p = CREATE_STRUCT('FIRST', 0, p, 'LAST', 3)
```

The resulting structure contains { FIRST: 0, A: 1, B: 'xxx', LAST: 3}.

Finally, consider the following statements:

```
s1 = {Struct1, Tag1:'AAA', Tag2:'BBB'}
s2 = {Struct2, TagA:100, TagB:200}
s3 = CREATE_STRUCT(NAME='Struct3', ['A','B','C'], 1, 2, s1, s2)
```

Here, the variable `s3` contains the following named structure:

```
{ Struct3, A: 1, B: 2, C: {Struct1, Tag1: 'AAA', Tag2: 'BBB'}, TagA: 100, TagB: 200 }
```

Note that the value of `s3.C` is itself a “Struct1” structure, since the structure variable `s1` was interpreted as a *Values* argument, whereas the structure variable `s2` was interpreted as a *Structures* argument, thus including the tags from the “Struct2” structure directly in the new structure.

Version History

Introduced: Pre 4.0

NAME keyword added: 6.1

See Also

[IDL_VALIDNAME](#), [N_TAGS](#), [TAG_NAMES](#), [Chapter 9, “Structures”](#) in the *Building IDL Applications* manual.

CREATE_VIEW

The `CREATE_VIEW` procedure sets the various system variables required to define a coordinate system and a 3-D view. This procedure builds the system viewing matrix (!P.T) in such a way that the correct aspect ratio of the data is maintained even if the display window is not square. `CREATE_VIEW` also sets the “Data” to “Normal” coordinate conversion factors (!X.S, !Y.S, and !Z.S) so that center of the unit cube will be located at the center of the display window.

`CREATE_VIEW` sets the following IDL system variables:

!P.T	!X.S	!Y.S	!Z.S
!P.T3D	!X.Style	!Y.Style	!Z.Style
!P.Position	!X.Range	!Y.Range	!Z.Range
!P.Clip	!X.Margin	!Y.Margin	!Z.Margin
!P.Region			

This routine is written in the IDL language. Its source code can be found in the file `create_view.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
CREATE_VIEW [, AX=value] [, AY=value] [, AZ=value] [, PERSP=value]
  [, /RADIANS] [, WINX=pixels] [, WINY=pixels] [, XMAX=scalar]
  [, XMIN=scalar] [, YMAX=scalar] [, YMIN=scalar] [, ZFAC=value]
  [, ZMAX=scalar] [, ZMIN=scalar] [, ZOOM=scalar or 3-element vector]
```

Arguments

This procedure has no required arguments.

Keywords

AX

A floating-point value specifying the orientation (X rotation) of the view. The default is 0.0.

AY

A floating-point value specifying the orientation (Y rotation) of the view. The default is 0.0.

AZ

A floating-point value specifying the orientation (Z rotation) of the view. The default is 0.0.

PERSP

A floating-point value specifying the perspective projection distance. A value of 0.0 indicates an isometric projection (NO perspective). The default is 0.0.

RADIANS

Set this keyword if AX, AY, and AZ are specified in radians. The default is degrees.

WINX

A long integer specifying the X size, in pixels, of the window that the view is being set up for. The default is 640.

WINY

A long integer specifying the Y size, in pixels, of the window that the view is being set up for. The default is 512.

XMAX

A scalar specifying the maximum data value on the X axis. The default is 1.0.

XMIN

A scalar specifying the minimum data value on the X axis. The default is 0.0.

YMAX

A scalar specifying the maximum data value on the Y axis. The default is 1.0.

YMIN

A scalar specifying the minimum data value on the Y axis. The default is 0.0.

ZFAC

Set this keyword to a floating-point value to expand or contract the view in the Z dimension. The default is 1.0.

ZMAX

A scalar specifying the maximum data value on the Z axis. The default is 1.0.

ZMIN

A scalar specifying the minimum data value on the Z axis. The default is 0.0.

ZOOM

A floating-point number or 3-element vector specifying the view zoom factor. If zoom is a single value then the view will be zoomed equally in all 3 dimensions. If zoom is a 3-element vector then the view will be scaled zoom[0] in X, zoom[1] in Y, and zoom[2] in Z. The default is 1.0.

Examples

Set up a view to display an iso-surface from volumetric data. First, create some data:

```
vol = FLTARR(40, 50, 30)
vol(3:36, 3:46, 3:26) = RANDOMU(S, 34, 44, 24)
FOR I = 0, 10 DO vol = SMOOTH(vol, 3)
```

Generate the iso-surface.

```
SHADE_VOLUME, vol, 0.2, polygon_list, vertex_list, /LOW
```

Set up the view. Note that the subscripts into the Vol array range from 0 to 39 in X, 0 to 49 in Y, and 0 to 29 in Z. As such, the 3-D coordinates of the iso-surface (vertex_list) may have the same range. Set XMIN, YMIN, and ZMIN to zero (the default), and set XMAX=39, YMAX=49, and ZMAX=29.

```
WINDOW, XSIZE = 600, YSIZE = 400
CREATE_VIEW, XMAX = 39, YMAX = 49, ZMAX = 29, $
  AX = (-60.0), AZ = (30.0), WINX = 600, WINY = 400, $
  ZOOM = (0.7), PERSP = (1.0)
```

Display the iso-surface in the specified view.

```
img = POLYSHADE(polygon_list, vertex_list, /DATA, /T3D)
TVSCL, img
```

Version History

Introduced: Pre 4.0

See Also

[SCALE3](#), [T3D](#)

CROSSP

The CROSSP function returns a vector that is the cross-product of two input vectors, *V1* and *V2*.

Syntax

Result = CROSSP(*V1*, *V2*)

Return Value

Returns a floating-point vector that is the cross-product of two 3-element vectors, *V1* and *V2*.

Arguments

V1, V2

Three-element vectors.

Version History

Introduced: Original

See Also

[“Matrix Multiplication”](#) in Chapter 2 of the *Building IDL Applications* manual.

CRVLENGTH

The CRVLENGTH function computes the length of a curve with a tabular representation, $Y[i] = F(X[i])$.

Warning

Data that is highly oscillatory requires a sufficient number of samples for an accurate curve length computation.

This routine is written in the IDL language. Its source code can be found in the file `crvlength.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CRVLENGTH(*X*, *Y* [, /DOUBLE])

Return Value

Returns the curve length.

Arguments

X

An n -element single- or double-precision floating-point vector. X must contain at least three elements, and values must be specified in ascending order. Duplicate X values will result in a warning message.

Y

An n -element single- or double-precision floating-point vector.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Example

```
;Define a 21-element vector of X-values:
x = [-2.00, -1.50, -1.00, -0.50, 0.00, 0.50, 1.00, 1.50, 2.00, $
2.50, 3.00, 3.50, 4.00, 4.50, 5.00, 5.50, 6.00, 6.50, $
7.00, 7.50, 8.00]

;Define a 21-element vector of Y-values:
y = [-2.99, -2.37, -1.64, -0.84, 0.00, 0.84, 1.64, 2.37, 2.99, $
3.48, 3.86, 4.14, 4.33, 4.49, 4.65, 4.85, 5.13, 5.51, $
6.02, 6.64, 7.37]

;Compute the length of the curve:
result = CRVLENGTH(x, y)

Print, result
```

IDL prints:

```
14.8115
```

Version History

Introduced: 5.0

See Also

[INT_TABULATED](#), [PNT_LINE](#)

CT_LUMINANCE

The CT_LUMINANCE function calculates the luminance of colors.

This routine is written in the IDL language. Its source code can be found in the file `ct_luminance.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CT_LUMINANCE( [R, G, B] [, BRIGHT=variable] [, DARK=variable]  
[, /READ_TABLES] )
```

Return Value

The function returns an array containing the luminance values of the specified colors. If the *R*, *G*, and *B* parameters are not specified, or if *R* is of integer, byte or long type, the result is a longword array with the same number of elements as the input arguments. Otherwise, the result is a floating-point array with the same number of elements as the input arguments.

Arguments

R

An array representing the red color table. If omitted, the color values from either the `COLORS` common block, or the current color table are used.

G

An array representing the green color table. This parameter is optional.

B

An array representing the blue color table. This parameter is optional.

Keywords

BRIGHT

Set this keyword to a named variable in which the array index of the brightest color is returned.

DARK

Set this keyword to a named variable in which the array index of the darkest color is returned.

READ_TABLES

Set this keyword, and don't specify the *R*, *G*, and *B* arguments, to read colors directly from the current colortable (using TVLCT, /GET) instead of using the COLORS common block.

Version History

Introduced: Pre 4.0

See Also

[GAMMA_CT](#), [STRETCH](#)

CTI_TEST

The CTI_TEST function constructs a “contingency table” from an array of observed frequencies and tests the hypothesis that the rows and columns are independent using an extension of the chi-square goodness-of-fit test.

This routine is written in the IDL language. Its source code can be found in the file `cti_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CTI_TEST( Obfreq [, COEFF=variable] [, /CORRECTED]
  [, CRAMV=variable] [, DF=variable] [, EXFREQ=variable]
  [, RESIDUAL=variable] )
```

Return Value

Returns a two-element vector containing the chi-square test statistic X2 and the one-tailed probability of obtaining a value of X2 or greater.

Arguments

Obfreq

An $m \times n$ array containing observed frequencies. *Obfreq* can contain either integer, single-, double-precision floating-point values.

Keywords

COEFF

Set this keyword to a named variable that will contain the Coefficient of Contingency. The Coefficient of Contingency is a non-negative scalar, in the interval [0.0, 1.0], which measures the degree of dependence within a contingency table. The larger the value of COEFF, the greater the degree of dependence.

CORRECTED

Set this keyword to use the “Yate’s Correction for Continuity” when computing the Chi-squared test statistic, X2. The Yate’s correction always decreases the magnitude of X2. In general, this keyword should be set for small sample sizes.

CRAMV

Set this keyword to a named variable that will contain Cramer's V. Cramer's V is a non-negative scalar, in the interval [0.0, 1.0], which measures the degree of dependence within a contingency table.

DF

Set this keyword to a named variable that will contain the number of degrees of freedom used to compute the probability of obtaining the value of the Chi-squared test statistic or greater. $DF = (n - 1) * (m - 1)$ where m and n are the number of columns and rows of the contingency table, respectively.

EXFREQ

Set this keyword to a named variable that will contain an array of m -columns and n -rows containing expected frequencies. The elements of this array are often referred to as the "cells" of the expected frequencies. The expected frequency of each cell is computed as the product of row and column marginal frequencies divided by the overall total of observed frequencies.

RESIDUAL

Set this keyword to a named variable that will contain an array of m -columns and n -rows containing signed differences between corresponding cells of observed frequencies and expected frequencies.

Examples

Define a 5-column and 4-row array of observed frequencies.

```
obfreq = [[748, 821, 786, 720, 672], $
          [ 74,  60,  51,  66,  50], $
          [ 31,  25,  22,  16,  15], $
          [  9,  10,   6,   5,   7]]
```

Test the hypothesis that the rows and columns of "obfreq" contain independent data at the 0.05 significance level.

```
result = CTI_TEST(obfreq, COEFF = coeff)
```

The result should be the two-element vector [14.3953, 0.276181].

The computed value of 0.276181 indicates that there is no reason to reject the proposed hypothesis at the 0.05 significance level. The Coefficient of Contingency returned in the parameter "coeff" (coeff = 0.0584860) also indicates the lack of

dependence between the rows and columns of the observed frequencies. Setting the `CORRECTED` keyword returns the two-element vector `[12.0032, 0.445420]` and `(coeff = 0.0534213)` resulting in the same conclusion of independence.

Version History

Introduced: Pre 4.0

See Also

[CORRELATE](#), [M_CORRELATE](#), [XSQ_TEST](#)

CURSOR

The CURSOR procedure is used to read the position of the interactive graphics cursor from the current graphics device. Note that not all graphics devices have interactive cursors. CURSOR enables the graphic cursor on the device and optionally waits for the operator to position it. On devices that have a mouse, CURSOR normally waits until a mouse button is pressed (or already down). If no mouse buttons are present, CURSOR waits for a key on the keyboard to be pressed.

The system variable !MOUSE is set to the button status. Each mouse button is assigned a bit in !MOUSE, bit 0 is the left most button, bit 1 the next, etc. See “!MOUSE” on page 4361 for details.

Using CURSOR with Draw Widgets

Note that the CURSOR procedure is only for use with IDL graphics windows. It should not be used with draw widgets. To obtain the cursor position and button state information from a draw widget, examine the X, Y, PRESS, and RELEASE fields in the structures returned by the draw widget in response to cursor events.

Using CURSOR with the TEK Device

Note that for the CURSOR procedure to work properly with Tektronix terminals, you may need to execute the command, `DEVICE, GIN_CHARS=6`.

Syntax

```
CURSOR, X, Y [, Wait / [, /CHANGE | , /DOWN | , /NOWAIT | , /UP | , /WAIT]]  
      [, /DATA | , /DEVICE, | , /NORMAL]
```

Arguments

X

A named variable to receive the cursor's current column position.

Y

A named variable to receive the cursor's current row position.

Wait

An integer that specifies the conditions under which `CURSOR` returns. This parameter can be used interchangeably with the keyword parameters listed below that specify the type of wait. The default value is 1. The table below describes each type of wait.

Note that not all modes of waiting work with all display devices.

Wait Value	Corresponding Keyword	Action
0	NOWAIT	Return immediately.
1	WAIT	Return if a button is down.
2	CHANGE	Return if a button is pressed, released, or the pointer is moved.
3	DOWN	Return when a button down transition is detected.
4	UP	Return when a button up transition is detected.

Table 3-8: Values for CURSOR Wait Parameter

Keywords

CHANGE

Set this keyword to wait for pointer movement or button transition within the currently selected window.

DATA

Set this keyword to return *X* and *Y* in data coordinates.

DOWN

Set this keyword to wait for a button down transition within the currently selected window.

DEVICE

Set this keyword to return *X* and *Y* in device coordinates.

NORMAL

Set this keyword to return X and Y in normalized coordinates.

NOWAIT

Set this keyword to read the pointer position and button status and return immediately. If the pointer is not within the currently selected window, the device coordinates -1, -1 are returned.

UP

Set this keyword to wait for a button up transition within the current window.

WAIT

Set this keyword to wait for a button to be depressed within the currently selected window. If a button is already pressed, return immediately.

Examples

Activate the graphics cursor, select a point in the graphics window, and return the position of the cursor in device coordinates. Enter:

```
CURSOR, X, Y, /DEVICE
```

Move the cursor over the graphics window and press the mouse button. The position of the cursor in device coordinates is stored in the variables X and Y . To label the location, enter:

```
XYOUTS, X, Y, 'X marks the spot.', /DEVICE
```

Version History

Introduced: Original

See Also

[RDPIX](#), [TVCRS](#), [CURSOR_CROSSHAIR](#) (and other `CURSOR_` keywords), [WIDGET_DRAW](#), “!MOUSE” on page 4361

CURVEFIT

The CURVEFIT function uses a gradient-expansion algorithm to compute a non-linear least squares fit to a user-supplied function with an arbitrary number of parameters. The user-supplied function may be any non-linear function where the partial derivatives are known or can be approximated. Iterations are performed until the chi square changes by a specified amount, or until a maximum number of iterations have been performed.

This routine is written in the IDL language. Its source code can be found in the file `curvefit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CURVEFIT( X, Y, Weights, A [, Sigma] [, CHISQ=variable] [, /DOUBLE]
  [, FITA=vector] [, FUNCTION_NAME=string] [, ITER=variable]
  [, ITMAX=value] [, /NODERIVATIVE] [, STATUS={0 | 1 | 2}] [, TOL=value]
  [, YERROR=variable] )
```

Return Value

Returns a vector of values for the dependent variables, as fitted by the function fit. If *A* is double-precision or if the `DOUBLE` keyword is set, calculations are performed in double-precision arithmetic, otherwise they are performed in single-precision arithmetic.

Arguments

X

An *n*-element vector of independent variables.

Y

A vector of dependent variables. *Y* must have the same number of elements as *F* returned by the user-defined function.

Weights

For instrumental (Gaussian) weighting, set $Weights_i = 1.0/\text{standard_deviation}(Y_i)^2$. For statistical (Poisson) weighting, $Weights_i = 1.0/Y_i$. For no weighting, set

$Weights_i = 1.0$. If *Weights* is set to an undefined variable then no weighting will be used.

A

A vector with as many elements as the number of terms in the user-supplied function, containing the initial estimate for each parameter. On return, the vector *A* contains the fitted model parameters.

Sigma

A named variable that will contain a vector of standard deviations for the elements of the output vector *A*.

Note

If *Weights* is omitted, then you are assuming that your supplied model is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned for the *Sigma* argument are multiplied by $\text{SQRT}(\text{CHISQ}/(N * M))$, where *N* is the number of points in *X*, and *M* is the number of coefficients. See Section 15.2 of *Numerical Recipes in C (Second Edition)* for details.

Keywords

CHISQ

Set this keyword to a named variable that will contain the value of the reduced chi-square goodness-of-fit statistic.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FITA

Set this keyword to a vector, with as many elements as *A*, which contains a zero for each fixed parameter, and a non-zero value for elements of *A* to fit. If not supplied, all parameters are taken to be non-fixed.

FUNCTION_NAME

Use this keyword to specify the name of the function to fit. If this keyword is omitted, CURVEFIT assumes that the IDL procedure `FUNCT` is to be used. If `FUNCT` is not already compiled, IDL compiles the function from the file `funct.pro`, located in the

lib subdirectory of the IDL distribution. `FUNCT` evaluates the sum of a Gaussian and a second-order polynomial.

The function to be fit must be written as an IDL procedure and compiled prior to calling `CURVEFIT`. The procedure must accept values of X (the independent variable), and A (the fitted function's initial parameter values). It must return values for F (the function's value at X), and optionally $PDER$ (a 2D array of partial derivatives).

The return value for F must have the same number of elements as Y . The return value for $PDER$ (if supplied) must be a 2D array with dimensions `[N_ELEMENTS(Y), N_ELEMENTS(A)]`.

See the *Example* section below for an example function.

ITER

Set this keyword equal to a named variable that will contain the actual number of iterations performed.

ITMAX

Set this keyword to specify the maximum number of iterations. The default value is 20.

NODERIVATIVE

If this keyword is set, the routine specified by the `FUNCTION_NAME` keyword will not be requested to provide partial derivatives. The partial derivatives will be estimated by `CURVEFIT` using forward differences. If analytical derivatives are available they should always be used.

STATUS

Set this keyword to a named variable that will contain an integer indicating the status of the computation. Possible return values are:

0	The computation was successful.
1	The computation failed. Chi-square was increasing without bounds.
2	The computation failed to converge in <code>ITMAX</code> iterations.

TOL

Use this keyword to specify the desired convergence tolerance. The routine returns when the relative decrease in chi-squared is less than TOL in one iteration. The default value is 1.0×10^{-3} .

YERROR

Set this keyword to a named variable that will contain the standard error between YFIT and Y.

Examples

Fit a function of the form $F(x) = a * \exp(b*x) + c$ to sample pairs contained in arrays *X* and *Y*. The partial derivatives are easily computed symbolically:

```
df/da = EXP(b*x)
df/db = a * x * EXP(b*x)
df/dc = 1.0
```

First, define a procedure to return $F(x)$ and the partial derivatives, given *X*. Note that *A* is an array containing the values *a*, *b*, and *c*.

```
PRO gfunct, X, A, F, pder
  bx = EXP(A[1] * X)
  F = A[0] * bx + A[2]

  ;If the procedure is called with four parameters, calculate the
  ;partial derivatives.
  IF N_PARAMS() GE 4 THEN $
    pder = [[bx], [A[0] * X * bx], [replicate(1.0, N_ELEMENTS(X))]]
  END
```

Compute the fit to the function we have just defined. First, define the independent and dependent variables:

```
X = FLOAT(INDGEN(10))
Y = [12.0, 11.0, 10.2, 9.4, 8.7, 8.1, 7.5, 6.9, 6.5, 6.1]

;Define a vector of weights.
weights = 1.0/Y

;Provide an initial guess of the function's parameters.
A = [10.0,-0.1,2.0]

;Compute the parameters.
yfit = CURVEFIT(X, Y, weights, A, SIGMA, FUNCTION_NAME='gfunct')
```

```
;Print the parameters returned in A.  
PRINT, 'Function parameters: ', A
```

IDL prints:

```
Function parameters:      9.91120      -0.100883      2.07773
```

Thus, the function that best fits the data is:

$$f(x) = 9.91120(e^{-0.100883x}) + 2.07773$$

Version History

Introduced: Original

YERROR keyword added: 5.6

FITA and STATUS keywords added: 6.0

See Also

[COMFIT](#), [GAUSS2DFIT](#), [GAUSSFIT](#), [LMFIT](#), [POLY_FIT](#), [REGRESS](#), [SFIT](#),
[SVDFIT](#)

CV_COORD

The CV_COORD function converts 2D and 3D coordinates between the rectangular, polar, cylindrical, and spherical coordinate systems.

This routine is written in the IDL language. Its source code can be found in the file `cv_coord.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CV_COORD( [, /DEGREES] [, /DOUBLE] [, FROM_CYLIN=cyl_coords |
  , FROM_POLAR=pol_coords | , FROM_RECT=rect_coords |
  , FROM_SPHERE=sph_coords] [, /TO_CYLIN | , /TO_POLAR | , /TO_RECT |
  , /TO_SPHERE] )
[, /=xxx] [, /=xxx]
[, /=xxx]
```

Return Value

If the value specified in the “FROM_” keyword is double precision, or if the DOUBLE keyword is set, then all calculations are performed in double precision and the returned value is double precision. Otherwise, single precision is used. If none of the “FROM_” keyword are specified, 0 is returned. If none of the “TO_” keywords are specified, the input coordinates are returned.

Arguments

This function has no required arguments. All data is passed in via keywords.

Keywords

DEGREES

If set, then the input and output coordinates are in degrees (where applicable). Otherwise, the angles are in radians.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FROM_CYLIN

A vector of the form [*angle*, *radius*, *z*], or a (3, *n*) array of cylindrical coordinates to convert.

FROM_POLAR

A vector of the form [*angle*, *radius*], or a (2, *n*) array of polar coordinates to convert.

FROM_RECT

A vector of the form [*x*, *y*] or [*x*, *y*, *z*], or a (2, *n*) or (3, *n*) array containing rectangular coordinates to convert.

FROM_SPHERE

A vector of the form [*longitude*, *latitude*, *radius*], or a (3, *n*) array of spherical coordinates to convert.

TO_CYLIN

If set, cylindrical coordinates are returned in a vector of the form [*angle*, *radius*, *z*], or a (3, *n*) array.

TO_POLAR

If set, polar coordinates are returned in a vector of the form [*angle*, *radius*], or a (2, *n*) array.

TO_RECT

If set, rectangular coordinates are returned in a vector of the form [*x*, *y*] or [*x*, *y*, *z*], or a (2, *n*) or (3, *n*) array.

TO_SPHERE

If set, spherical coordinates are returned in a vector of the form [*longitude*, *latitude*, *radius*], or a (3, *n*) array.

Examples

Convert from spherical to cylindrical coordinates:

```
sph_coord = [[45.0, -60.0, 10.0], [0.0, 0.0, 0.0]]
rect_coord = CV_COORD(FROM_SPHERE=sph_coord, /TO_CYLIN, /DEGREES)
```

Convert from rectangular to polar coordinates:

```
rect_coord = [10.0, 10.0]  
polar_coord = CV_COORD(FROM_RECT=rect_coord, /TO_POLAR)
```

Version History

Introduced: Pre 4.0

See Also

[CONVERT_COORD](#), [COORD2TO3](#), [CREATE_VIEW](#), [SCALE3](#), [T3D](#)

CVTTOBM

The CVTTOBM function converts a byte array in which each byte represents one pixel into a “bitmap byte array” in which each bit represents one pixel. This is useful when creating bitmap labels for buttons created with the WIDGET_BUTTON function.

Most of IDL’s image file format reading functions (READ_BMP, READ_PICT, etc.) return a byte array which must be converted before use as a button label. Note that there is one exception to this rule; the READ_X11_BITMAP routine returns a bitmap byte array that needs no conversion before use.

This routine is written in the IDL language. Its source code can be found in the file `cvttobm.pro` in the `lib` subdirectory of the IDL distribution.

Note

IDL supports color bitmaps for button labels. The IDL GUIBuilder has a Bitmap Editor that allows you to create color bitmaps for button labels. The BITMAP keyword to WIDGET_BUTTON specifies that the button label is a color bitmap.

Syntax

Result = CVTTOBM(*Array* [, THRESHOLD=*value*{0 to 255}])

Return Value

Returns a bitmap byte array. Bitmap byte arrays are monochrome; by default, CVTTOBM converts pixels that are darker than the median value to black and pixels that are lighter than the median value to white. You can supply a different threshold value via the THRESHOLD keyword.

Arguments

Array

A 2-dimensional pixel array, one byte per pixel.

Keywords

THRESHOLD

A byte value (or an integer value between 0 and 255) to be used as a threshold value when determining if a particular pixel is black or white. If `THRESHOLD` is not specified, the threshold is calculated to be the average of the input array.

Examples

The following example creates a bitmap button label from a byte array:

```
; Create a byte array:
image = BYTSCL(DIST(100))
; Create a widget base:
base = WIDGET_BASE(/COLUMN)

; Use CVTTOBM to create a bitmap byte array for a button label:
button = WIDGET_BUTTON(base, VALUE = CVTTOBM(image))

; Realize the widget:
WIDGET_CONTROL, base, /REALIZE
```

Version History

Introduced: 5.0

See Also

[WIDGET_BUTTON](#), [XBM_EDIT](#), “Using the Bitmap Editor” in Chapter 27 of the *Building IDL Applications* manual.

CW_ANIMATE

The CW_ANIMATE function creates a compound widget that displays an animated sequence of images using off-screen windows known as *pixmap*s. The speed and direction of the display can be adjusted using the widget interface.

CW_ANIMATE provides the graphical interface used by the XINTERANIMATE procedure, which is the preferred routine for displaying animation sequences in most situations. Use this widget instead of XINTERANIMATE when you need to run multiple instances of the animation widget simultaneously. Note that if more than one animation widget is running, they will have to share resources and will display images more slowly than a single instance of the widget.

This routine is written in the IDL language. Its source code can be found in the file `cw_animate.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_ANIMATE

Unlike XINTERANIMATE, using the CW_ANIMATE widget requires calls to two separate procedures, CW_ANIMATE_LOAD and CW_ANIMATE_RUN, to load the images to be animated and to run the animation. Alternatively, you can supply a vector of pre-existing pixmap window IDs, eliminating the need to use CW_ANIMATE_LOAD. The vector of pixmaps is commonly obtained from a call to CW_ANIMATE_GETP applied to a previous animation widget. Once the images are loaded, they are displayed by copying the images from the pixmap or buffer to the visible draw widget.

See the documentation for CW_ANIMATE_LOAD, CW_ANIMATE_RUN, and CW_ANIMATE_GETP for more information.

The only event returned by CW_ANIMATE indicates that the user has clicked on the “End Animation” button. The parent application should use this as a signal to kill the animation widget via WIDGET_CONTROL. When the widget is destroyed, the pixmaps used in the animation are destroyed as well, unless they were saved by a call to CW_ANIMATE_GETP.

See the animation widget’s help file (available by clicking the “Help” button on the widget) for more information about the widget’s controls.

Syntax

```
Result = CW_ANIMATE( Parent, Sizex, Sizey, Nframes [, /NO_KILL]
  [, OPEN_FUNC=string] [, PIXMAPS=vector] [, TAB_MODE=value]
  [, /TRACK] [, UNAME=string] [, UVALUE=value] )
```

Return Value

This function returns the widget ID of the newly-created animation widget.

Arguments

Parent

The widget ID of the parent widget.

Size_x

The width of the displayed image, in pixels.

Size_y

The height of the displayed image, in pixels

Nframes

The number of frames in the animation sequence.

Keywords

NO_KILL

Set this keyword to omit the “End Animation” button from the animation widget.

OPEN_FUNC

Set this keyword equal to a scalar string specifying the name of a user-written function that loads animation data. If a function is specified, an “Open ...” button is added to the animation widget.

PIXMAPS

Use this keyword to provide the animation widget with a vector of pre-existing pixmap (off screen window) IDs. This vector is usually obtained from a call to `CW_ANIMATE_GETP` applied to a previous animation widget.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is

inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-9: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TRACK

Set this keyword to cause the frame slider to track the frame number of the currently-displayed frame.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget’s base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using [WIDGET_CONTROL](#) and [WIDGET_INFO](#).

Widget Events Returned by the CW_ANIMATE Widget

The only event returned by this widget indicates that the user has pressed the DONE button. The parent application should use this as a signal to kill the animation widget via WIDGET_CONTROL.

Examples

Assume the following event handler procedure exists:

```
PRO EHANDLER, EV
  WIDGET_CONTROL, /DESTROY, EV.TOP
end
```

Tip

If you wish to create this event handler starting from the IDL command prompt, remember to begin with the `.RUN` command.

Enter the following commands to open the file `ABNORM.DAT` (a series of images of a human heart) and load the images it contains into an array `H`.

```
OPENR, 1, FILEPATH('abnorm.dat', SUBDIR = ['examples', 'data'])
H = BYTARR(64, 64, 16)
READU, 1, H
CLOSE, 1
H = REBIN(H, 128, 128, 16)
```

Create an instance of the animation widget and load the frames. Note that because the animation widget is realized before the call to `CW_ANIMATE_LOAD`, the frames

are displayed as they are loaded. This provides the user with an indication of how things are progressing.

```
base = WIDGET_BASE(TITLE = 'Animation Widget')
animate = CW_ANIMATE(base, 128, 128, 16)
WIDGET_CONTROL, /REALIZE, base
FOR I=0,15 DO CW_ANIMATE_LOAD, animate, FRAME=I, IMAGE=H[*,*,I]
```

Save the pixmap window IDs for future use:

```
CW_ANIMATE_GETP, animate, pixmap_vect
```

Start the animation:

```
CW_ANIMATE_RUN, animate
XMANAGER, 'CW_ANIMATE Demo', base, EVENT_HANDLER = 'EHANDLER'
```

Pressing the “End Animation” button kills the application.

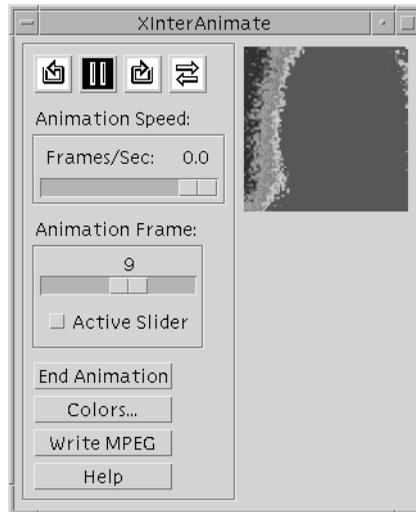


Figure 3-13: The animation interface created by CW_ANIMATE

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[CW_ANIMATE_LOAD](#), [CW_ANIMATE_RUN](#), [CW_ANIMATE_GETP](#),
[XINTERANIMATE](#)

CW_ANIMATE_GETP

The CW_ANIMATE_GETP procedure gets a copy of the vector of pixmap window IDs being used by a CW_ANIMATE animation widget. If this routine is called, CW_ANIMATE does not destroy the pixmaps when it is destroyed. You can then provide the pixmaps to a later instance of CW_ANIMATE to re-use them, skipping the pixmap creation and rendering step (CW_ANIMATE_LOAD).

CW_ANIMATE provides the graphical interface used by the XINTERANIMATE procedure, which is the preferred routine for displaying animation sequences in most situations. Use this widget instead of XINTERANIMATE when you need to run multiple instances of the animation widget simultaneously. Note that if more than one animation widget is running, they will have to share resources and will display images more slowly than a single instance of the widget.

This routine is written in the IDL language. Its source code can be found in the file `cw_animate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
CW_ANIMATE_GETP, Widget, Pixmaps [, /KILL_ANYWAY]
```

Arguments

Widget

The widget ID of the animation widget (created with CW_ANIMATE) that contains the pixmaps.

Pixmaps

A named variable that will contain a vector of the window IDs of the pixmap windows.

Keywords

KILL_ANYWAY

Set this keyword to ensure that the pixmaps are destroyed anyway when CW_ANIMATE exits, despite the fact that CW_ANIMATE_GETP has been called.

Example

See “[CW_ANIMATE](#)” on page 384.

Version History

Introduced: Pre 4.0

See Also

[CW_ANIMATE](#), [CW_ANIMATE_LOAD](#), [CW_ANIMATE_RUN](#),
[XINTERANIMATE](#)

CW_ANIMATE_LOAD

The CW_ANIMATE_LOAD procedure creates an array of pixmaps which are loaded into a CW_ANIMATE compound widget.

CW_ANIMATE provides the graphical interface used by the XINTERANIMATE procedure, which is the preferred routine for displaying animation sequences in most situations. Use this widget instead of XINTERANIMATE when you need to run multiple instances of the animation widget simultaneously. Note that if more than one animation widget is running, they will have to share resources and will display images more slowly than a single instance of the widget.

This routine is written in the IDL language. Its source code can be found in the file `cw_animate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
CW_ANIMATE_LOAD, Widget [, /CYCLE] [, FRAME=value{0 to NFRAMES}]
  [, IMAGE=value] [, /ORDER] [, WINDOW=[window_num [, X0, Y0, Sx, Sy]]]
  [, XOFFSET=pixels] [, YOFFSET=pixels]
```

Arguments

Widget

The widget ID of the animation widget (created with CW_ANIMATE) into which the image should be loaded.

Keywords

CYCLE

Set this keyword to cause the animation to cycle. Normally, frames are displayed going either forward or backward. If CYCLE is set, the animation reverses direction after the last frame in either direction is displayed.

FRAME

The frame number to be loaded. This is a value between 0 and NFRAMES. If not supplied, frame 0 is loaded.

IMAGE

The image to be loaded. IMAGE can either be a 2D or a 3D (24-bit) image.

ORDER

Set this keyword to display images from the top down instead of the default bottom up. This keyword is only used when loading images with the IMAGE keyword.

WINDOW

When this keyword is specified, an image is copied from an existing window to the animation pixmap. Under some windowing systems, this technique is much faster than reading from the display and then loading with the IMAGE keyword.

The value of this parameter is either an IDL window number (in which case the entire window is copied), or a vector containing the window index and the rectangular bounds of the area to be copied. For example:

```
WINDOW = [Window_Number, X0, Y0, Sx, Sy]
```

XOFFSET

The horizontal offset, in pixels from the left of the frame, of the image in the destination window.

YOFFSET

The vertical offset, in pixels from the bottom of the frame, of the image in the destination window.

Example

See the documentation for CW_ANIMATE for an example using this procedure. Note that if the widget is realized before calls to CW_ANIMATE_LOAD, the frames are displayed as they are loaded.

Version History

Introduced: Pre 4.0

See Also

[CW_ANIMATE](#), [CW_ANIMATE_GETP](#), [CW_ANIMATE_RUN](#),
[XINTERANIMATE](#)

CW_ANIMATE_RUN

The CW_ANIMATE_RUN procedure displays a series of images that have been loaded into a CW_ANIMATE compound widget by a call to CW_ANIMATE_LOAD.

CW_ANIMATE provides the graphical interface used by the XINTERANIMATE procedure, which is the preferred routine for displaying animation sequences in most situations. Use this widget instead of XINTERANIMATE when you need to run multiple instances of the animation widget simultaneously. Note that if more than one animation widget is running, they will have to share resources and will display images more slowly than a single instance of the widget.

This routine is written in the IDL language. Its source code can be found in the file `cw_animate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
CW_ANIMATE_RUN, Widget [, Rate{0 to 100}] [, NFRAMES=value] [, /STOP]
```

Arguments

Widget

The widget ID of the animation widget (created with CW_ANIMATE) that will display the animation.

Rate

A value between 0 and 100 that represents the speed of the animation as a percentage of the maximum display rate. The fastest animation has a value of 100 and the slowest has a value of 0. The default animation rate is 100.

The animation rate can also be adjusted after the animation has begun by changing the value of the “Animation Speed” slider.

Keywords

NFRAMES

Set this keyword equal to the number of frames to animate. This number must be less than or equal to the *Nframes* argument to CW_ANIMATE.

STOP

If this keyword is set, the animation is stopped.

Example

See “[CW_ANIMATE](#)” on page 384.

Version History

Introduced: Pre 4.0

See Also

[CW_ANIMATE](#), [CW_ANIMATE_GETP](#), [CW_ANIMATE_LOAD](#),
[XINTERANIMATE](#)

CW_ARCBALL

The `CW_ARCBALL` function creates a compound widget for intuitively specifying three-dimensional orientations.

The user drags a simulated track-ball with the mouse to interactively obtain arbitrary rotations. Sequences of rotations may be cascaded. The rotations may be unconstrained (about any axis), constrained to the view X, Y, or Z axes, or constrained to the object's X, Y, or Z axis.

This widget is based on "ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse," by Ken Shoemake, Computer Graphics Laboratory, University of Pennsylvania, Philadelphia, PA 19104.

This widget can generate any rotation about any axis. Note, however, that not all rotations are compatible with the IDL `SURFACE` procedure, which is restricted to rotations that project the object Z axis parallel to the view Y axis.

This routine is written in the IDL language. Its source code can be found in the file `cw_arcball.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_ARCBALL

Use the command:

```
WIDGET_CONTROL, id, GET_VALUE = matrix
```

to return the current 3x3 rotation matrix in the variable `matrix`.

You can set the arcball to new rotation matrix using the command:

```
WIDGET_CONTROL, id, SET_VALUE = matrix
```

after the widget is initially realized.

Syntax

```
Result = CW_ARCBALL( Parent [, COLORS=array] [, /FRAME]
  [, LABEL=string] [, RETAIN={0 | 1 | 2}] [, SIZE=pixels] [, TAB_MODE=value]
  [, /UPDATE] [, UNAME=string] [, UVALUE=value] [, VALUE=array] )
```

Return Value

This function returns the widget ID of the newly-created ARCBALL widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

COLORS

A 6-element array containing the color indices to be used.

- Colors[0] = view axis color,
- Colors[1] = object axis color,
- Colors[2] = XZ plane +Y side (body top) color,
- Colors[3] = YZ plane (fin) color,
- Colors[4] = XZ plane -Y side (body bottom),
- Colors[5] = background color.

For devices that are using indexed color (i.e., DECOMPOSED=0), the default value for COLORS is [1 , 7 , 2 , 3 , 7 , 0], which yields good colors with the TEK_COLOR table: (white, yellow, red, green, yellow, black). For devices that are using decomposed color (i.e., DECOMPOSED=1), the default value is an array of corresponding decomposed (rather than indexed) colors: (white, yellow, red, green, yellow, black).

For more information on decomposed color, refer to the [DECOMPOSED](#) keyword to the DEVICE routine.

FRAME

Set this keyword to draw a frame around the widget.

LABEL

Set this keyword to a string containing the widget's label.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, "Using Language Catalogs"](#) in the *Building IDL Applications* manual.

RETAIN

Set this keyword to zero, one, or two to specify how backing store should be handled for the draw widget. `RETAIN=0` specifies no backing store. `RETAIN=1` requests that the server or window system provide backing store. `RETAIN=2` specifies that IDL provide backing store directly. See “[Backing Store](#)” on page 4286 for details.

SIZE

The size of the square drawable area containing the arcball, in pixels. The default is 192.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-10: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UPDATE

Set this keyword to cause the widget will send an event each time the mouse button is released after a drag operation. By default, events are only sent when the “Update” button is pressed.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

VALUE

Set this keyword to a 3 x 3 array that will be the initial value for the rotation matrix. `VALUE` must be a valid rotation matrix (no translation or perspective) where $\text{TRANSPPOSE}(\text{VALUE}) = \text{INVERSE}(\text{VALUE})$. This can be the upper-left corner of `!P.T` after executing the command

```
T3D, /RESET, ROTATE = [x,y,z].
```

The default is the identity matrix.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget’s base widget. This means that many keywords to the `WIDGET_CONTROL` and `WIDGET_INFO` routines that affect or return information on base widgets can be used with compound widgets.

In addition, you can use the `GET_VALUE` and `SET_VALUE` keywords to `WIDGET_CONTROL` to obtain or set the 3 x 3 rotation matrix in the arcball widget.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by the CW_ARCBALL Widget

Arcball widgets generate event structures with the following definition:

```
event = {ID:0L, TOP:0L, HANDLER:0L, VALUE:fltarr(3,3) }
```

The `VALUE` field contains the 3 x 3 array representing the new rotation matrix.

Examples

See the procedure `ARCBALL_TEST`, contained in the `cw_arcball.pro` file. To test `CW_ARCBALL`, enter the following commands:

```
.RUN cw_arcball
ARCBALL_TEST
```

This results in the following:

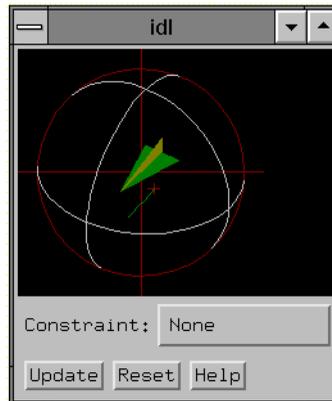


Figure 3-14: The `CW_ARCBALL` widget.

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[CREATE_VIEW](#), [SCALE3](#), [T3D](#)

CW_BGROU

The CW_BGROU function creates a widget base of buttons. It handles the details of creating the proper base (standard, exclusive, or non-exclusive) and filling in the desired buttons. Events for the individual buttons are handled transparently, and a CW_BGROU event returned. This event can return any one of the following:

- the index of the button within the base,
- the widget ID of the button,
- the name of the button,
- an arbitrary value taken from an array of user values.

Only buttons with textual names are handled by this widget. Bitmaps are not understood.

This routine is written in the IDL language. Its source code can be found in the file `cw_bgrou.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_BGROU( Parent, Names [, BUTTON_UVALUE=array]
  [, COLUMN=value] [, EVENT_FUNC=string] [{, /EXCLUSIVE | ,
  /NONEXCLUSIVE} | [, SPACE=pixels] [, XPAD=pixels] [, YPAD=pixels]]
  [, FONT=font] [, FRAME=width] [, IDS=variable] [, /LABEL_LEFT=string | ,
  /LABEL_TOP=string] [, /MAP] [, /NO_RELEASE] [, /RETURN_ID | ,
  /RETURN_INDEX | , /RETURN_NAME] [, ROW=value] [, /SCROLL]
  [, SET_VALUE=value] [, SPACE=value] [, TAB_MODE=value]
  [, X_SCROLL_SIZE=width] [, Y_SCROLL_SIZE=height]
  [, SET_VALUE=value] [, UNAME=string] [, UVALUE=value]
  [, XOFFSET=value] [, XSIZE=width] [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

This function returns the widget ID of the newly-created button group widget.

Arguments

Parent

The widget ID of the parent widget.

Names

A string array, one string per button, giving the name of each button.

Keywords

BUTTON_UVALUE

An array of user values to be associated with each button and returned in the event structure. If this keyword is set, the user values are always returned, even if any of the RETURN_ID, RETURN_INDEX, or RETURN_NAME keywords are set.

COLUMN

Buttons will be arranged in the number of columns specified by this keyword.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget. This function is called with the return value structure whenever a button is pressed, and follows the conventions for user-written event functions.

EXCLUSIVE

Set this keyword to cause buttons to be placed in an exclusive base, in which only one button can be selected at a time.

FONT

The name of the font to be used for the button titles. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See [“About Device Fonts”](#) on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

FRAME

Specifies the width of the frame to be drawn around the base.

IDS

A named variable in which the button IDs will be stored, as a longword vector.

LABEL_LEFT

Set this keyword to a string creating a text label to the left of the buttons.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

LABEL_TOP

Set this keyword to a string creating a text label above the buttons.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

MAP

Set this keyword to cause the base to be mapped when the widget is realized (the default).

NONEXCLUSIVE

Set this keyword to cause buttons to be placed in an non-exclusive base, in which any number of buttons can be selected at once.

NO_RELEASE

If set, button release events will not be returned.

RETURN_ID

Set this keyword to return the widget ID of the button in the VALUE field of returned events. This keyword is ignored if the BUTTON_UVALUE keyword is set.

RETURN_INDEX

Set this keyword to return the zero-based index of the button within the base in the VALUE field of returned events. This keyword is ignored if the BUTTON_UVALUE keyword is set. THIS IS THE DEFAULT.

RETURN_NAME

Set this keyword to return the name of the button within the base in the VALUE field of returned events. This keyword is ignored if the BUTTON_UVALUE keyword is set.

ROW

Buttons will be arranged in the number of rows specified by this keyword.

SCROLL

If set, the base will include scroll bars to allow viewing a large base through a smaller viewport.

SET_VALUE

Allows changing the current state of toggle buttons (i.e., exclusive and nonexclusive groups of buttons). The behavior of SET_VALUE differs between EXCLUSIVE and NONEXCLUSIVE CW_BGROU widgets. With EXCLUSIVE CW_BGROU widgets, the argument to SET_VALUE is the id of the widget to be turned on. With NONEXCLUSIVE CW_BGROU widgets the argument to SET_VALUE should be an array of on/off flags for the array of buttons.

SPACE

The space, in pixels, to be left around the edges of a row or column major base.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-11: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

XOFFSET

The X offset of the widget relative to its parent.

XPAD

The horizontal space, in pixels, between children of a row or column major base.

XSIZE

The width of the base.

X_SCROLL_SIZE

The width of the viewport if SCROLL is specified.

YOFFSET

The Y offset of the widget relative to its parent.

YPAD

The vertical space, in pixels, between children of a row or column major base.

YSIZE

The height of the base.

Y_SCROLL_SIZE

The height of the viewport if SCROLL is specified.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

In addition, you can use the [GET_VALUE](#) and [SET_VALUE](#) keywords to WIDGET_CONTROL to obtain or set the value of the button group. The values for different types of CW_BGROUP widgets is shown in the table below:

Type	Value
normal	None
exclusive	Index of currently set button
non-exclusive	Vector indicating the position of each button (1-set, 0-unset)

Table 3-12: Button Group Values

See [“Writing Compound Widgets”](#) in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by the `CW_BGROUP` Widget

Button Group widgets generates event structures with the following definition:

```
event = {ID:0L, TOP:0L, HANDLER:0L, SELECT:0, VALUE:0 }
```

The `SELECT` field is passed through from the button event. `VALUE` is either the `INDEX`, `ID`, `NAME`, or `BUTTON_UVALUE` of the button, depending on how the widget was created.

Version History

Introduced: Pre 4.0

`TAB_MODE` keyword added: 6.1

See Also

[CW_PDMENU](#), [WIDGET_BUTTON](#)

CW_CLR_INDEX

The CW_CLR_INDEX function creates a compound widget for the selection of a color index. A horizontal color bar is displayed. Clicking on the bar sets the color index.

This routine is written in the IDL language. Its source code can be found in the file `cw_clr_index.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_CLR_INDEX( Parent [, COLOR_VALUES=vector |  
    [, NCOLORS=value] [, START_COLOR=value]]  
    [, EVENT_FUNC='function_name'] [, /FRAME] [, LABEL=string]  
    [, NCOLORS=value] [, START_COLOR=value] [, TAB_MODE=value]  
    [, UNAME=string] [, UVALUE=value] [, VALUE=value] [, XSIZE=pixels]  
    [, YSIZE=pixels] )
```

Return Value

This function returns the widget ID of the newly-created color index widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

COLOR_VALUES

A vector of color indices containing the colors to be displayed in the color bar. If omitted, NCOLORS and START_COLOR specify the range of color indices.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget. This function is called with the return value structure whenever a button is pressed, and follows the conventions for user-written event functions.

FRAME

If set, a frame will be drawn around the widget.

LABEL

A text label that appears to the left of the color bar.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

NCOLORS

The number of colors to place in the color bar. The default is !D.N_COLORS.

START_COLOR

Set this keyword to the starting color index, placed at the left of the bar.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.

Table 3-13: TAB_MODE Keyword Options

Value	Description
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-13: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

VALUE

Set this keyword to the index of the color that is to be initially selected. The default is the `START_COLOR`.

XSIZE

The width of the color bar in pixels. The default is 192.

YSIZE

The height of the color bar in pixels. The default is 12.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

In addition, you can use the [GET_VALUE](#) and [SET_VALUE](#) keywords to WIDGET_CONTROL to obtain or set the value of the color selection widget. The value of a CW_CLR_INDEX widget is the index of the color selected.

See “[Writing Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using WIDGET_CONTROL and WIDGET_INFO.

Widget Events Returned by the CW_CLR_INDEX Widget

This widget generates event structures with the following definition:

```
Event = {CW_COLOR_INDEX, ID: base, TOP: ev.top, HANDLER: 0L,  
VALUE:c}
```

The VALUE field is the color index selected.

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[CW_COLORSSEL](#), [XLOADCT](#), [XPALETTE](#)

CW_COLORSEL

The `CW_COLORSEL` function creates a compound widget that displays all the colors in the current colormap in a 16 x 16 (320 x 320 pixels) grid. To select a color index, the user moves the mouse pointer over the desired color square and presses any mouse button. Alternatively, the color index can be selected by moving one of the three sliders provided around the grid.

This routine is written in the IDL language. Its source code can be found in the file `cw_colorsel.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_COLORSEL

The command:

```
WIDGET_CONTROL, widgetID, SET_VALUE = -1
```

informs the widget to initialize itself and redraw. It should be called when any of the following happen:

- the widget is realized,
- the widget needs redrawing,
- the brightest or darkest color has changed.

To set the current color index, use the command:

```
WIDGET_CONTROL, widgetID, SET_VALUE = index
```

To retrieve the current color index and store it in the variable `var`, use the command:

```
WIDGET_CONTROL, widgetID, GET_VALUE = var
```

Syntax

```
Result = CW_COLORSEL( Parent [, /FRAME]  
[, TAB_MODE=value][, UNAME=string] [, UVALUE=value]  
[, XOFFSET=value] [, YOFFSET=value] )
```

Return Value

This function returns the widget ID of the newly-created color index widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

FRAME

If set, a frame is drawn around the widget.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-14: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `FIND_BY_UNAME` keyword to the `WIDGET_INFO` function. The `UNAME` should be unique to the widget hierarchy; `FIND_BY_UNAME` returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

XOFFSET

The X offset position

YOFFSET

The Y offset position

Widget Events Returned by the CW_COLORSEL Widget

This widget generates event structures with the following definition:

```
Event = {COLORSEL_EVENT, ID: base, TOP: ev.top, HANDLER: 0L, VALUE:c}
```

The `VALUE` field is the color index selected.

Version History

Introduced: Pre 4.0

`TAB_MODE` keyword added: 6.1

See Also

[CW_CLR_INDEX](#), [XLOADCT](#), [XPALETTE](#)

CW_DEFROI

The CW_DEFROI function creates a compound widget that allows the user to define a region of interest within a widget draw window.

Warning

This is a *modal* widget. No other widget applications will be responsive while this widget is in use. Also, since CW_DEFROI has its own event-handling loop, it should not be created as a child of a modal base.

This routine is written in the IDL language. Its source code can be found in the file `cw_defroi.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_DEFROI( Draw [, IMAGE_SIZE=vector] [, OFFSET=vector]
  [, /ORDER] [, /RESTORE] [, TAB_MODE=value] [, ZOOM=vector] )
```

Return Value

The is function returns an array of subscripts defining the region. If no region is defined, the scalar -1 is returned.

Arguments

Draw

The widget ID of draw window in which to draw the region. Note that the draw window must have both `BUTTON` and `MOTION` events enabled (see [WIDGET_DRAW](#) for more information).

Keywords

IMAGE_SIZE

The size of the underlying array, expressed as a two element vector: `[columns, rows]`. Default is the size of the draw window divided by the value of `ZOOM`.

OFFSET

The offset of lower left corner of image within the draw window. Default = `[0,0]`.

ORDER

Set this keyword to return inverted subscripts, as if the array were output from top to bottom.

RESTORE

Set this keyword to restore the draw window to its previous appearance on exit. Otherwise, the regions remain on the drawable.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-15: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

ZOOM

If the image array was expanded (using REBIN, for example) specify this two element vector containing the expansion factor in X and Y. Default = [1,1]. Both elements of ZOOM must be integers.

Widget Events Returned by the CW_DEFROI Widget

Region definition widgets do not return an event structure.

Examples

The following two procedures create a region-of-interest widget and its event handler. Create a file containing the program code using a text editor and compile using the .RUN command, or type .RUN at the IDL prompt and enter the lines interactively.

First, create the event handler:

```

PRO test_event, ev

; The common block holds variables that are shared between the
; routine and its event handler:
COMMON T, draw, dbutt, done, image

; Define what happens when you click the "Draw ROI" button:
IF ev.id EQ dbutt THEN BEGIN
  ; The ROI definition will be stored in the variable Q:
  Q = CW_DEFROI(draw)
  IF (Q[0] NE -1) THEN BEGIN
    ; Show the size of the ROI definition array:
    HELP, Q
    ; Duplicate the original image.
    image2 = image

    ; Set the points in the ROI array Q equal to a single
    ; color value:
    image2(Q)=!P.COLOR-1

    ; Get the window ID of the draw widget:
    WIDGET_CONTROL, draw, GET_VALUE=W

    ; Set the draw widget as the current graphics window:
    WSET, W

    ; Load the image plus the ROI into the draw widget:
    TV, image2
  ENDIF

```

```

ENDIF

; Define what happens when you click the "Done" button:
IF ev.id EQ done THEN WIDGET_CONTROL, ev.top, /DESTROY

END

```

Next, create a draw widget that can call CW_DEFROI. Note that you *must* specify both button events and motion events when creating the draw widget, if it is to be used with CW_DEFROI.

```

PRO test
COMMON T, draw, dbutt, done, image

; Create a base to hold the draw widget and buttons:
base = WIDGET_BASE(/COLUMN)

; Create a draw widget that will return both button and
; motion events:
draw = WIDGET_DRAW(base, XSIZE=256, YSIZE=256, /BUTTON, /MOTION)
dbutt = WIDGET_BUTTON(base, VALUE='Draw ROI')
done = WIDGET_BUTTON(base, VALUE='Done')
WIDGET_CONTROL, base, /REALIZE

; Get the widget ID of the draw widget:
WIDGET_CONTROL, draw, GET_VALUE=W

; Set the draw widget as the current graphics window:
WSET, W

; Create an original image:
image = BYTSCL(SIN(DIST(256)))

; Display the image in the draw widget:
TV, image

; Start XMANAGER:
XMANAGER, "test", base

END

```

This results in the following:

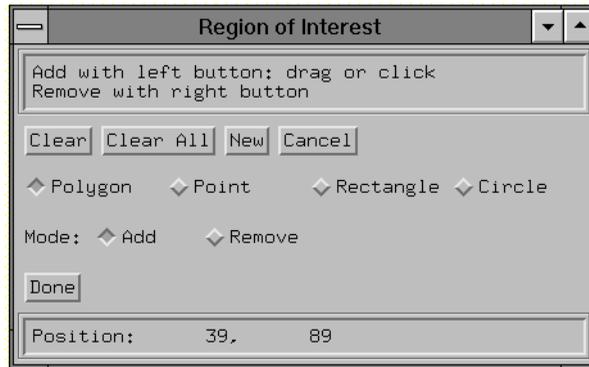


Figure 3-15: The Region of Interest Definition Widget

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[DEFROI](#)

CW_FIELD

The CW_FIELD function creates a widget data entry field. The field consists of a label and a text widget. CW_FIELD can create string, integer, or floating-point fields. The default is an editable string field.

This routine is written in the IDL language. Its source code can be found in the file `cw_field.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_FIELD( Parent [, /ALL_EVENTS] [, /COLUMN]
  [, /FIELDFONT=font] [, /FLOATING | , /INTEGER | , /LONG | , /STRING]
  [, /FONT=string] [, /FRAME=pixels] [, /NOEDIT] [, /RETURN_EVENTS]
  [, /ROW] [, /STRING=string] [, /TAB_MODE=value] [, /TEXT_FRAME=pixels]
  [, /TITLE=string] [, /UNAME=string] [, /UVALUE=value] [, /VALUE=value]
  [, /XSIZE=characters] [, /YSIZE=lines] )
```

Return Value

This function returns the widget ID of the newly-created field widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

ALL_EVENTS

Like RETURN_EVENTS but return an event whenever the contents of a text field have changed.

COLUMN

Set this keyword to center the label above the text field. The default is to position the label to the left of the text field.

FIELDFONT

A string containing the name of the font to use for the TEXT part of the field.

FLOATING

Set this keyword to have the field accept only floating-point values. Any number or string entered is converted to its floating-point equivalent.

FONT

A string containing the name of the font to use for the TITLE of the field. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

FRAME

The width, in pixels, of a frame to be drawn around the entire field cluster. The default is no frame.

INTEGER

Set this keyword to have the field accept only integer values. Any number or string entered is converted to its integer equivalent (using FIX). For example, if 12.5 is entered in this type of field, it is converted to 12.

LONG

Set this keyword to have the field accept only long integer values. Any number or string entered is converted to its long integer equivalent (using LONG).

NOEDIT

Normally, the value in the text field can be edited. Set this keyword to make the field non-editable.

RETURN_EVENTS

Set this keyword to make CW_FIELD return an event when a carriage return is pressed in a text field. The default is not to return events. Note that the value of the text field is always returned when the following command is used:

```
WIDGET_CONTROL, field, GET_VALUE = X
```

ROW

Set this keyword to position the label to the left of the text field. This is the default.

STRING

Set this keyword to have the field accept only string values. Numbers entered in the field are converted to their string equivalents. This is the default.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-16: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TEXT_FRAME

Set this keyword to the width in pixels of a frame to be drawn around the text field. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances. Under Microsoft Windows, text widgets always have a frame of width 1 pixel.

TITLE

A string containing the text to be used as the label for the field. The default is “Input Field”.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

VALUE

The initial value in the text widget. This value is automatically converted to the type set by the `STRING`, `INTEGER`, and `FLOATING` keywords described below.

XSIZE

An explicit horizontal size (in characters) for the text input area. The default is to let the window manager size the widget. Using the `XSIZE` keyword is not recommended.

YSIZE

An explicit vertical size (in lines) for the text input area. The default is 1.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

In addition, you can use the [GET_VALUE](#) and [SET_VALUE](#) keywords to WIDGET_CONTROL to obtain or set the value of the field. If one of the FLOATING, INTEGER, LONG, or STRING keywords to CW_FIELD is set, values set with the SET_VALUE keyword to WIDGET_CONTROL will be forced to the appropriate type. Values returned by the GET_VALUE keyword to WIDGET_CONTROL will be of the type specified when the field widget is created. Note that if the field contains string information, returned values will be contained in a string *array* even if the field contains only a single string.

See “[Writing Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using WIDGET_CONTROL and WIDGET_INFO.

Widget Events Returned by the CW_FIELD Widget

This widget generates event structures with the following definition:

```
event = { ID:0L, TOP:0L, HANDLER: 0L, VALUE:'', TYPE:0 , UPDATE:0 }
```

The VALUE field is the value of the field. TYPE specifies the type of data contained in the field and can be any of the following: 0=string, 1=float, 2=integer, 3=long integer (the value of TYPE is determined by setting one of the STRING, FLOAT, INTEGER, or LONG keywords). UPDATE contains a zero if the field has not been altered or a one if it has.

Examples

The code below creates a main base with a field cluster attached to it. The cluster accepts string input, has the title “Name”, and has a frame around it:

```
base = WIDGET_BASE()
field = CW_FIELD(base, TITLE = "Name", /FRAME)
WIDGET_CONTROL, base, /REALIZE
```

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[WIDGET_LABEL](#), [WIDGET_TEXT](#)

CW_FILESEL

The CW_FILESEL function is a compound widget for file selection.

This routine is written in the IDL language. Its source code can be found in the file `cw_filesel.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_FILESEL ( Parent [, /FILENAME] [, FILTER=string array]
                    [, /FIX_FILTER] [, /FRAME] [, /IMAGE_FILTER] [, /MULTIPLE | , /SAVE]
                    [, /PATH=string] [, /TAB_MODE=value] [, /UNAME=string] [, /UVALUE=value]
                    [, /WARN_EXIST] )
```

Return Value

This function returns the widget ID of the newly-created file selection widget.

Arguments

Parent

The widget ID of the parent.

Keywords

FILENAME

Set this keyword to have the initial filename filled in the filename text area.

FILTER

Set this keyword to an array of strings determining the filter types. If not set, the default is “All Files”. All files containing the chosen filter string will be displayed as possible selections. “All Files” is a special filter which returns all files in the current directory.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Example:

```
FILTER=["All Files", ".txt"]
```

Multiple filter types may be used per filter entry, using a comma as the separator.

Example:

```
FILTER=[".jpg, .jpeg", ".txt, .text"]
```

FIX_FILTER

If set, the user can not change the file filter.

FRAME

If set, a frame is drawn around the widget.

IMAGE_FILTER

If set, the filter “Image Files” will be added to the end of the list of filters. If set, and FILTER is not set, “Image Files” will be the only filter displayed. Valid image files are determined from QUERY_IMAGE.

MULTIPLE

If set, the file selection list will allow multiple filenames to be selected. The filename text area will not be editable in this case. It is illegal to specify both /SAVE and /MULTIPLE.

PATH

Set this keyword to the initial path the widget is to start in. The default is the current directory.

SAVE

Set this keyword to create a widget with a “Save” button instead of an “Open” button. It is illegal to specify both /SAVE and /MULTIPLE.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-17: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the WIDGET_INFO function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

WARN_EXIST

Set this keyword to produce a question dialog if the user selects a file that already exists. This keyword is useful when creating a “write” dialog. The default is to allow any filename to be quietly accepted, whether it exists or not.

Keywords to WIDGET_CONTROL

You can use the `GET_UVALUE` and `SET_UVALUE` keywords to `WIDGET_CONTROL` to obtain or set the user value of this widget. Use the command to read the currently selected filename (or filenames if `MULTIPLE` is set) including the full path:

```
WIDGET_CONTROL, id, GET_VALUE=filenames
```

To set the value of the filename, use the following command:

```
WIDGET_CONTROL, id, SET_VALUE=value
```

where *value* is a scalar string (or string array) containing the filenames, including the full path.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by CW_FILESEL

This widget generates event structures with the following definition:

```
Event = {FILESEL_EVENT, ID:0L, TOP:0L, HANDLER:0L, VALUE:'',
        DONE:0L, FILTER:''}
```

The `ID` field is the widget ID of the `CW_FILESEL` widget. The `TOP` field contains the widget ID of the top-level widget. The `HANDLER` field is always set to zero. The `VALUE` field is a string containing the most recent filename selected, if any.

Note

Even if `MULTIPLE` is set, `VALUE` will only contain the most recently selected filename. To retrieve all of the currently selected filenames, use the `GET_VALUE` keyword to `WIDGET_CONTROL`.

The `DONE` field can be any of the following:

- 0 = User selected a file but didn’t double-click, or the user changed filters (in this case the `VALUE` field will be an empty string.)

- 1 = User pressed “Open”/“Save” or double-clicked on a file.
- 2 = User pressed “Cancel”.

The FILTER field is a string containing the current filter.

Examples

This example creates a CW_FILESEL widget that is used to select image files for display. Note how the DONE tag of the event structure returned by CW_FILESEL is used to determine which button was pressed, and how the VALUE tag is used to obtain the file that was selected:

```

PRO image_opener_event, event

    WIDGET_CONTROL, event.top, GET_UVALUE=state, /NO_COPY

    CASE event.DONE OF
        0: BEGIN
            state.file = event.VALUE
            WIDGET_CONTROL, event.top, SET_UVALUE=state, /NO_COPY
        END
        1: BEGIN
            IF (state.file NE '') THEN BEGIN
                img = READ_IMAGE(state.file)
                TV, img
            ENDIF
            WIDGET_CONTROL, event.top, SET_UVALUE=state, /NO_COPY
        END
        2: WIDGET_CONTROL, event.top, /DESTROY
    ENDCASE

END

PRO image_opener

    DEVICE, DECOMPOSED=0, RETAIN=2

    base = WIDGET_BASE(TITLE = 'Open Image', /COLUMN)
    filesel = CW_FILESEL(base, /IMAGE_FILTER, FILTER='All Files')
    file=''
    state = {file:file}

    WIDGET_CONTROL, base, /REALIZE
    WIDGET_CONTROL, base, SET_UVALUE=state, /NO_COPY
    XMANAGER, 'image_opener', base

END

```

Version History

Introduced: 5.3

TAB_MODE keyword added: 6.1

See Also

[DIALOG_PICKFILE](#), [FILEPATH](#)

CW_FORM

The CW_FORM function is a compound widget that simplifies creating small forms which contain text, numeric fields, buttons, lists, and droplists. Event handling is also simplified.

This routine is written in the IDL language. Its source code can be found in the file `cw_form.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_FORM

The form has a value that is a structure with a tag/value pair for each field in the form. Use the command

```
WIDGET_CONTROL, id, GET_VALUE=v
```

to read the current value of the form. To set the value of one or more tags, use the command

```
WIDGET_CONTROL, id, SET_VALUE={ Tag:value, ..., Tag:value }
```

Syntax

```
Result = CW_FORM( [Parent,] Desc [, /COLUMN] [, IDS=variable]
  [, TAB_MODE=value] [, TITLE=string] [, UNAME=string]
  [, UVALUE=value] )
```

Return Value

If the argument *Parent* is present, the returned value of this function is the widget ID of the newly-created form widget. If *Parent* is omitted, the form realizes itself as a modal, top-level widget and CW_FORM returns a structure containing the value of each field in the form when the user finishes.

Arguments

Parent

The widget ID of the parent widget. Omit this argument to create a modal, top-level widget.

Desc

A string array describing the form. Each element of the string array contains two or more comma-delimited fields. Each string has the following format:

'Depth, Item, Initial value, Settings'

Use the backslash character (“\”) to escape commas that appear within fields. To include the backslash character, escape it with another backslash.

The fields are defined as follows:

- **Depth**

A digit defining the level at which the element will be placed on the form. Nesting is used primarily for layout, with row or column bases.

This field must contain the digit 0, 1, or 2, with the following effects:

- 0 = continue the current nesting level.
- 1 = begin a new level under the current level.
- 2 = last element at the current level.

- **Item**

A label defining the type of element to be placed in the form. *Item* must be one of the following: BASE, BUTTON, DROPLIST, FLOAT, INTEGER, LABEL, LIST, or TEXT.

BASEs and LABELs do not return a value in the widget value structure. The other items return the following value types:

Item	Description
BUTTON	An integer or integer array. For single buttons, the value is 1 if the button is set, or 0 if it is not set. For exclusive button groups, the value is the index of the currently set button. For non-exclusive button groups, the value is an array containing an element for each button. Array elements are set to 1 if the corresponding button is set, or 0 if it is not set.
DROPLIST	An integer. The value set in the widget value structure is the zero-based index of the item is selected.

Table 3-18: Values for the Item field

Item	Description
FLOAT	A floating-point value. The value set in the widget value structure is the floating-point value of the field.
INTEGER	An integer. The value set in the widget value structure is the integer value of the field.
LIST	An integer. The value set in the widget value structure is the zero-based index of the item is selected.
TEXT	A string. The value set in the widget value structure is the string value of the field.

Table 3-18: Values for the *Item* field

- **Initial value**

The initial value of the field. The *Initial value* field is left empty for BASEs.

For BUTTON, DROPLIST, and LIST items, the value field contains one or more item names, separated by the | character. Strings do not need to be enclosed in quotes. For example, the following line defines an exclusive button group with buttons labeled “one,” “two,” and “three.”

```
'0, BUTTON, one|two|three, EXCLUSIVE'
```

For FLOAT, INTEGER, LABEL, and TEXT items, the value field contains the initial value of the field.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

- **Settings**

The *Settings* field contains one of the following keywords or keyword=*value* pairs. Keywords are used to specify optional attributes or options. Any number of keywords may be included in the description string.

Note that preceding keywords with a “/” character has no effect. Simply including a keyword in the *Settings* field enables that option.

Keyword	Description
CENTER	Specifies alignment of LABEL items.
COLUMN	If present, specifies column layout in BASES or for BUTTON groups.
EXCLUSIVE	If present, makes an exclusive set of BUTTONs. The default is nonexclusive.
FONT= <i>font name</i>	If present, the font for the item is specified. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “ About Device Fonts ” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.
EVENT= <i>function</i>	Specifies the name of a user-written event function that is called whenever the element is changed. The event function is called with the widget event structure as a parameter. It may return an event structure or zero to indicate that no further event processing is desired.
FRAME	If present, a frame is drawn around the item. Valid only for BASEs.
LABEL_LEFT= <i>label</i>	Place a label to the left of the item. This keyword is valid with BUTTON, DROPLIST, FLOAT, INTEGER and TEXT items.
LABEL_TOP= <i>label</i>	Place a label above the item. This keyword is valid with BUTTON, DROPLIST, FLOAT, INTEGER and TEXT items.
LEFT	Specifies alignment of LABEL items.

Table 3-19: Values for the Settings Field

Keyword	Description
NO_RELEASE	If present, exclusive and non-exclusive buttons generate only select events. This keyword has no effect on regular buttons.
QUIT	If the form widget is created as a top-level, modal widget, when the user activates an item defined with this keyword, the form is destroyed and its widget value returned in the widget value structure of CW_FORM. For non-modal form widgets, events generated by changing this item have their QUIT field set to 1.
RIGHT	Specifies alignment of LABEL items.
ROW	If present, specifies row layout in BASES or for BUTTON groups.
SET_VALUE= <i>value</i>	Sets the initial value of BUTTON groups or DROPLISTS. For droplists and exclusive button groups, <i>value</i> should be the zero-based index of the item selected.
TAG= <i>name</i>	The tag name of this element in the widget's value structure. If not specified, the tag name is TAG <i>nnn</i> , where <i>nnn</i> is the zero-based index of the item in the <i>Desc</i> array.
WIDTH= <i>n</i>	Specifies the width, in characters, of a TEXT, INTEGER, or FLOAT item.

Table 3-19: Values for the Settings Field (Continued)

Keywords

COLUMN

Set this keyword to make the orientation of the form vertical. If COLUMN is not set, the form is laid out in a horizontal row.

IDS

Set this keyword equal to a named variable into which the widget id of each widget corresponding to an element in the *Desc* array is stored.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-20: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TITLE

Set this keyword equal to a scalar string containing the title of the top level base. TITLE is not used if the form widget has a parent widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

Set this keyword equal to the user value associated with the form widget.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget's base widget. This means that many keywords to the `WIDGET_CONTROL` and `WIDGET_INFO` routines that affect or return information on base widgets can be used with compound widgets.

In addition, you can use the [GET_VALUE](#) and [SET_VALUE](#) keywords to `WIDGET_CONTROL` to obtain or set the value of the form. The form has a value that is a structure with a tag/value pair for each field in the form. Use the command

```
WIDGET_CONTROL, id, GET_VALUE=v
```

to read the current value of the form. To set the value of one or more tags, use the command

```
WIDGET_CONTROL, id, SET_VALUE={ Tag:value, ..., Tag:value }
```

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by the CW_FORM Widget

This widget generates event structures each time the value of the form is changed. The event structure has the following definition:

```
Event = { ID:0L, TOP:0L, HANDLER:0L, TAG:'', VALUE:0, QUIT:0 }
```

The ID field is the widget ID of the `CW_FORM` widget. The TOP field is the widget ID of the top-level widget. The TAG field contains the tag name of the field that

changed. The VALUE field contains the new value of the changed field. The QUIT field contains a zero if the quit flag is not set, or one if it is set.

Examples

Define a form with a label, two groups of vertical buttons (one non-exclusive and the other exclusive), a text field, an integer field, and “OK” and “Done” buttons. If either the “OK” or “Done” buttons are pressed, the form exits.

Begin by defining a string array describing the form:

```
desc = [ $
  '0, LABEL, Centered Label, CENTER', $
  '1, BASE,, ROW, FRAME', $
  '0, BUTTON, B1|B2|B3, LABEL_TOP=Nonexclusive:', $
  + 'COLUMN, TAG=bg1', $
  '2, BUTTON, E1|E2|E2, EXCLUSIVE, LABEL_TOP=Exclusive:', $
  + 'COLUMN, TAG=bg2', $
  '0, TEXT, , LABEL_LEFT=Enter File name:, WIDTH=12,' $
  + 'TAG=fname', $
  '0, INTEGER, 0, LABEL_LEFT=File size:, WIDTH=6, TAG=fsize', $
  '1, BASE,, ROW', $
  '0, BUTTON, OK, QUIT,' $
  + 'TAG=OK', $
  '2, BUTTON, Cancel, QUIT']
```

To use the form as a modal widget:

```
a = CW_FORM(desc, /COLUMN)
```

When the form is exited, (when the user presses the OK or Cancel buttons), a structure is returned as the function’s value. We can examine the structure by entering:

```
HELP, /STRUCTURE, a
```

IDL Output			Meaning
BG1	INT	Array[3]	Set buttons = 1, unset = 0.
BG2	INT	1	Second button of exclusive button group was set.
FNAME	STRING	'test.dat'	Value of the text field
FSIZE	LONG	120	Value of the integer field

Table 3-21: Output from HELP, /STRUCTURE

IDL Output			Meaning
OK	LONG	1	This button was pressed
TAG8	LONG	0	This button wasn't pressed

Table 3-21: Output from HELP, /STRUCTURE

Note

If the “Cancel” button is pressed, the “OK” field is set to 0.

To use CW_FORM inside another widget:

```
a = WIDGET_BASE(TITLE='Testing')
b = CW_FORM(a, desc, /COLUMN)
WIDGET_CONTROL, a, /REALIZE
XMANAGER, 'Test', a
```

The event handling procedure (in this example, called TEST_EVENT), may use the TAG field of the event structure to determine which field changed and perform any data validation or special actions required. It can also get and set the value of the widget by calling WIDGET_CONTROL.

Version History

Introduced: 4.0

TAB_MODE keyword added: 6.1

CW_FSLIDER

The CW_FSLIDER function creates a slider that selects floating-point values.

This routine is written in the IDL language. Its source code can be found in the file `cw_fslider.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_FSLIDER

To get or set the value of a CW_FSLIDER widget, use the GET_VALUE and SET_VALUE keywords to WIDGET_CONTROL.

Note

The CW_FSLIDER widget is based on the WIDGET_SLIDER routine, which accepts only integer values. Because conversion between integers and floating-point numbers necessarily involves round-off errors, the slider value returned by CW_FSLIDER may not exactly match the input value, even when a floating-point number is entered in the slider's text field as an ASCII value. For more information on floating-point issues, see [“Accuracy & Floating-Point Operations”](#) in Chapter 22 of the *Using IDL* manual.

Syntax

```
Result = CW_FSLIDER( Parent [, /DOUBLE] [, /DRAG] [, /EDIT]
  [, FORMAT=string] [, /FRAME] [, MAXIMUM=value] [, MINIMUM=value]
  [, SCROLL=units] [, /SUPPRESS_VALUE] [, TAB_MODE=value]
  [, TITLE=string] [, UNAME=string] [, UVALUE=value]
  [, VALUE=initial_value] [, XSIZE=length | {, /VERTICAL [, YSIZE=height]})
```

Return Value

This function returns the widget ID of the newly-created slider widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

DOUBLE

Set this keyword to return double-precision values in the VALUE field of widget events generated by CW_FSLIDER. Explicitly set DOUBLE=0 to ensure that values returned in the VALUE field are single-precision. By default, CW_FSLIDER will return double-precision values if any of the values specified by the MINIMUM, MAXIMUM, or VALUE keywords is double-precision, or single-precision otherwise.

Note

The value returned by the GET_VALUE keyword to WIDGET_CONTROL is the value contained in the VALUE field of the widget event structure.

DRAG

Set this keyword to cause events to be generated continuously when the slider is adjusted. The default is DRAG=0, in which case events are generated only when the mouse is released. Note that on slow systems, /DRAG performance can be inadequate.

EDIT

Set this keyword to make the slider label editable. The default is EDIT=0. If EDIT is set, the GET_VALUE keyword to WIDGET_CONTROL will return the value of the slider label if it has been changed.

Note

If the user edits the slider label but does not press Enter, the slider position will not be updated to the new value. In this case, the widget programmer is responsible for using GET_VALUE to retrieve the new value of the slider label, followed by SET_VALUE to set the new value and update the slider position.

FORMAT

Provides the format in which the slider value is displayed. This should be a format as accepted by the STRING procedure. The default FORMAT is '(G13.6)'

FRAME

Set this keyword to have a frame drawn around the widget. The default is FRAME=0.

MAXIMUM

The maximum value of the slider. The default is `MAXIMUM=100`.

MINIMUM

The minimum value of the slider. The default is `MINIMUM=0`.

SCROLL

Set the `SCROLL` keyword to a floating-point value specifying the number of floating-point units the scroll bar should move when the user clicks the left mouse button inside the slider area (Motif) or on the slider arrows (Windows), but not on the slider itself. The default on both platforms is $0.01 \times (\text{MAXIMUM} - \text{MINIMUM})$, which is 1% of the slider range.

SUPPRESS_VALUE

If this keyword is set, the current slider value is not displayed.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.

Table 3-22: `TAB_MODE` Keyword Options

Value	Description
3	Navigate only off of the widget.

Table 3-22: *TAB_MODE* Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TITLE

Set this keyword to a string defining the title of the slider.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

VALUE

The initial value of the slider.

VERTICAL

If set, the slider will be oriented vertically. The default is horizontal.

XSIZE

The length of horizontal sliders.

YSIZE

The height of vertical sliders.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

You can use the [GET_VALUE](#) and [SET_VALUE](#) keywords to WIDGET_CONTROL to obtain or set the value of the slider. In addition, you can use the SET_VALUE keyword to change the minimum and maximum values of the slider by setting the keyword equal to a three-element vector [*value*, *min*, *max*].

Note

The SET_SLIDER_MAX and SET_SLIDER_MIN keywords to WIDGET_CONTROL and the SLIDER_MIN_MAX keyword to WIDGET_INFO *do not* work with floating point sliders created with CW_FSLIDER.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using WIDGET_CONTROL and WIDGET_INFO.

Widget Events Returned by the CW_FSLIDER Widget

This widget generates event structures with the following definition:

```
Event = { ID:0L, TOP:0L, HANDLER:0L, VALUE:0.0, DRAG:0 }
```

The VALUE field is the floating-point value selected by the slider. The DRAG field reports on whether events are generated continuously (when the DRAG keyword is set) or only when the mouse button is released (the default).

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[WIDGET_SLIDER](#)

CW_LIGHT_EDITOR

The CW_LIGHT_EDITOR function creates a compound widget to edit properties of existing IDLgrLight objects in a view. Lights cannot be added or removed from a view using this widget. However, lights can be “turned off or on” by hiding or showing them (i.e., HIDE property).

Syntax

```
Result = CW_LIGHT_EDITOR (Parent [, /DIRECTION_DISABLED]
    [, /DRAG_EVENTS] [, FRAME=width] [, /HIDE_DISABLED]
    [, LIGHT=objref(s)] [, /LOCATION_DISABLED] [, TAB_MODE=value]
    [, /TYPE_DISABLED] [, /UNAME=string] [, UVALUE=value]
    [, XSIZE=pixels] [, YSIZE=pixels] [, XRANGE=vector] [, YRANGE=vector]
    [, ZRANGE=vector] )
```

Return Value

This function returns the widget ID of a newly-created light editor.

Arguments

Parent

The widget ID of the parent widget for the new light editor.

Keywords

DIRECTION_DISABLED

Set this keyword to make the direction widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

DRAG_EVENTS

Set this keyword to cause events to be generated continuously while a slider in the compound widget is being dragged or when the mouse cursor is being dragged across the draw widget portion of the compound widget. By default, events are only generated when the mouse comes to rest at its final position and the mouse button is released.

When this keyword is set, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

Note

Under Microsoft Windows, sliders do not generate these events, but behave just like regular sliders.

FRAME

The value of this keyword specifies the width of a frame (in pixels) to be drawn around the borders of the widget. Note that this keyword is only a ‘hint’ to the toolkit, and may be ignored in some instances. The default is no frame.

HIDE_DISABLED

Set this keyword to make the hide widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

LIGHT

Set this keyword to one or more object references to `IDLgrLight` to edit. This will replace the current set of lights being edited with the list of lights from this keyword.

LOCATION_DISABLED

Set this keyword to make the location widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-23: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TYPE_DISABLED

Set this keyword to make the light type widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event. The default is to allow this property to be changed.

UNAME

Set this keyword to a string that can be used to identify the widget. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The ‘user value’ to be assigned to the widget. Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget

in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created. If UVALUE is not present, the widget's initial user value is undefined.

XRANGE

A two-element vector defining the data range in the x direction. This keyword is used to determine the valid range for the light's location and direction properties

XSIZE

The width of the drawable area in pixels. The default width is 180.

YRANGE

A two-element vector defining the data range in the y direction. This keyword is used to determine the valid range for the light's location and direction properties.

YSIZE

The height of the drawable area in pixels. The default height is 180.

ZRANGE

A two-element vector defining the data range in the z direction. This keyword is used to determine the valid range for the light's location and direction properties

WIDGET_CONTROL Keywords

The widget ID returned by this compound widget is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with this compound widget (e.g., UNAME, UVALUE).

GET_VALUE

Set this keyword to a named variable to contain the current value of the widget. An IDLgrLight object reference of the currently selected light is returned. The value of a widget can be set with the SET_VALUE keyword to this routine.

SET_VALUE

Sets the value of the specified light editor compound widget. This widget accepts an IDLgrLight object reference of the light in the list of lights to make as the current selection. The property values are retrieved from the light object and the light editor controls are updated to reflect those properties.

Widget Events Returned by the CW_LIGHT_EDITOR Widget

There are variations of the light editor event structure depending on the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER). The different light editor event structures are described below.

Light Selected

This is the type of structure returned when the light selected in the light list box is modified by a user.

```
{ CW_LIGHT_EDITOR_LS, ID:0L, TOP:0L, HANDLER:0L, LIGHT:OBJ_NEW() }
```

LIGHT specifies the object ID of the new light selection.

Light Modified

This is the type of structure returned when the user has modified a light property. This event maybe generated continuously if the DRAG_EVENTS keyword was set. See DRAG_EVENTS above.

```
{ CW_LIGHT_EDITOR_LM, ID:0L, TOP:0L, HANDLER:0L }
```

The value of the light editor will need to be retrieved (i.e., CW_LIGHT_EDITOR_GET) in order to determine the extent of the actual user modification.

Version History

Introduced: 5.3

TAB_MODE keyword added: 6.1

See Also

[CW_LIGHT_EDITOR_GET](#), [CW_LIGHT_EDITOR_SET](#), [IDLgrLight](#)

CW_LIGHT_EDITOR_GET

The CW_LIGHT_EDITOR_GET procedure gets the CW_LIGHT_EDITOR properties.

Syntax

```
CW_LIGHT_EDITOR_GET, WidgetID [, DIRECTION_DISABLED=variable]  
[, DRAG_EVENTS=variable] [, HIDE_DISABLED=variable]  
[, LIGHT=variable] [, LOCATION_DISABLED=variable]  
[, TYPE_DISABLED=variable] [, XSIZE=variable] [, YSIZE=variable]  
[, XRANGE=variable] [, YRANGE=variable] [, ZRANGE=variable]
```

Arguments

WidgetID

The widget ID of the CW_LIGHT_EDITOR compound widget.

Keywords

DIRECTION_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the direction widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

DRAG_EVENTS

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to cause events to be generated continuously while a slider in the compound widget is being dragged or when the mouse cursor is being dragged across the draw widget portion of the compound widget.

When this keyword is set, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

Note

Under Microsoft Windows, sliders do not generate these events, but behave just like regular sliders.

HIDE_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the hide widget portion of the compound widget unchangeable by the user.

LIGHT

Set this keyword to a named variable that will contain one or more object references to IDLgrLight.

LOCATION_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the location widget portion of the compound widget unchangeable by the user.

TYPE_DISABLED

Set this keyword to a named variable that will contain a boolean value indicating whether this option has been set to make the light type widget portion of the compound widget unchangeable by the user.

XRANGE

Set this keyword to a named variable that will contain a two-element vector defining the data range in the x direction.

XSIZE

Set this keyword to a named variable that will contain the width of the drawable area in pixels.

YRANGE

Set this keyword to a named variable that will contain a two-element vector defining the data range in the y direction.

YSIZE

Set this keyword to a named variable that will contain the height of the drawable area in pixels.

ZRANGE

Set this keyword to a named variable that will contain a two-element vector defining the data range in the z direction.

Version History

Introduced: 5.3

See Also

[CW_LIGHT_EDITOR](#), [CW_LIGHT_EDITOR_SET](#), [IDLgrLight](#)

CW_LIGHT_EDITOR_SET

The CW_LIGHT_EDITOR procedure sets the CW_LIGHT_EDITOR properties.

Syntax

```
CW_LIGHT_EDITOR_SET, WidgetID [, /DIRECTION_DISABLED]
  [, /DRAG_EVENTS] [, /HIDE_DISABLED] [, LIGHT=objref(s)]
  [, /LOCATION_DISABLED] [, /TYPE_DISABLED] [, XSIZE=pixels]
  [, YSIZE=pixels] [, XRANGE=vector] [, YRANGE=vector] [, ZRANGE=vector]
```

Arguments

WidgetID

The widget ID of the CW_LIGHT_EDITOR compound widget.

Keywords

DIRECTION_DISABLED

Set this keyword to make the direction widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

DRAG_EVENTS

Set this keyword to cause events to be generated continuously while a slider in the compound widget is being dragged or when the mouse cursor is being dragged across the draw widget portion of the compound widget.

When this keyword is set, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

Note

Under Microsoft Windows, sliders do not generate these events, but behave just like regular sliders.

HIDE_DISABLED

Set this keyword to make the hide widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

LIGHT

Set this keyword to one or more object references to `IDLgrLight` to edit. This will replace the current set of lights being edited with the list of lights from this keyword.

LOCATION_DISABLED

Set this keyword to make the location widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

TYPE_DISABLED

Set this keyword to make the light type widget portion of the compound widget unchangeable by the user. It will appear insensitive and will not generate an event.

XRANGE

A two-element vector defining the data range in the x direction. This keyword is used to determine the valid range for the light's location and direction properties.

XSIZE

The width of the drawable area in pixels.

YRANGE

A two-element vector defining the data range in the y direction. This keyword is used to determine the valid range for the light's location and direction properties.

YSIZE

The height of the drawable area in pixels.

ZRANGE

A two-element vector defining the data range in the z direction. This keyword is used to determine the valid range for the light's location and direction properties.

Version History

Introduced: 5.3

See Also

[CW_LIGHT_EDITOR](#), [CW_LIGHT_EDITOR_GET](#), [IDLgrLight](#)

CW_ORIENT

The CW_ORIENT function creates a compound widget that provides a means to interactively adjust the three-dimensional drawing transformation and resets the !P.T system variable field to reflect the changed orientation.

This routine is written in the IDL language. Its source code can be found in the file `cw_orient.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_ORIENT( Parent [, AX=degrees] [, AZ=degrees] [, /FRAME]
  [, TAB_MODE=value] [, TITLE=string] [, UNAME=string] [, UVALUE=value]
  [, XSIZE=width] [, YSIZE=height] )
```

Return Value

This function returns the widget ID of the newly-created orientation-adjustment widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

AX

The initial rotation in the X direction. The default is 30 degrees.

AZ

The initial rotation in the Z direction. The default is 30 degrees.

FRAME

Set this keyword to draw a frame around the widget.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-24: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TITLE

The title of the widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

XSIZE

Determines the width of the widget. The default is 100.

YSIZE

Determines the height of the widget. The default is 100.

Keywords to `WIDGET_CONTROL` and `WIDGET_INFO`

The widget ID returned by most compound widgets is actually the ID of the compound widget’s base widget. This means that many keywords to the `WIDGET_CONTROL` and `WIDGET_INFO` routines that affect or return information on base widgets can be used with compound widgets.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by the `CW_ORIENT` Widget

`CW_ORIENT` only returns events when the three dimensional drawing transformation has been altered. The `!P.T` system variable field is automatically updated to reflect the new orientation.

Version History

Introduced: Pre 4.0

`TAB_MODE` keyword added: 6.1

See Also

[CW_ARCBALL](#), [T3D](#)

CW_PALETTE_EDITOR

The CW_PALETTE_EDITOR function creates a compound widget to display and edit color palettes. The palette editor is a base that contains a drawable area to display the color palette, a set of vectors that represent the palette and an optional histogram.

Graphics Area Components

Reference Color bar

A gray scale color bar is displayed at the top of the graphics area for reference purposes.

Palette Colorbar

A color bar containing a display of the current palette is displayed below the reference color bar.

Channel and Histogram Display

The palette channel vectors are displayed below the palette colorbar. The Red channel is displayed in red, the Green channel in green, the Blue channel in blue, and the optional Alpha channel in purple. The optional Histogram vector is displayed in Cyan.

An area with a white background represents the current selection, with gray background representing the area outside of the current selection. Yellow drag handles are an additional indicator of the selection endpoints. These selection endpoints represent the range for some editing operations. In addition, cursor X,Y values and channel pixel values at the cursor location are displayed in a status area below the graphics area.

Interactive Capabilities

Color Space

A droplist allows selection of RGB, HSV or HLS color spaces. RGB is the default color space.

Note

Regardless of the color space in use, the color vectors retrieved with the GET_VALUE keyword to widget control are always in the RGB color space.

Editing Mode

A droplist allows selection of the editing mode. Freehand is the default editing mode.

Note

Unless noted below, editing operations apply only to the channel vectors currently selected for editing and apply only to the portion of the vectors within the selection indicators.

Editing Mode	Description
Freehand	The user can click and drag in the graphics area to draw a new curve. Editable channel vectors will be modified to use the new curve for that part of the X range within the selection that was drawn in Freehand mode.
Line Segment	A click, drag and release operation defines the start point and end point of a line segment. Editable channel vectors will be modified to use the new curve for that part of the X range within the selection that was drawn in Line Segment mode.
Barrel Shift	Click and drag operations in the horizontal direction cause the editable curves to be shifted right or left, with the portion which is shifted off the end of selection area wrapping around to appear on the other side of the selection area. Only the horizontal component of drag movement is used.
Slide	Click and drag operations in the horizontal direction cause the editable curves to be shifted right or left. Unlike the Barrel Shift mode, the portion of the curves shifted off the end of the selection area does not wrap around. Only the horizontal component of drag movement is used.
Stretch	Click and drag operations in the horizontal direction cause the editable curves to be compressed or expanded. Only the horizontal component of drag movement is used.

Table 3-25: CW_PALETTE_EDITOR Editing Mode Options

The following table describes the buttons that provide editing operations but do not require cursor input:

Button	Operation
Ramp	Causes the selected part of the editable curves to be replaced with a linear ramp from 0 to 255.
Smooth	Causes the selected part of the editable curves to be smoothed.
Posterize	Causes the selected part of the editable curves to be replaced with a series of steps.
Reverse	Causes the selected part of the editable curves to be reversed in the horizontal direction.
Invert	Causes the selected part of the editable curves to be flipped in the vertical direction.
Duplicate	Causes the selected part of the editable curves to be compressed by 50% and duplicated to produce two contiguous copies of the channel vectors within the initial selection.
Load PreDefined	Leads to additional choices of pre-defined palettes. Loading a pre-defined palette replaces only the selected portion of the editable color channels, respecting of the settings of the selection endpoints and editable checkboxes. This allows loading only a single channel or only a portion of a pre-defined palette.

Table 3-26: CW_PALETTE_EDITOR Button Operations

Channel Display and Edit

A row of checkboxes allows the user to indicate which channels of Red, Green, Blue and the optional Alpha channel should be displayed. A second row of checkboxes allows the user to indicate which channels should be edited by the current editing operation. The checkboxes for the Alpha channel will be sensitive only if an Alpha channel is loaded.

Zoom

Four buttons allow the user to zoom the display of the palette:

Button	Description
	Zooms to show the current selection.
+	Zooms in 50%
-	Zooms out 100%
1:1	Returns the display to the full palette

Table 3-27: Palette Zoom Options

Scrolling of the Palette Window

When the palette is zoomed to a scale greater than 1:1 the scroll bar at the bottom of the graphics area can be used to view a different part of the palette.

Syntax

```
Result = CW_PALETTE_EDITOR (Parent [, DATA=array] [, FRAME=width]
    [, HISTOGRAM=vector] [, /HORIZONTAL] [, SELECTION=[start, end]]
    [, TAB_MODE=value] [, UNAME=string] [, UVALUE=value] [, XSIZE=width]
    [, YSIZE=height] )
```

Return Value

This function returns the widget ID of the newly created palette editor.

Arguments

Parent

The widget ID of the parent widget for the new palette editor.

Keywords

DATA

A 3x256 byte array containing the initial color values for Red, Green and Blue channels. The value supplied can also be a 4x256 byte array containing the initial

color values and the optional Alpha channel. The value supplied can also be an IDLgrPalette object reference. If an IDLgrPalette object reference is supplied it is used internally and is not destroyed on exit. If an object reference is supplied the ALPHA keyword to the CW_PALETTE_EDITOR_SET routine can be used to supply the data for the optional Alpha channel.

FRAME

The value of this keyword specifies the width of a frame (in pixels) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances. The default is no frame.

HISTOGRAM

A 256 element byte vector containing the values for the optional histogram curve.

HORIZONTAL

Set this keyword for a horizontal layout for the compound widget. This consists of the controls to the right of the display area. The default is a vertical layout with the controls below the display area.

SELECTION

The selection is a two element vector defining the starting and ending point of the selection region of color indexes. The default is [0,255].

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-28: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The ‘user value’ to be assigned to the widget. Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created. If `UVALUE` is not present, the widget's initial user value is undefined.

XSIZE

The width of the drawable area in pixels. The default width is 256.

YSIZE

The height of the drawable area in pixels. The default height is 256.

WIDGET_CONTROL Keywords for Palette Editor

The widget ID returned by this compound widget is actually the ID of the compound widget's base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with this compound widget (e.g., UNAME, UVALUE).

GET_VALUE

Set this keyword to a named variable to contain the current value of the widget. A 3xn (RGB) or 4xn (i.e., RGB and ALPHA) array containing the palette is returned.

The value of a widget can be set with the SET_VALUE keyword to this routine.

SET_VALUE

Sets the value of the specified palette editor compound widget. This widget accepts a 3xn (RGB) or 4xn (i.e., RGB and ALPHA) array representing the value of the palette to be set. Another type of argument accepted is an IDLgrPalette object reference. If an IDLgrPalette object reference is supplied it is used internally and is not destroyed on exit.

Widget Events Returned by the CW_PALETTE_EDITOR Widget

There are variations of the palette editor event structure depending on the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER). The different palette editor event structures are described below.

Selection Moved

This is the type of structure returned when one of the vertical bars that define the selection region is moved by a user.

```
{ CW_PALETTE_EDITOR_SM, ID:0L, TOP:0L, HANDLER:0L, SELECTION:[0,255]}
```

SELECTION indicates a two element vector defining the starting and ending point of the selection region of color indexes.

Palette Edited

This is the type of structure returned when the user has modified the color palette.

```
{ CW_PALETTE_EDITOR_PM, ID:0L, TOP:0L, HANDLER:0L }
```

The value of the palette editor will need to be retrieved (i.e., WIDGET_CONTROL, GET_VALUE) in order to determine the extent of the actual user modification.

Version History

Introduced: 5.3

TAB_MODE keyword added: 6.1

See Also

[CW_PALETTE_EDITOR_GET](#), [CW_PALETTE_EDITOR_SET](#), [IDLgrPalette](#)

CW_PALETTE_EDITOR_GET

The CW_PALETTE_EDITOR_GET procedure gets the CW_PALETTE_EDITOR properties.

Syntax

```
CW_PALETTE_EDITOR_GET, WidgetID [, ALPHA=variable]  
[, HISTOGRAM=variable]
```

Arguments

WidgetID

The widget ID of the CW_PALETTE_EDITOR compound widget.

Keywords

ALPHA

Set this keyword to a named variable that will contains the optional alpha curve.

HISTOGRAM

Set this keyword to a named variable that will contains the optional histogram curve.

Version History

Introduced: 5.3

See Also

[CW_PALETTE_EDITOR](#), [CW_PALETTE_EDITOR_SET](#), [IDLgrPalette](#)

CW_PALETTE_EDITOR_SET

The CW_PALETTE_EDITOR_SET procedure sets the CW_PALETTE_EDITOR properties.

Syntax

```
CW_PALETTE_EDITOR_SET, WidgetID [, ALPHA=byte_vector]  
[, HISTOGRAM=byte_vector]
```

Arguments

WidgetID

The widget ID of the CW_PALETTE_EDITOR compound widget.

Keywords

ALPHA

A 256 element byte vector that describes the alpha component of the color palette. The alpha value may also be set to the scalar value zero to remove the alpha curve from the display.

HISTOGRAM

The histogram is an vector to be plotted below the color palette. This keyword can be used to display a distribution of color index values to facilitate editing the color palette. The histogram value may also be set to the scalar value zero to remove the histogram curve from the display.

Version History

Introduced: 5.3

See Also

[CW_PALETTE_EDITOR](#), [CW_PALETTE_EDITOR_GET](#), [IDLgrPalette](#)

CW_PDMENU

The CW_PDMENU function creates widget pulldown menus. It has a simpler interface than the XPDMENU procedure, which it replaces. Events for the individual buttons are handled transparently, and a CW_PDMENU event returned. This event can return any one of the following:

- the Index of the button within the base
- the widget ID of the button
- the name of the button.
- the fully qualified name of the button. This allows different sub-menus to contain buttons with the same name in an unambiguous way.

Only buttons with textual names are handled by this widget. Bitmaps are not understood.

This routine is written in the IDL language. Its source code can be found in the file `cw_pdmenu.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_PDMENU( Parent, Desc [, /COLUMN] [, /CONTEXT_MENU]
  [, DELIMITER=string] [, FONT=value] [, /MBAR [, /HELP]] [, IDS=variable]
  [, /RETURN_ID | , /RETURN_INDEX | , /RETURN_NAME | ,
  /RETURN_FULL_NAME] [, TAB_MODE=value] [, UNAME=string]
  [, UVALUE=value] [, XOFFSET=value] [, YOFFSET=value] )
```

Return Value

This function returns the widget ID of the newly-created pulldown menu widget.

Arguments

Parent

The widget ID of the parent widget.

Desc

An array of strings or structures. If *Desc* is an array of strings, each element contains the flag field, followed by a backslash character, followed by the name of the menu

item, optionally followed by another backslash character and the name of an event-processing procedure for that element. A string element of the *Desc* array would look like:

```
'n\item_name'
```

or

```
'n\item_name\event_proc'
```

where *n* is the flag field and *item_name* is the name of the menu item. The flag field is a bitmask that controls how the button is interpreted; appropriate values for the flag field are shown in the following table. If the *event_proc* field is present, it is the name of an event-handling procedure for the menu element and all of its children.

If *Desc* is an array of structures, each structure must have the following definition:

```
{CW_PDMENU_S, flags:0, name:''}
```

The name tag is a string field with the following components:

```
'item_name'
```

or

```
'item_name\event_proc'
```

where *item_name* is the name of the menu item. If the *event_proc* field is present, it is the name of an event-handling procedure for the menu element and all of its children

The flags field is a bitmask that controls how the button is interpreted; appropriate values for the flag field are shown in the following table. Note that if *Desc* is an array of structures, you cannot specify individual event-handling procedures for each element.

Value	Meaning
0	This button is neither the beginning nor the end of a pulldown level.
1	This button is the root of a sub-pulldown menu. The sub-buttons start with the next button.
2	This button is the last button at the current pulldown level. The next button belongs to the same level as the current parent button. If the name field is not specified (or is an empty string), no button is created, and the next button is created one level up in the hierarchy.

Table 3-29: Button Flag Bit Meanings

Value	Meaning
3	This button is the root of a sub-pulldown menu, but it is also the last entry of the current level.

Table 3-29: Button Flag Bit Meanings (Continued)

Keywords

COLUMN

Set this keyword to create a vertical column of menu buttons. The default is to create a horizontal row of buttons.

CONTEXT_MENU

Set this keyword to create the pulldown menu within a context-sensitive menu. If `CONTEXT_MENU` is set, *Parent* must be the widget ID of a base widget, which also has the `CONTEXT_MENU` keyword set. The return value of `CW_PDMENU` is the *Parent's* widget ID. If `CONTEXT_MENU` is set, setting the `UNAME` or `UVALUE` on the same call to `CW_PDMENU` will override the `UNAME` or `UVALUE` values of the parent base.

For more on creating context menus, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual and the `CONTEXT_MENU` keyword to [WIDGET_BASE](#).

DELIMITER

The character used to separate the parts of a fully qualified name in returned events. The default is to use the “.” character.

FONT

The name of the font to be used for the button titles. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

HELP

If the `MBAR` keyword is set, and one of the buttons on the menubar has the label “help” (case insensitive) then that button is created with the `/HELP` keyword to give it

any special appearance it is supposed to have on a menubar. For example, Motif expects help buttons to be on the right.

IDS

A named variable in which the button IDs will be stored as a longword vector.

MBAR

Set this keyword to create a menubar pulldown. If MBAR is set, *Parent* must be the widget ID of a menubar belonging to a top-level base, and the return value of CW_PDMENU is the *Parent* widget ID. For an example demonstrating the use of the MBAR keyword, see [Example 2](#) below. Also see the [MBAR](#) keyword to WIDGET_BASE. If MBAR is set, setting the UNAME or UVALUE on the same call to CW_PDMENU will override the UNAME or UVALUE values of the parent base.

RETURN_ID

If this keyword is set, the VALUE field of returned events will contain the widget ID of the button.

RETURN_INDEX

If this keyword is set, the VALUE field of returned events will contain the zero-based index of the button within the base. THIS IS THE DEFAULT.

RETURN_NAME

If this keyword is set, the VALUE field of returned events will be the name of the selected button.

RETURN_FULL_NAME

Set this keyword and the VALUE field of returned events will be the fully qualified name of the selected button. This means that the names of all the buttons from the topmost button of the pulldown menu to the selected one are concatenated with the delimiter specified by the DELIMITER keyword. For example, if the top button was named COLORS, the second level button was named BLUE, and the selected button was named LIGHT, the returned value would be

```
COLORS.BLUE.LIGHT
```

This allows different submenus to have buttons with the same name (e.g., COLORS.RED.LIGHT).

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-30: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget. If the `MBAR` keyword is set, the value specified for `UVALUE` is also assigned as the `UVALUE` of the parent menu (i.e., the widget specified by the *Parent* argument in the call to `CW_PDMENU`).

XOFFSET

The X offset of the widget relative to its parent.

YOFFSET

The Y offset of the widget relative to its parent.

Keywords to `WIDGET_CONTROL` and `WIDGET_INFO`

The widget ID returned by most compound widgets is actually the ID of the compound widget’s base widget. This means that many keywords to the `WIDGET_CONTROL` and `WIDGET_INFO` routines that affect or return information on base widgets can be used with compound widgets.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by the `CW_PDMENU` Widget

This widget generates event structures with the following definition:

```
event = { ID:0L, TOP:0L, HANDLER:0L, VALUE:0 }
```

`VALUE` is either the `INDEX`, `ID`, `NAME`, or `FULL_NAME` of the button, depending on how the widget was created.

Examples

Example 1

The following is the description of a menu bar with two buttons: “Colors” and “Quit”. Colors is a pulldown containing the colors “Red”, “Green”, “Blue”, “Cyan”, and “Magenta”. Blue is a sub-pulldown containing “Light”, “Medium”, “Dark”, “Navy”, and “Royal.”

The following small program can be used with the above description to create the specified menu:

```

PRO PD_EXAMPLE
  desc = [ '1\Colors' , $
          '0\Red' , $
          '0\Green' , $
          '1\Blue' , $
          '0\Light' , $
          '0\Medium' , $
          '0\Dark' , $
          '0\Navy' , $
          '2\Royal' , $
          '0\Cyan' , $
          '2\Magenta' , $
          '2\Quit' ]

  ; Create the widget:
  base = WIDGET_BASE()
  menu = CW_PDMENU(base, desc, /RETURN_FULL_NAME)
  WIDGET_CONTROL, /REALIZE, base

  ;Provide a simple event handler:
  REPEAT BEGIN
    ev = WIDGET_EVENT(base)
    PRINT, ev.value
  END UNTIL ev.value EQ 'Quit'
  WIDGET_CONTROL, /DESTROY, base

END

```

The *Desc* array could also have been defined using a structure for each element. The following array of structures creates the same menu as the array of strings shown above. Note, however, that if the *Desc* array is composed of structures, you cannot specify individual event-handling routines.

First, make sure *CW_PDMENU_S* structure is defined:

```
junk = {CW_PDMENU_S, flags:0, name:'' }
```

Define the menu:

```

desc = [ { CW_PDMENU_S, 1, 'Colors' }, $
        { CW_PDMENU_S, 0, 'Red' }, $
        { CW_PDMENU_S, 0, 'Green' }, $
        { CW_PDMENU_S, 1, 'Blue' }, $
        { CW_PDMENU_S, 0, 'Light' }, $
        { CW_PDMENU_S, 0, 'Medium' }, $
        { CW_PDMENU_S, 0, 'Dark' }, $
        { CW_PDMENU_S, 0, 'Navy' }, $
        { CW_PDMENU_S, 2, 'Royal' }, $
        { CW_PDMENU_S, 0, 'Cyan' }, $
        { CW_PDMENU_S, 2, 'Magenta' }, $

```

```
{ CW_PDMENU_S, 2, 'Quit' } ]
```

Example 2

This example demonstrates the use of the MBAR keyword to CW_PDMENU to populate the “Colors” menu item on a menu bar created using WIDGET_BASE.

```
PRO mbar_event, event

    WIDGET_CONTROL, event.id, GET_UVALUE=uval

    CASE uval OF
        'Quit': WIDGET_CONTROL, /DESTROY, event.top
    ELSE: PRINT, event.value
    ENDCASE

END

PRO mbar

    ; Create the base widget:
    base = WIDGET_BASE(TITLE = 'Example', MBAR=bar, XSIZE=200, $
        UVALUE='base')

    file_menu = WIDGET_BUTTON(bar, VALUE='File', /MENU)
    file_bttn1=WIDGET_BUTTON(file_menu, VALUE='Quit', $
        UVALUE='Quit')

    colors_menu = WIDGET_BUTTON(bar, VALUE='Colors', /MENU)

    ; Define array for colors menu items:
    desc = [ '0\Red' , $
        '0\Green' , $
        '1\Blue' , $
        '0\Light' , $
        '0\Medium' , $
        '0\Dark' , $
        '0\Navy' , $
        '2\Royal' , $
        '0\Cyan' , $
        '2\Magenta' ]

    ; Create colors menu items. Note that the Parent argument is
    ; set to the widget ID of the parent menu:
    colors = CW_PDMENU(colors_menu, desc, /MBAR, $
        /RETURN_FULL_NAME, UVALUE='menu')

    WIDGET_CONTROL, /REALIZE, base
```

```
XMANAGER, 'mbar', base, /NO_BLOCK
```

```
END
```

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[CW_BGROUP](#), [WIDGET_DROPLIST](#)

CW_RGBSLIDER

The CW_RGBSLIDER function creates a compound widget that provides three sliders for adjusting color values. The RGB, CMY, HSV, and HLS color systems can all be used. No matter which color system is in use, the resulting color is always supplied in RGB, which is the base system for IDL.

This routine is written in the IDL language. Its source code can be found in the file `cw_rgbslider.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_RGBSLIDER

The CW_RGBSLIDER widget consists of a pulldown menu which allows the user to change between the supported color systems, and three color adjustment sliders, allowing the user to select a new color value.

Syntax

```
Result = CW_RGBSLIDER( Parent [, /CMY | , /HSV | , /HLS | , /RGB]
    [, /COLOR_INDEX | , GRAPHICS_LEVEL={1 | 2}] [, /DRAG] [, /FRAME]
    [, LENGTH=value] [, /RGB] [, TAB_MODE=value] [, UNAME=string]
    [, UVALUE=value] [, VALUE=[r, g, b]] [, /VERTICAL] )
```

Return Value

This function returns the widget ID of the newly-created color adjustment widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

CMY

If set, the initial color system used is CMY.

COLOR_INDEX

Set this keyword to display a small rectangle with the selected color. The color is updated as the values are changed. The color initially displayed in this rectangle corresponds to the value specified with the VALUE keyword. If using Object Graphics, it is recommended that you set the GRAPHICS_LEVEL keyword to 2, in which case the COLOR_INDEX keyword is ignored.

DRAG

Set this keyword and events will be generated continuously when the sliders are adjusted. If not set, events will only be generated when the mouse button is released. Note: On slow systems, /DRAG performance can be inadequate. The default is DRAG = 0.

FRAME

If set, a frame will be drawn around the widget. The default is FRAME = 0.

GRAPHICS_LEVEL

Set this keyword to 2 to use Object Graphics. Set to 1 for Direct Graphics (the default). If set to 2, a small rectangle is displayed with the selected color. The color is updated as the values are changed. The color initially displayed in this rectangle corresponds to the value specified with the VALUE keyword. If this keyword is set, the COLOR_INDEX keyword is ignored.

HSV

If set, the initial color system used is HSV.

HLS

If set, the initial color system used is HLS.

LENGTH

The length of the sliders. The default = 256.

RGB

If set, the initial color system used is RGB. This is the default.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-31: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

VALUE

Set this keyword to a 3-element $[r, g, b]$ vector representing the initial RGB value for the CW_RGBSLIDER widget. If the GRAPHICS_LEVEL keyword is set to 2, the color swatch will also initially display this RGB value.

VERTICAL

If set, the sliders will be oriented vertically. The default is VERTICAL = 0.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget’s base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using WIDGET_CONTROL and WIDGET_INFO.

Widget Events Returned by the CW_RGBSLIDER Widget

This widget generates event structures with the following definition:

```
event = {ID:0L, TOP:0L, HANDLER:0L, R:0B, G:0B, B:0B }
```

The ‘R’, ‘G’, and ‘B’ fields contain the Red, Green and Blue components of the selected color. Note that CW_RGBSLIDER reports back the Red, Green, and Blue values no matter which color system is selected.

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[CW_CLR_INDEX](#), [XLOADCT](#), [XPALETTE](#)

CW_TMPL

The CW_TMPL procedure is a template for compound widgets that use the XMANAGER. Use this template instead of writing your compound widgets from “scratch”. This template can be found in the file `cw_tmpl.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = CW_TMPL( Parent [, TAB_MODE=value] [, UNAME=string]
                [, UVALUE=value] )
```

Arguments

Parent

The widget ID of the parent widget of the new compound widget.

Keywords

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.

Table 3-32: TAB_MODE Keyword Options

Value	Description
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-32: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UVALUE

A user-specified value for the compound widget.

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[XMNG_TMPL](#)

CW_ZOOM

The `CW_ZOOM` function creates a compound widget that displays two images: an original image in one window and a portion of the original image in another. The user can select the center of the zoom region, the zoom scale, the interpolation style, and the method of indicating the zoom center.

This routine is written in the IDL language. Its source code can be found in the file `cw_zoom.pro` in the `lib` subdirectory of the IDL distribution.

Using CW_ZOOM

The value of the `CW_ZOOM` widget is the original, un-zoomed image to be displayed (a two-dimensional array). To change the contents of the `CW_ZOOM` widget, use the command:

```
WIDGET_CONTROL, id, SET_VALUE = array
```

where `id` is the widget ID of the `CW_ZOOM` widget and `array` is the image array. The value of `CW_ZOOM` cannot be set until the widget has been realized. Note that the size of the original window, set with the `XSIZE` and `YSIZE` keywords to `CW_ZOOM`, must be the size of the input array.

To return the current zoomed image as displayed by `CW_ZOOM` in the variable `array`, use the command:

```
WIDGET_CONTROL, id, GET_VALUE = array
```

Syntax

```
Result = CW_ZOOM( Parent [, /FRAME] [, MAX=scale] [, MIN=scale]
  [, RETAIN={0 | 1 | 2}] [, SAMPLE=value] [, SCALE=value]
  [, TAB_MODE=value] [, /TRACK] [, UNAME=string] [, UVALUE=value]
  [, XSIZE=width] [, X_SCROLL_SIZE=width] [, X_ZSIZE=zoom_width]
  [, YSIZE=height] [, Y_SCROLL_SIZE=height] [, Y_ZSIZE=zoom_height] )
```

Return Value

This function returns the widget ID of the newly-created zoom widget.

Arguments

Parent

The widget ID of the parent widget.

Keywords

FRAME

If set, a frame will be drawn around the widget. The default is `FRAME = 0`.

MAX

The maximum zoom scale, which must be greater than or equal to 1. The default is 20.

MIN

The minimum zoom scale, which must be greater than or equal to 1. The default is 1.

RETAIN

Set this keyword to zero, one, or two to specify how backing store should be handled for both windows. `RETAIN=0` specifies no backing store. `RETAIN=1` requests that the server or window system provide backing store. `RETAIN=2` specifies that IDL provide backing store directly. See [“Backing Store”](#) on page 4286 for details.

SAMPLE

Set to zero for bilinear interpolation, or to a non-zero value for nearest neighbor interpolation. Bilinear interpolation gives higher quality results, but requires more time. The default is 0.

SCALE

The initial integer scale factor to use for the zoomed image. The default is `SCALE = 4`. The scale must be greater than or equal to 1.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-33: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

TRACK

Set this keyword and events will be generated continuously as the cursor is moved across the original image. If not set, events will only be generated when the mouse button is released. Note: On slow systems, /TRACK performance can be inadequate. The default is 0.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UVALUE

The “user value” to be assigned to the widget.

XSIZE

The width of the window (in pixels) for the original image. The default is XSIZE = 500. Note that XSIZE *must* be set to the width of the original image array for the image to display properly.

X_SCROLL_SIZE

The width of the visible part of the original image. This may be smaller than the actual width controlled by the XSIZE keyword. The default is 0, for no scroll bar.

X_ZSIZE

The width of the window for the zoomed image. The default is 250.

YSIZE

The height of the window (in pixels) for the original image. The default is 500. Note that YSIZE *must* be set to the height of the original image array for the image to display properly.

Y_SCROLL_SIZE

The height of the visible part of the original image. This may be smaller than the actual height controlled by the YSIZE keyword. The default is 0, for no scroll bar.

Y_ZSIZE

The height of the window for the zoomed image. The default is 250.

Keywords to WIDGET_CONTROL and WIDGET_INFO

The widget ID returned by most compound widgets is actually the ID of the compound widget’s base widget. This means that many keywords to the WIDGET_CONTROL and WIDGET_INFO routines that affect or return information on base widgets can be used with compound widgets.

In addition, you can use the [GET_VALUE](#) and [SET_VALUE](#) keywords to WIDGET_CONTROL to obtain or set the value of the zoom widget. The value of the CW_ZOOM widget is the original, un-zoomed image to be displayed (a two-dimensional array). To change the contents of the CW_ZOOM widget, use the command:

```
WIDGET_CONTROL, id, SET_VALUE = array
```

where `id` is the widget ID of the `CW_ZOOM` widget and `array` is the image array. The value of `CW_ZOOM` cannot be set until the widget has been realized. Note that the size of the original window, set with the `XSIZE` and `YSIZE` keywords to `CW_ZOOM`, must be the size of the input array.

To return the current zoomed image as displayed by `CW_ZOOM` in the variable `array`, use the command:

```
WIDGET_CONTROL, id, GET_VALUE = array
```

See “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual for a more complete discussion of controlling compound widgets using `WIDGET_CONTROL` and `WIDGET_INFO`.

Widget Events Returned by the CW_ZOOM Widget

When the “Report Zoom to Parent” button is pressed, this widget generates event structures with the following definition:

```
event = { ZOOM_EVENT, ID:0L, TOP:0L, HANDLER:0L, $
          XSIZE:0L, YSIZE:0L, X0:0L, Y0:0L, X1:0L, Y1:0L }
```

The `XSIZE` and `YSIZE` fields contain the dimensions of the zoomed image. The `X0` and `Y0` fields contain the coordinates of the lower left corner of the original image, and the `X1` and `Y1` fields contain the coordinates of the upper right corner of the original image.

Examples

The following code samples illustrate a use of the `CW_ZOOM` widget.

First, create an event-handler. Note that clicking on the widget’s “Zoom” button displays IDL’s help output on the console.

```
PRO widzoom_event, event

    WIDGET_CONTROL, event.id, GET_UVALUE=uvalue
    CASE uvalue OF
        'ZOOM': HELP, /STRUCT, event
        'DONE': WIDGET_CONTROL, event.top, /DESTROY
    ENDCASE

END
```

Next, create the widget program:

```

PRO widzoom

  OPENR, lun, FILEPATH('people.dat', $
    SUBDIR = ['examples', 'data']), /GET_LUN
  image=BYTARR(192,192)
  READU, lun, image
  FREE_LUN, lun
  sz = SIZE(image)

  base=WIDGET_BASE(/COLUMN)
  zoom=CW_ZOOM(base, XSIZE=sz[1], YSIZE=sz[2], UVALUE='ZOOM')
  done=WIDGET_BUTTON(base, VALUE='Done', UVALUE='DONE')
  WIDGET_CONTROL, base, /REALIZE

  WIDGET_CONTROL, zoom, SET_VALUE=image
  XMANAGER, 'widzoom', base

END

```

Once you have entered these programs, type “widzoom” at the IDL command prompt to run the widget application.

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

See Also

[ZOOM](#), [ZOOM_24](#)

DBLARR

The DBLARR function create a double-precision, floating-point vector or array of the specified dimensions.

Syntax

Result = DBLARR(*D*₁[, ..., *D*₈] [, /NOZERO])

Return Value

Returns a double-precision, floating-point vector or array.

Arguments

*D*_{*i*}

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, DBLARR sets every element of the result to zero. If NOZERO is set, this zeroing is not performed and DBLARR executes faster.

Examples

To create D, an 3-element by 3-element, double-precision, floating-point array with every element set to 0.0, enter:

```
D = DBLARR(3, 3)
```

Version History

Introduced: Original

See Also

[COMPLEXARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#), [LON64ARR](#), [LONARR](#),
[MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

DCINDGEN

The DCINDGEN function creates a complex, double-precision, floating-point array with the specified dimensions. Each element of the array has its real part set to the value of its one-dimensional subscript. The imaginary part is set to zero.

Syntax

$$Result = DCINDGEN(D_1 [, \dots, D_8])$$

Return Value

Returns a complex, double-precision, floating-point array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create DC, a 4-element vector of complex values with the real parts set to the value of their subscripts, enter:

```
DC = DCINDGEN(4)
```

Version History

Introduced: 4.0

See Also

[BINDGEN](#), [CINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#), [LINDGEN](#), [SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

DCOMPLEX

The DCOMPLEX function returns double-precision complex scalars or arrays given one or two scalars or arrays. If only one parameter is supplied, the imaginary part of the result is zero, otherwise it is set to the value of the *Imaginary* parameter.

Parameters are first converted to double-precision floating-point. If either or both of the parameters are arrays, the result is an array, following the same rules as standard IDL operators. If three parameters are supplied, DCOMPLEX extracts fields of data from *Expression*.

Syntax

$$Result = DCOMPLEX(Real [, Imaginary])$$

or

$$Result = DCOMPLEX(Expression, Offset [, D_1 [, \dots, D_8]])$$

Return Value

Returns double-precision complex scalars or arrays given one or two scalars or arrays.

Arguments

Real

Scalar or array to be used as the real part of the complex result.

Imaginary

Scalar or array to be used as the imaginary part of the complex result.

Expression

The expression from which data is to be extracted.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as complex data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid floating-point value and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Create a complex array from two integer arrays by entering the following commands:

```

; Create the first integer array:
A = [1,2,3]

; Create the second integer array:
B = [4,5,6]

; Make A the real parts and B the imaginary parts of the new
; complex array:
C = DCOMPLEX(A, B)

; See how the two arrays were combined:
PRINT, C

```

IDL prints:

```
( 1.0000000, 4.0000000)( 2.0000000, 5.0000000)
( 3.0000000, 6.0000000)
```

The real and imaginary parts of the complex array can be extracted as follows:

```
; Print the real part of the complex array C:
PRINT, 'Real Part: ', REAL_PART(C)

; Print the imaginary part of the complex array C:
PRINT, 'Imaginary Part: ', IMAGINARY(C)
```

IDL prints:

```
Real Part:          1.0000000    2.0000000    3.0000000
Imaginary Part:    4.0000000    5.0000000    6.0000000
```

Version History

Introduced: 4.0

See Also

[BYTE](#), [COMPLEX](#), [CONJ](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [IMAGINARY](#), [LONG](#),
[LONG64](#), [REAL_PART](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

DCOMPLEXARR

The DCOMPLEXARR function returns a complex, double-precision, floating-point vector or array.

Syntax

Result = DCOMPLEXARR(D_1 [, ..., D_8] [, /NOZERO])

Return Value

Returns a complex, double-precision, floating-point vector or array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, DCOMPLEXARR sets every element of the result to zero. If the NOZERO keyword is set, this zeroing is not performed, and DCOMPLEXARR executes faster.

Examples

To create an empty, 5-element by 5-element, complex array DC, enter:

```
DC = DCOMPLEXARR(5, 5)
```

Version History

Introduced: 4.0

See Also

[COMPLEXARR](#), [DBLARR](#), [FLTARR](#), [INTARR](#), [LON64ARR](#), [LONARR](#),
[MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

DEFINE_KEY

The `DEFINE_KEY` procedure programs the keyboard function *Key* with the string *Value*, or with one of the actions specified by the available keywords.

`DEFINE_KEY` is primarily intended for use with IDL's UNIX command line interface, but it has limited applications in the Microsoft Windows environment as well.

Note

Key bindings for the UNIX graphical interface (IDLDE) can be created via X Window resources. See IDL's resource file, located in your IDL distribution and described in [Chapter 8, "Customizing IDL on Motif Systems"](#) in the *Using IDL* manual, for details on key bindings.

Syntax

```
DEFINE_KEY, Key [, Value] [, /MATCH_PREVIOUS] [, /NOECHO]
[, /TERMINATE]
```

UNIX-Only Keywords: [, /BACK_CHARACTER] [, /BACK_WORD] [, /CONTROL] [, /ESCAPE] [, /DELETE_CHARACTER] [, /DELETE_CURRENT] [, /DELETE_EOL] [, /DELETE_LINE] [, /DELETE_WORD] [, /END_OF_FILE] [, /END_OF_LINE] [, /ENTER_LINE] [, /FORWARD_CHARACTER] [, /FORWARD_WORD] [, /INSERT_OVERSTRIKE_TOGGLE] [, /NEXT_LINE] [, /PREVIOUS_LINE] [, /RECALL] [, /REDRAW] [, /START_OF_LINE]

Arguments

Key

A scalar string containing the name of a function key to be programmed. IDL maintains an internal list of function key names and the escape sequences they send. Different keys are available for mapping in the UNIX command-line interface and the Windows IDL Development Environment, as described below.

UNIX — Under UNIX, `DEFINE_KEY` allows you to set the values of two distinctly different types of keys:

- Control characters: Any of the 26 control characters (CTRL+A through CTRL+Z) can be associated with specific actions by specifying the `CONTROL` keyword. Control characters are the unprintable ASCII characters

at the beginning of the ASCII character set. They are usually entered by holding down the Control key while the corresponding letter key is pressed.

- **Function keys:** Most terminals (and terminal emulators) send escape sequences when a function key is pressed. An escape sequence is a sequence of characters starting the ASCII Escape character. Escape sequences follow strict rules that allow applications such as IDL to determine when the sequence is complete. For instance, the left arrow key on most machines sends the sequence <ESC>[D. The available function keys and the escape sequences they send vary from keyboard to keyboard; IDL cannot be built to recognize all of the different keyboards in existence. The ESCAPE keyword allows you to program IDL with the escape sequences for your keyboard. When you press the function key, IDL will recognize the sequence and take the appropriate action.

If *Key* is not already on IDL's internal list, you must use the ESCAPE keyword to specify the escape sequence, otherwise, *Key* alone will suffice. The available function keys and the escape sequences they send vary from keyboard to keyboard. The SETUP_KEYS procedure should be used once at the beginning of the session to enter the keys for the current keyboard. The following table describes the standard key definitions.

Editing Key	Function
Ctrl+A	Move cursor to start of line
Ctrl+B	Move cursor left one word
Ctrl+D	EOF if current line is empty, EOL otherwise
Ctrl+E	Move to end of line
Ctrl+F	Move cursor right one word
Ctrl+K	Erase from the cursor to the end of the line
Ctrl+N	Move back one line in the recall buffer
Ctrl+R	Retype current line
Ctrl+U	Delete from current position to start of line
Ctrl+W	Delete previous word
Ctrl+X	Delete current character

Table 3-34: Standard Key Definitions for UNIX

Editing Key	Function
Backspace, Delete	Delete previous character
ESC-I	Overstrike/insert toggle
ESC-Delete	Delete previous word
Up Arrow	Move back one line in the recall buffer
Down Arrow	Move forward one line in the recall buffer
Left Arrow	Move left one character
Right Arrow	Move right one character
R13	Move cursor left one word (Sun keyboards)
R15	Move cursor right one word (Sun keyboards)
<i>^text</i>	Recall the first line containing <i>text</i> . If <i>text</i> is blank, recall the previous line
<i>Other Characters</i>	Insert character at the current cursor position

Table 3-34: Standard Key Definitions for UNIX (Continued)

Windows — Under Windows, function keys F2, F4, and F12 can be customized.

In IDL for Windows, three special variables can be used to expand the current mouse selection, the current line, or the current filename into the output of defined keys.

Variable	Expansion
%f	filename of the currently-selected IDL Editor window
%l	number of the current line in an IDL Editor window
%s	Currently-selected text from an IDL Output Log or Editor window

Table 3-35: Variable expansions for defined keys

For example, to define F2 as a key that executes an IDL PRINT command with the current mouse selection as its argument, use the command:

```
DEFINE_KEY, 'F2', 'PRINT, "%S"
```

Dragging the mouse over any text in an IDL Editor or Output Log window and pressing F2 causes the selected text to be given as the argument to the IDL PRINT procedure. The %l and %f variables can be used in a similar fashion.

Value

The scalar string that will be printed (as if it had been typed manually at the keyboard) when *Key* is pressed. If *Value* is not present, and no function is specified for the key with one of the keywords, the key is cleared so that nothing happens when it is pressed.

Keywords

BACK_CHARACTER

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move the current cursor position left one character.

BACK_WORD

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move the current cursor position left one word.

CONTROL

This keyword is only available on UNIX platforms.

Set this keyword to indicate that *Key* is the name of a control key. The default is for *Key* to define a function key escape sequence. To view the names used by IDL for the control keys, type the following at the Command Input Line:

```
HELP, /ALL_KEYS
```

Warning

The CONTROL and ESCAPE keywords are mutually exclusive and cannot be specified together.

DELETE_CHARACTER

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to delete the character to the left of the cursor.

DELETE_CURRENT

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to delete the character directly underneath the cursor.

DELETE_EOL

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to delete from the cursor position to the end of the line.

DELETE_LINE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to delete all characters to the left of the cursor.

DELETE_WORD

This keyword is only available on UNIX platforms.

Set this keyword to programs *Key* to delete the word to the left of the cursor.

END_OF_LINE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move the cursor to the end of the line.

END_OF_FILE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to exit IDL if the current line is empty, and to end the current input line if the current line is not empty.

ENTER_LINE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to enter the current line (i.e., the action normally performed by the “Return” key).

ESCAPE

This keyword is only available on UNIX platforms.

A scalar string that specifies the escape sequence that corresponds to *Key*. See “[Defining New Function Keys](#)” on page 509 for further details.

Warning

The CONTROL and ESCAPE keywords are mutually exclusive and cannot be specified together.

FORWARD_CHARACTER

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move the current cursor position right one character.

FORWARD_WORD

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move the current cursor position right one word.

INSERT_OVERSTRIKE_TOGGLE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to toggle between “insert” and “overstrike” mode. When characters are typed into the middle of a line, insert mode causes the trailing characters to be moved to the right to make room for the new ones. Overstrike mode causes the new characters to overwrite the existing ones.

MATCH_PREVIOUS

Set this keyword to program *Key* to prompt the user for a string, and then search the saved command buffer for the most recently issued command that contains that string. If a match is found, the matching command becomes the current command, otherwise the last command entered is used. Under UNIX, the default match key is the up caret “^” key when pressed in column 1.

NEXT_LINE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move forward one command in the saved command buffer and make that command the current one.

NOECHO

Set this keyword to enter the Value assigned to *Key* when pressed, without echoing the string to the screen. This feature is useful for defining keys that perform such actions as erasing the screen. If NOECHO is set, TERMINATE is also assumed to be set.

PREVIOUS_LINE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move back one command in the saved command buffer and make that command the current one.

RECALL

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to prompt the user for a command number. The saved command corresponding to the entered number becomes the current command. In order to view the currently saved commands and the number currently associated with each, enter the IDL command:

```
HELP, /RECALL COMMANDS
```

Example

The RECALL operation remembers the last command number entered, and if the user simply presses return, it recalls the command currently associated with that saved number. Since the number associated with a given command increases by one each time a new command is saved, this feature can be used to quickly replay a sequence of commands.

```
IDL> PRINT, 1
1
IDL> PRINT, 2
2
IDL> HELP, /RECALL_COMMANDS
Recall buffer length: 20
1          PRINT, 2
2          PRINT, 1
```

User presses key tied to RECALL.

```
IDL>
```

Line 2 is requested.

```
Recall Line #: 2
```

Saved command 2 is recalled.

```
IDL> PRINT, 1  
1
```

User presses return.

```
Recall Line #:
```

Saved command 2 is recalled again.

```
IDL> PRINT, 2  
2
```

REDRAW

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to retype the current line.

START_OF_LINE

This keyword is only available on UNIX platforms.

Set this keyword to program *Key* to move the cursor to the start of the line.

TERMINATE

If this keyword is set, and *Value* is present, pressing *Key* terminates the current input operation after its assigned value is entered. Essentially, an implicit carriage return is added to the end of *Value*.

Examples

Defining New Function Keys

Under UNIX, IDL can handle arbitrary function keys. When adding a definition for a function key that is not built into IDL's default list of recognized keys, you must use the ESCAPE keyword to specify the escape sequence it sends. For example, to add a function key named "HELP" which sends the escape sequence <Escape>[28~, use the command:

```
DEFINE_KEY, 'HELP', ESCAPE = '\033[28~'
```

This command adds the HELP key to the list of keys understood by IDL. Since only the key name and escape sequence were specified, pressing the HELP key will do nothing. The Value argument, or one of the keywords provided to specify command line editing functions, could have been included in the above statement to program it with an action.

Once a key is defined using the ESCAPE keyword, it is contained in the internal list of function keys. It can then be subsequently redefined without specifying the escape sequence.

It is convenient to include commonly used key definitions in a startup file, so that they will always be available. See “[Startup Files](#)” in Chapter 1 of the *Using IDL* manual.

Usually, the SETUP_KEYS procedure is used to define the function keys found on the keyboard, so it is not necessary to specify the ESCAPE keyword. For example, to program key “F2” on a Sun keyboard to redraw the current line:

```
SETUP_KEYS
DEFINE_KEY, 'F2', /REDRAW
```

The CONTROL keyword alters the action that IDL takes when it sees the specified characters defining the control keys. IDL may not be able to alter the behavior of some control characters. For example, CTRL+S and CTRL+Q are usually reserved by the operating system for flow control. Similarly, CTRL+Z is usually The UNIX suspend character.

Example

CTRL+D is the UNIX end-of-file character. It is a common UNIX convention (followed by IDL) for programs to quit upon encountering CTRL+D. However, CTRL+D is also used by some text editors to delete characters. To disable IDL default handling of CTRL+D, type the following:

```
DEFINE_KEY, /CONTROL, '^D'
```

To print a reminder of how to exit IDL properly, type the following:

```
DEFINE_KEY, /CONTROL, '^D', "print, 'Enter EXIT to quit IDL'", $
/NOECHO, /TERMINATE
```

To use CTRL+D to delete characters, type the following:

```
DEFINE_KEY, /CONTROL, '^D', /DELETE_CURRENT
```

Version History

Introduced: Original

See Also

[GET_KBRD](#), [SETUP_KEYS](#)

DEFINE_MSGBLK

The `DEFINE_MSGBLK` procedure defines and loads a new message block into the currently running IDL session. Messages are issued from a message block using the `MESSAGE` procedure, which handles the details of IDL message display, including proper formatting, setting the values of the `!ERROR_STATE` system variable, and displaying traceback information if execution halts. See [MESSAGE](#) for details.

A message block is a collection of messages that are loaded into IDL as a single unit. Each block contains the messages required for a specific application. At startup, IDL contains a single internal message block named `IDL_MBLK_CORE`, which contains the standard messages required by the IDL system. Dynamically loadable modules (DLMs) usually define additional message blocks for their own needs when they are loaded. At the IDL programming level, the `DEFINE_MSGBLK` and `DEFINE_MSGBLK_FROM_FILE` procedures can be used to define and load message blocks. You can use the `HELP, /MESSAGES` command to see the currently defined message blocks.

Syntax

```
DEFINE_MSGBLK, BlockName, ErrorNames, ErrorFormats  
[, /IGNORE_DUPLICATE] [, PREFIX = PrefixStr]
```

Note

IDL will force the values of the message block name, the individual message names, and any message prefix string to upper case before installing the message block. Because IDL is generally case-insensitive, you do not need to use upper case when supplying these values to the `DEFINE_MSGBLK` or `MESSAGE` procedures. The values stored in the `!ERROR_STATE` system variable will, however, be upper-case strings. If you do string comparisons with values in `!ERROR_STATE`, you should take this case-folding into account.

Arguments

BlockName

A string giving the name of the message block to be defined. Block names must be unique within the IDL system. We recommend that you follow the advice given in [Chapter 5, “Library Authoring”](#) in the *Building IDL Applications* manual when selecting the message block name in order to avoid name conflicts. Use of the `PREFIX` keyword is also recommended to enforce a consistent naming convention.

ErrorNames

An array of strings giving the names of the messages defined by the message block.

ErrorFormats

An array of strings giving the formats for the messages defined by the message block. Note that the format string can include both static text (displayed every time the error is displayed) and dynamic text (specified when the error is generated by a call to the MESSAGE procedure). Each format is matched with the corresponding name in *ErrorNames*. For this reason, *ErrorFormats* should have the same number of elements as *ErrorNames*. We recommend the use of the PREFIX keyword to enforce a consistent naming scheme for your messages.

Error formats are simplified `printf`-style format strings. For more information on format strings, see “[C printf-Style Quoted String Format Code](#)” in Chapter 11 of the *Building IDL Applications* manual.

Keywords

IGNORE_DUPLICATE

Attempts to define a given *BlockName* more than once in the same IDL session usually cause DEFINE_MSGBLK to issue an error and stop execution of the IDL program. Specify IGNORE_DUPLICATE to cause DEFINE_MSGBLK to quietly ignore attempts to redefine a message block. In this case, no error is issued and execution continues. The original message block remains installed and available for use.

PREFIX

It is a common and recommended practice to give each message name defined in *ErrorNames* a common unique prefix that identifies it as an error from a specific message block. However, specifying this prefix in each entry of *ErrorNames* is tedious and error prone. The PREFIX keyword can be used to specify a prefix string that will be added to each element of *ErrorNames*.

Examples

This example defines a message block called ROADRUNNER that contains 2 messages:

```
name = ['BADPLAN', 'RRNOTCAUGHT']  
fmt = ['Bad plan detected: %s.', 'Road Runner not captured.']  
DEFINE_MSGBLK, PREFIX = 'ACME_M_', 'ROADRUNNER', name, fmt
```

Once this message block is loaded, the ACME_M_BADPLAN message can be issued using the following statement:

```
MESSAGE, NAME = 'acme_m_badplan', BLOCK = 'roadrunner', $  
    'Exploding bridge while standing underneath'
```

This MESSAGE statement produces the output similar to:

```
% Bad plan detected: Exploding bridge while standing underneath.  
% Execution halted at: $MAIN$
```

The IDL command:

```
HELP, /STRUCTURES, !ERROR_STATE
```

can be used to examine the effect of this message on IDL's error state.

Version History

Introduced: 5.5

See Also

[DEFINE_MSGBLK_FROM_FILE](#), [MESSAGE](#)

DEFINE_MSGBLK_FROM_FILE

The `DEFINE_MSGBLK_FROM_FILE` procedure reads the definition of a message block from a file, and uses `DEFINE_MSGBLK` to load it into the currently running IDL session. Messages are issued from a message block using the `MESSAGE` procedure, which handles the details of IDL message display, including proper formatting, setting the values of the `!ERROR_STATE` system variable, and displaying traceback information if execution halts. See [MESSAGE](#) for details.

Note

For large message blocks, `DEFINE_MSGBLK_FROM_FILE` can be more convenient than `DEFINE_MSGBLK`.

Format of Message Definition Files

A message definition file has a simple structure designed to ease the specification of message blocks. Any line starting with the character `@` specifies information about the message block. Any line not starting with an `@` character is ignored, and can be used for comments, documentation, notes, or other human readable information. All such text is ignored by `DEFINE_MSGBLK_FROM_FILE`.

There are three different types of lines starting with `@` allowed in a message definition file:

@IDENT *name*

Specifies the name of the message block being defined. There should be exactly one such line in every message definition file. If the `BLOCK` keyword to `DEFINE_MSGBLK_FROM_FILE` is specified, the `@IDENT` line is ignored and can be omitted. RSI recommends always specifying an `@IDENT` line.

Note

IDL will force the string specified by the `@IDENT` line to upper case before installing the message block. You do not need to use upper case when supplying the `@IDENT name` string, but `!ERROR_STATE.BLOCK` will always contain an upper-case string.

@PREFIX *PrefixStr*

If present, specifies a prefix string to be applied to the beginning of each message name in the message block. There should be at most one such line in every message definition file. If the `PREFIX` keyword to `DEFINE_MSGBLK_FROM_FILE` is

specified, the @PREFIX line is ignored and can be omitted. RSI recommends always specifying an @PREFIX line.

Note

IDL will force the string specified by the @PREFIX line to upper case before installing the message block. You do not need to use upper case when supplying the @PREFIX *PrefixStr* string, but !ERROR_STATE.BLOCK will always contain an upper-case string.

@ *MessageName* "*MessageFormat*"

Specifies a single message name and format string pair. The format string should be delimited with double quotes. A message definition file should contain one such line for every message it defines.

This routine is written in the IDL language. Its source code can be found in the file `define_msgblk_from_file.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
DEFINE_MSGBLK_FROM_FILE, Filename [, BLOCK = BlockName]
    [, /IGNORE_DUPLICATE] [, PREFIX = PrefixStr] [, /VERBOSE]
```

Note

IDL will force the values of the message block name, the individual message names, and any message prefix string to upper case before installing the message block. Because IDL is generally case-insensitive, you do not need to use upper case when supplying these values to the DEFINE_MSGBLK_FROM_FILE or MESSAGE procedures. The values stored in the !ERROR_STATE system variable will, however, be upper-case strings. If you do string comparisons with values in !ERROR_STATE, you should take this case-folding into account.

Arguments

Filename

The name of the file containing the message block definition. The contents of this file must be formatted as described in the section “[Format of Message Definition Files](#)”.

Keywords

BLOCK

If present, this keyword specifies the name of the message block. Normally, this keyword is not specified, and an @IDENT line in the message file specifies the name of the block. We recommend that you follow the advice given in [Chapter 5, “Library Authoring”](#) in the *Building IDL Applications* manual when selecting this name in order to avoid name clashes. Use of a prefix is also recommended to enforce a consistent naming convention.

Note

IDL will force the string specified by the BLOCK keyword to upper case before installing the message block. You do not need to use upper case when supplying the BLOCK string to the DEFINE_MSGBLK_FROM_FILE procedure, but !ERROR_STATE.BLOCK will always contain an upper-case string.

IGNORE_DUPLICATE

Attempts to define a given *BlockName* more than once in the same IDL session usually cause DEFINE_MSGBLK to issue an error and stop execution of the IDL program. Specify IGNORE_DUPLICATE to cause DEFINE_MSGBLK to quietly ignore attempts to redefine a message block. In this case, no error is issued and execution continues. The original message block remains installed and available for use.

PREFIX

If present, this keyword specifies a prefix string to be applied to the beginning of each message name in the message block. Normally, this keyword is not specified, and an @PREFIX line in the message file specifies the prefix string. We recommend the use of a prefix to enforce a consistent naming scheme for your messages.

Note

IDL will force the string specified by the PREFIX keyword to upper case before installing the message block. You do not need to use upper case when supplying the PREFIX string to the DEFINE_MSGBLK_FROM_FILE procedure, but !ERROR_STATE.MSG_PREFIX will always contain an upper-case string.

VERBOSE

If set, causes `DEFINE_MSGBLK_FROM_FILE` to print informational messages describing the message block loaded.

Examples

The following example uses the same message block as in the example given for [“DEFINE_MSGBLK”](#) on page 511, but uses a message definition file to create the message block. The first step is to create a message definition file called `roadrunner.msg` containing the following lines:

```
; Message definition file for ROADRUNNER message block
@IDENT roadrunner
@PREFIX ACME_M_
@      BADPLAN "Bad plan detected: %s."
@      RRNOTCAUGHT "Road Runner not captured."
```

Use the following statement to load in the message block:

```
DEFINE_MSGBLK_FROM_FILE, 'roadrunner.msg'
```

Note

A message block can only be defined once during an IDL session. If you see a message that looks like this:

```
% Attempt to install an existing message block: ROADRUNNER.
% Execution halted at: $MAIN$
```

the `ROADRUNNER` message block has already been defined. You must either exit and restart IDL or issue the `.FULL_RESET_SESSION` executive command.

Once this message block is loaded, the `ACME_M_BADPLAN` message can be issued using the following statement:

```
MESSAGE, NAME = 'acme_m_badplan', BLOCK='roadrunner', $
      'Exploding bridge while standing underneath'
```

This `MESSAGE` statement produces the output similar to:

```
% Bad plan detected: Exploding bridge while standing underneath.
% Execution halted at: $MAIN$
```

The IDL command:

```
HELP, /STRUCTURES, !ERROR_STATE
```

can be used to examine the effect of this message on IDL's error state.

Version History

Introduced: 5.5

See Also

[DEFINE_MSGBLK](#), [MESSAGE](#)

DEFROI

The DEFROI function defines an irregular region of interest of an image using the image display system and the cursor and mouse. DEFROI only works for interactive, pixel oriented devices with a cursor and an exclusive or writing mode. Regions may have at most 1000 vertices.

Warning

DEFROI does not function correctly when used with draw widgets. See [CW_DEFROI](#).

This routine is written in the IDL language. Its source code can be found in the file `defroi.pro` in the `lib` subdirectory of the IDL distribution.

Using DEFROI

After calling DEFROI, click in the image with the left mouse button to mark points on the boundary of the region of interest. The points are connected in sequence. Alternatively, press and hold the left mouse button and drag to draw a curved region. Click the middle mouse button to erase points. The most recently-placed point is erased first. Click the right mouse button to close the region. The function returns after the region has been closed.

Syntax

```
Result = DEFROI( Sx, Sy [, Xverts, Yverts] [, /NOREGION] [, /NOFILL]
               [, /RESTORE] [, X0, Y0=device_coord, device_coord] [, ZOOM=factor] )
```

Return Value

Returns a vector of subscripts of the pixels inside the region. The lowest bit in which the write mask is enabled is changed.

Arguments

Sx, Sy

Integers specifying the horizontal and vertical size of image, in pixels.

Xverts, Yverts

Named vectors that will contain the vertices of the enclosed region.

Keywords

NOREGION

Set this keyword to inhibit the return of the pixel subscripts.

NOFILL

Set this keyword to inhibit filling of the defined region on completion.

RESTORE

Set this keyword to restore the display to its original state upon completion.

X0, Y0

Set these keywords equal to the coordinates of the lower left corner of the displayed image (in device coordinates). If omitted, the default value (0,0) is used.

ZOOM

Set this keyword equal to the zoom factor. If not specified, a value of 1 is assumed.

Example

```
; Create an image:
TVSCL, DIST(200,200)

; Call DEFROI. The cursor becomes active in the graphics window.
; Define a region and click the right mouse button:
X = DEFROI(200, 200)

; Print subscripts of points included in the defined region:
PRINT, X
```

Version History

Introduced: Original

See Also

[CW_DEFROI](#)

DEFSYSV

The DEFSYSV procedure creates a new system variable called *Name* initialized to *Value*.

Syntax

```
DEFSYSV, Name, Value [, Read_Only] [, EXISTS=variable]
```

Arguments

Name

A scalar string containing the name of the system variable to be created. All system variable names must begin with the character '!'.

Value

An expression from which the type, structure, and initial value of the new system variable is taken. *Value* can be a scalar, array, or structure.

Read_Only

If the *Read_Only* argument is present and nonzero, the value of the newly-created system variable cannot be changed (i.e., the system variable is read-only, like the !PI system variable). Otherwise, the value of the new system variable can be modified.

Keywords

EXISTS

Set this keyword to a named variable that returns 1 if the system variable specified by *Name* exists. If this keyword is specified, *Value* can be omitted. For example, the following commands could be used to check that the system variable XYZ exists:

```
DEFSYSV, '!XYZ', EXISTS = i
IF i EQ 1 THEN PRINT, '!XYZ exists' ELSE PRINT, $
    '!XYZ does not exist'
```

Examples

To create a new, floating-point, scalar system variable called !NEWVAR with an initial value of 2.0, enter:

```
DEFSYSV, '!NEWVAR', 2.0
```

You can both define and use a system variable within a single procedure:

```
PRO foo
  DEFSYSV, '!foo', $
    'Rocky, watch me pull a squirrel out of my hat!'

  ; Print !foo after defining it:
  PRINT, !foo
END
```

Version History

Introduced: Original

See Also

[Appendix D, “System Variables”](#)

DELVAR

The DELVAR procedure deletes variables from the main IDL program level. DELVAR frees any memory used by the variable and removes it from the main program's symbol table. The following restrictions apply:

- DELVAR can only be called from the main program level.
- If a main program is created with the .RUN or .RNEW command, then each time DELVAR is called, this main program is erased. Variables that are not deleted remain unchanged.

Syntax

```
DELVAR,  $V_1$ , ...,  $V_n$ 
```

Arguments

V_i

One or more named variables to be deleted.

Examples

Suppose that the variable Q is defined at the main program level. Q can be deleted by entering:

```
DELVAR, Q
```

Version History

Introduced: Pre 4.0

See Also

[TEMPORARY](#)

DENDRO_PLOT

Given a hierarchical tree cluster, usually created by the CLUSTER_TREE function, the DENDRO_PLOT procedure draws a two-dimensional dendrite plot on the current direct graphics device.

This routine is written in the IDL language. Its source code can be found in the file `dendro_plot.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
DENDRO_PLOT, Clusters, Linkdistance [, LABEL_CHARSIZE=value]
  [, LABEL_CHARTHICK=value] [, LABEL_COLOR=value]
  [, LABEL_NAMES=vector] [, LABEL_ORIENTATION=value]
  [, LINECOLOR=value] [, ORIENTATION={1|2|3|4}] [, /OVERPLOT]
```

Graphics Keywords: [, BACKGROUND=*color_index*] [, CHARSIZE=*value*]
 [, CHARTHICK=*integer*] [, CLIP=[*X₀*, *Y₀*, *X₁*, *Y₁*]] [, COLOR=*value*] [, /DATA |
 /DEVICE | /NORMAL] [, FONT=*integer*] [, LINSTYLE={0 | 1 | 2 | 3 | 4 | 5}]
 [, /NOCLIP] [, /NODATA] [, /NOERASE] [, POSITION=[*X₀*, *Y₀*, *X₁*, *Y₁*]]
 [, PSYM=*integer*{0 to 10}] [, SUBTITLE=*string*] [, SYMSIZE=*value*] [, /T3D]
 [, THICK=*value*] [, TICKLEN=*value*] [, TITLE=*string*]
 [, {X | Y | Z}CHARSIZE=*value*]
 [, {X | Y | Z}GRIDSTYLE=*integer*{0 to 5}]
 [, {X | Y | Z}MARGIN=[*left*, *right*]]
 [, {X | Y | Z}MINOR=*integer*]
 [, {X | Y | Z}RANGE=[*min*, *max*]]
 [, {X | Y | Z}STYLE=*value*]
 [, {X | Y | Z}THICK=*value*]
 [, {X | Y | Z}TICK_GET=*variable*]
 [, {X | Y | Z}TICKFORMAT=*string*]
 [, {X | Y | Z}TICKINTERVAL=*value*]
 [, {X | Y | Z}TICKLAYOUT=*scalar*]
 [, {X | Y | Z}TICKLEN=*value*]
 [, {X | Y | Z}TICKNAME=*string_array*]
 [, {X | Y | Z}TICKS=*integer*]
 [, {X | Y | Z}TICKUNITS=*string*]
 [, {X | Y | Z}TICKV=*array*]
 [, {X | Y | Z}TITLE=*string*]
 [, ZVALUE=*value*{0 to 1}]

Arguments

Clusters

A 2-by- $(m-1)$ input array containing the cluster indices, where m is the number of items in the original dataset. This array is usually the result of the CLUSTER_TREE function.

Linkdistance

An $(m-1)$ -element input vector containing the distances between cluster items, as returned by the Linkdistance argument to the CLUSTER_TREE function.

Keywords

LABEL_CHARSIZE

The overall character size for the leaf labels when Hershey or TrueType fonts are selected. This keyword does not apply when hardware fonts are selected. If this keyword is omitted, then the value of the CHARSIZE keyword is used.

LABEL_CHARTHICK

An integer value specifying the line thickness of the vector drawn font characters for the leaf labels. This keyword has no effect when used with the hardware drawn fonts. If this keyword is omitted, then the value of the CHARTHICK keyword is used.

LABEL_COLOR

The color index of the leaf labels. If this keyword is omitted, then the value of the COLOR keyword is used.

LABEL_NAMES

Set this keyword to an m -element string array containing the leaf labels. If this keyword is omitted then leaves are labelled using integers $0 \dots m-1$. If this keyword is set to a scalar value then the leaf labels are not drawn.

Note

The LABEL_NAMES should be input in their original order, corresponding to the m items of the original dataset. DENDRO_PLOT will automatically rearrange the labels to match the order of the leaf nodes in the dendrogram.

LABEL_ORIENTATION

Specifies the counterclockwise angle in degrees from horizontal of the text baseline for the leaf labels. If this keyword is omitted then labels are drawn horizontally, unless `ORIENTATION=0` or `2`, in which case the labels are drawn vertically if the largest label has more than two characters.

LINECOLOR

The color index of the dendrite lines. If this keyword is omitted, then the value of the `COLOR` keyword is used.

ORIENTATION

Set this keyword to an integer giving the orientation of the dendrite plot. Possible values are:

Value	Orientation
0	(Default): Bottom-to-top. Leaf nodes are drawn at the bottom.
1	Left-to-right. Leaf nodes are drawn at the left.
2	Top-to-bottom. Leaf nodes are drawn at the top.
3	Right-to-left. Leaf nodes are drawn at the right.

Table 3: ORIENTATION Values

OVERPLOT

Set this keyword to avoid creating a new dendrite plot and axes. Instead, the dendrogram lines and leaf labels are drawn on the existing plot window.

Note

If `OVERPLOT` is specified, then for `ORIENTATION = 0` or `2`, all keywords which affect the Y axis are ignored. Conversely, for `ORIENTATION = 1` or `3`, all keywords which affect the X axis are ignored.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above: `BACKGROUND`, `CHARSIZE`, `CHARTHICK`, `CLIP`, `COLOR`, `DATA`, `DEVICE`, `FONT`, `LINestyle`, `NOCLIP`, `NODATA`, `NOERASE`,

NORMAL, POSITION, PSYM, SUBTITLE, SYMSIZE, T3D, THICK, TICKLEN, TITLE, [XYZ]CHARSIZE, [XYZ]GRIDSTYLE, [XYZ]MARGIN, [XYZ]MINOR, [XYZ]RANGE, [XYZ]STYLE, [XYZ]THICK, [XYZ]TICKFORMAT, [XYZ]TICKINTERVAL, [XYZ]TICKLAYOUT, [XYZ]TICKLEN, [XYZ]TICKNAME, [XYZ]TICKS, [XYZ]TICKUNITS, [XYZ]TICKV, [XYZ]TICK_GET, [XYZ]TITLE, ZVALUE.

Note

For ORIENTATION=0 or 2, the XRANGE, XTICKLEN, XTICKS, XTICKV, and XTICKNAME keywords are ignored. For ORIENTATION=1 or 3, the YRANGE, YTICKLEN, YTICKS, YTICKV, and YTICKNAME keywords are ignored.

Note

For the dendrite plot the default values for XSTYLE and YSTYLE are 1.

Example

```

; Given a set of points in two-dimensional space.
m = 20
data = 7*RANDOMN(-1, 2, m)

; Compute the Euclidean distance between each point.
distance = DISTANCE_MEASURE(data)

; Compute the cluster analysis.
clusters = CLUSTER_TREE(distance, linkdistance, LINKAGE = 2)

DENDRO_PLOT, clusters, linkdistance, $
  POSITION = [0.08, 0.1, 0.48, 0.9], $
  XSTYLE = 9, YSTYLE = 9, $
  XTITLE = 'Leaf', YTITLE = 'Distance'

DENDRO_PLOT, clusters, linkdistance, $
  ORIENTATION = 1, /NOERASE, $
  POSITION=[0.56, 0.1, 0.96, 0.9], $
  XSTYLE = 9, YSTYLE = 9, $
  XTITLE = 'Distance', YTITLE = 'Leaf'

```

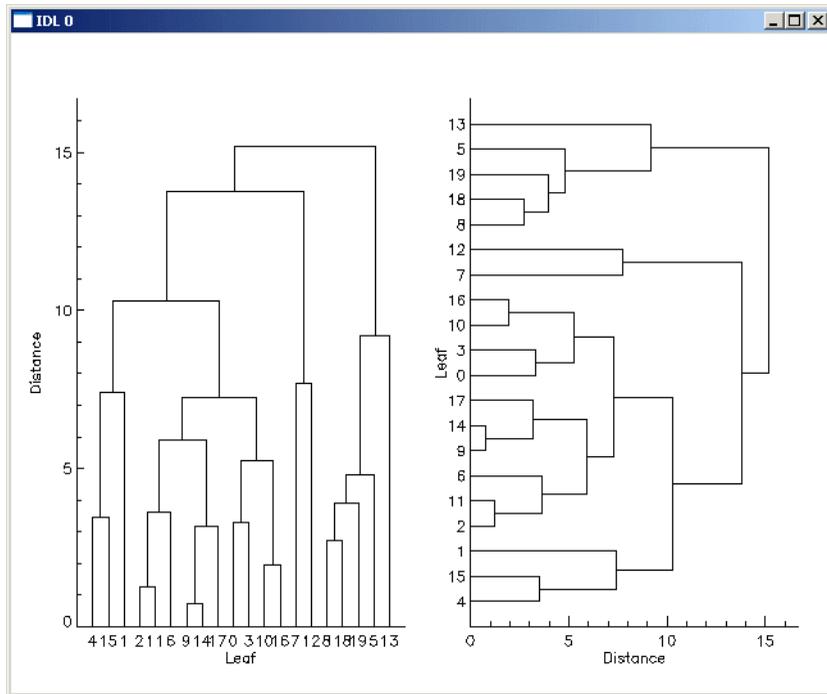


Figure 3-16: Dendroplot

Version History

Introduced: 6.1

See Also

[CLUSTER_TREE](#), [DENDROGRAM](#), [DISTANCE_MEASURE](#)

DENDROGRAM

Given a hierarchical tree cluster, usually created by the `CLUSTER_TREE` function, the `DENDROGRAM` procedure constructs a dendrogram and returns a set of vertices and connectivity that can be used to visualize the dendrite plot.

This routine is written in the IDL language. Its source code can be found in the file `dendro_plot.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
DENDROGRAM, Clusters, Linkdistance, Outverts, Outconn  
[, LEAFNODES=variable]
```

Arguments

Clusters

A 2-by- $(m-1)$ input array containing the cluster indices, where m is the number of items in the original dataset. This array is usually the result of the `CLUSTER_TREE` function.

Linkdistance

An $(m-1)$ -element input vector containing the distances between cluster items, as returned by the `Linkdistance` argument to the `CLUSTER_TREE` function.

Outverts

Set this argument to a named variable that will contain a 2-dimensional array of floating-point vertices that make up the dendrogram.

Outconn

Set this argument to a named variable that will contain an output array of connectivity values. The `Outverts` and `Outconn` arguments can be passed directly to `IDLgrPolyline` to construct the dendrite plot.

Keywords

LEAFNODES

Set this keyword to a named variable that will return a vector of integers giving the order of leaf nodes within the dendrogram. The LEAFNODES keyword is useful for labeling the nodes in a dendrite plot.

Example

```

; Given a set of points in two-dimensional space.
m = 20
DATA = 7*RANDOMN(-1, 2, m)

; Compute the Euclidean distance between each point.
distance = DISTANCE_MEASURE(data)

; Compute the cluster analysis.
clusters = CLUSTER_TREE(distance, linkdistance, LINKAGE=2)

; Create the dendrogram.
DENDROGRAM, clusters, linkdistance, outverts, outconn, $
    LEAFNODES = LEAFNODES

PRINT, STRTRIM(LEAFNODES, 2)

OPOLY = OBJ_NEW('IDLgrPolyline', outverts, $
    POLYLINES = outconn)
LOC = FLTARR(2, m)
LOC[0, *] = FINDGEN(m)
OTEXT = OBJ_NEW('IDLgrText', STRTRIM(LEAFNODES,2), $
    ALIGNMENT = 1, VERTICAL_ALIGN = 0.5, $
    BASELINE = [0,1,0], UPDIR = [-1,0,0], $
    CHAR_DIM = [1,1], LOCATIONS = loc)
OAXIS = OBJ_NEW('IDLgrAxis', 1, /EXACT, $
    LOCATION = [-1,0,0], RANGE = [0, MAX(linkdistance)])
OAXIS -> GetProperty, TICKTEXT = oTick
OTICK -> SetProperty, CHAR_DIM = [1,1]
OMODEL = OBJ_NEW('IDLgrModel')
OMODEL -> Add, oPoly
OMODEL -> Add, oText
OMODEL -> Add, oAxis
XOBJVIEW, oModel

```

When this code is run, IDL prints:

```
4 15 1 2 11 6 9 14 17 0 3 10 16 7 12 8 18 19 5 13
```

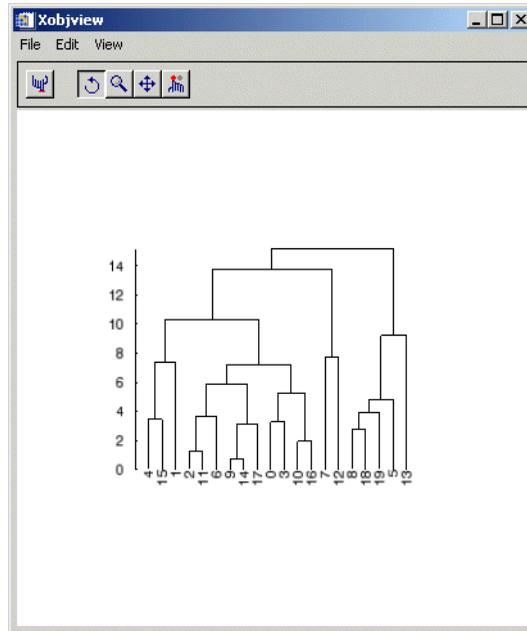


Figure 3-17: Dendrogram Plot

Version History

Introduced: 6.1

See Also

[CLUSTER_TREE](#), [DENDRO_PLOT](#), [DISTANCE_MEASURE](#)

DERIV

The DERIV function performs numerical differentiation using 3-point, Lagrangian interpolation.

Syntax

Result = DERIV([X,] Y)

Return Value

Returns the derivative of the numerical differentiation.

Arguments

X

Differentiate with respect to this variable. If omitted, unit spacing for Y (i.e., $X_i = i$) is assumed.

Y

The variable to be differentiated.

Examples

```
X = [ 0.1, 0.3, 0.4, 0.7, 0.9]
Y = [ 1.2, 2.3, 3.2, 4.4, 6.6]
PRINT, DERIV(Y)
PRINT, DERIV(X,Y)
```

IDL prints:

```
1.20000  1.00000  1.05000  1.70000  2.70000
3.16666  7.83333  7.75000  8.20000  13.8000
```

Version History

Introduced: Original

See Also

[DERIVSIG](#)

DERIVSIG

The DERIVSIG function computes the standard deviation of a derivative as found by the DERIV function, using the input variables of DERIV and the standard deviations of those input variables.

Syntax

$$\text{Result} = \text{DERIVSIG}([X, Y, \text{Sig}_x] \text{Sig}_y)$$

Return Value

Returns the standard deviation of a derivative as found by the DERIV function.

Arguments

X

Differentiate with respect to this variable. If omitted, unit spacing for Y (i.e., $X_i = i$) is assumed.

Y

The variable to be differentiated. Omit if X is omitted.

Sigx

The standard deviation of X (either vector or constant). Use “0.0” if the abscissa is exact; omit if X is omitted.

Sigy

The standard deviation of Y. *Sigy* must be a vector if the other arguments are omitted, but may be either a vector or a constant if X, Y, and *Sigx* are supplied.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[DERIV](#)

DETERM

The DETERM function computes the determinant of an n by n array. LU decomposition is used to represent the input array in triangular form. The determinant is then computed as the product of diagonal elements of the triangular form. Row interchanges are tracked during the LU decomposition to ensure the correct sign.

This routine is written in the IDL language. Its source code can be found in the file `determ.pro` in the `lib` subdirectory of the IDL distribution.

Note

If you are working with complex inputs, instead use the `LA_DETERM` procedure.

Syntax

Result = DETERM(*A* [, /CHECK] [, /DOUBLE] [, ZERO=*value*])

Return Value

Returns the determinant of an n by n array

Arguments

A

An n by n single- or double-precision floating-point array.

Keywords

CHECK

Set this keyword to check *A* for singularity. The determinant of a singular array is returned as zero if this keyword is set. Run-time errors may result if *A* is singular and this keyword is not set.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ZERO

Use this keyword to set the absolute value of the floating-point zero. A floating-point zero on the main diagonal of a triangular array results in a zero determinant. For single-precision inputs, the default value is 1.0×10^{-6} . For double-precision inputs, the default value is 1.0×10^{-12} . Setting this keyword to a value less than the default may improve the precision of the result.

Examples

```
; Define an array A:
A = [[ 2.0,  1.0,  1.0], $
      [ 4.0, -6.0,  0.0], $
      [-2.0,  7.0,  2.0]]

; Compute the determinant:
PRINT, DETERM(A)
```

IDL prints:

```
-16.0000
```

Version History

Introduced: Original

See Also

[COND](#), [INVERT](#), [LA_DETERM](#)

DEVICE

The DEVICE procedure provides device-dependent control over the current graphics device (as set by the SET_PLOT routine). The IDL graphics procedures and functions are device-independent. That is, IDL presents the user with a consistent interface to all devices. However, most devices have extra abilities that are not directly available through this interface. DEVICE is used to access and control such abilities. It is used by specifying various keywords that differ from device to device.

See [Appendix A, “IDL Graphics Devices”](#) for a description of the keywords available for each graphics device.

Syntax

Note

Each keyword to DEVICE is followed by the device(s) to which it applies.

DEVICE

```
[, /AVANTGARDE |, /BKMAN |, /COURIER |, /HELVETICA |, /ISOLATIN1 |,
/PALATINO |, /SCHOOLBOOK |, /SYMBOL |, /TIMES |, /ZAPFCHANCERY |,
/ZAPFDINGBATS {PS}]
[, /AVERAGE_LINES{REGIS}]
[, /BINARY |, /NCAR |, /TEXT {CGM}]
[, BITS_PER_PIXEL={1 | 2 | 4 | 8} {PS}]
[, /BOLD{PS}]
[, /BOOK{PS}]
[, /BYPASS_TRANSLATION{WIN, X}]
[, /CLOSE{Z}]
[, /CLOSE_DOCUMENT{PRINTER}]
[, /CLOSE_FILE{CGM, HP, METAFILE, PCL, PS, REGIS, TEK}]
[, /CMYK{PS}]
[, /COLOR{PCL, PS}]
[, COLORS=value{CGM, TEK}]
[, COPY=[Xsource, Ysource, cols, rows, Xdest, Ydest [, Window_index]]{WIN, X}]
[, /CURSOR_CROSSHAIR{WIN, X}]
[, CURSOR_IMAGE=value{ 16-element short int vector} {WIN, X}]
[, CURSOR_MASK=value{WIN, X}]
[, /CURSOR_ORIGINAL{WIN, X}]
[, CURSOR_STANDARD=value{WIN: arrow=32512,
I-beam=32513, hourglass=32514, black cross=32515, up arrow=32516,
```

```

size(Windows)=32640, icon(Windows)=32641, size NW-SE=32642, size NE-
SW=32643, size E-W=32644, size N-S=32645}{X: one of the values in file
cursorfonts.h}}
[, CURSOR_XY=[x,y]{WIN, X}]
[, /DECOMPOSED{WIN, X}]
[, /DIRECT_COLOR{X}]
[, EJECT={0 | 1 | 2}{HP}]
[, ENCAPSULATED={0 | 1}{PS}]
[, ENCODING={1 (binary) | 2 (text) | 3 (NCAR binary)}{CGM}]
[, FILENAME=filename{CGM, HP, METAFILE, PCL, PS, REGIS, TEK}]
[, /FLOYD{PCL, X}]
[, FONT_INDEX=integer{PS}]
[, FONT_SIZE=points{PS}]
[, GET_CURRENT_FONT=variable{METAFILE, PRINTER, WIN, X}]
[, GET_DECOMPOSED=variable{WIN, X}]
[, GET_FONTNAMES=variable{METAFILE, PRINTER, WIN, X}]
[, GET_FONTNUM=variable{METAFILE, PRINTER, WIN, X}]
[, GET_GRAPHICS_FUNCTION=variable{WIN, X, Z}]
[, GET_PAGE_SIZE=variable{PRINTER}]
[, GET_SCREEN_SIZE=variable{WIN, X}]
[, GET_VISUAL_DEPTH=variable{WIN, X}]
[, GET_VISUAL_NAME=variable{WIN, X}]
[, GET_WINDOW_POSITION=variable{WIN, X}]
[, GET_WRITE_MASK=variable{X, Z}]
[, GIN_CHARS=number_of_characters{TEK}]
[, GLYPH_CACHE=number_of_glyphs{METAFILE, PRINTER, PS, WIN, Z}]
[, /INCHES{HP, PCL, METAFILE, PRINTER, PS}]
[, /INDEX_COLOR{METAFILE, PRINTER}]
[, /ITALIC{PS}]
[, /LANDSCAPE | , /PORTRAIT{HP, PCL, PRINTER, PS}]
[, LANGUAGE_LEVEL={1 | 2}{PS}]
[, /DEMI | , /LIGHT | , /MEDIUM | , /NARROW | , /OBLIQUE{PS}]
[, OPTIMIZE={0 | 1 | 2}{PCL}] [, /ORDERED{PCL, X}]
[, OUTPUT=scalar string{HP, PS}]
[, /PIXELS{PCL}]
[, PLOT_TO=logical unit num{REGIS, TEK}]
[, /PLOTTER_ON_OFF{HP}]
[, /POLYFILL{HP}]
[, PRE_DEPTH=value{PS}]
[, PRE_XSIZE=width{PS}]
[, PRE_YSIZE=height{PS}]

```

```

[, /PREVIEW{PS}]
[, PRINT_FILE=filename{WIN}]
[, /PSEUDO_COLOR{X}]
[, RESET_STRING=string{TEK}]
[, RESOLUTION=value{PCL}]
[, RETAIN={0 | 1 | 2}{WIN, X}]
[, SCALE_FACTOR=value{PRINTER, PS}]
[, SET_CHARACTER_SIZE=[font size, line spacing]{CGM, HP, METAFILE, PCL,
PS, REGIS, TEK, WIN, X, Z}]
[, SET_COLORMAP=value{14739-element byte vector}{PCL}]
[, SET_COLORS=value{2 to 256}{Z}]
[, SET_FONT=scalar string{METAFILE, PRINTER, PS, WIN, Z}]
[, SET_GRAPHICS_FUNCTION=code{0 to 15}{WIN, X, Z}]
[, SET_RESOLUTION=[width, height]{Z}]
[, SET_STRING=string{TEK}]
[, SET_TRANSLATION=variable{X}]
[, SET_WRITE_MASK=value{0 to 2n-1 for n-bit system}{X, Z}]
[, STATIC_COLOR=value{bits per pixel}{X}]
[, STATIC_GRAY=value{bits per pixel}{X}]
[, /TEK4014{TEK}]
[, /TEK4100{TEK}]
[, THRESHOLD=value{PCL, X}]
[, TRANSLATION=variable{WIN, X}]
[, TRUE_COLOR=value{bits per pixel}{METAFILE, PRINTER, X}]
[, /TT_FONT{METAFILE, PRINTER, PS, WIN, X, Z}]
[, /TTY{REGIS, TEK}]
[, /VT240 | , /VT241 | , /VT340 | , /VT341 {REGIS}]
[, WINDOW_STATE=variable{WIN, X}]
[, XOFFSET=value{HP, PCL, PRINTER, PS}]
[, XON_XOFF={0 | 1 (default)}{HP}]
[, XSIZE=width{HP, METAFILE, PCL, PRINTER, PS}]
[, YOFFSET=value{HP, PCL, PRINTER, PS}]
[, YSIZE=height{HP, PCL, METAFILE, PRINTER, PS}]
[, Z_BUFFERING={0 | 1 (default)}{Z}]

```

Keywords

See “[Keywords Accepted by the IDL Devices](#)” on page 4246.

Examples

The following example retains the name of the current graphics device, sets plotting to the PostScript device, makes a PostScript file, then resets plotting to the original device:

```
; The NAME field of the !D system variable contains the name of the
; current plotting device.
mydevice = !D.NAME

; Set plotting to PostScript:
SET_PLOT, 'PS'

; Use DEVICE to set some PostScript device options:
DEVICE, FILENAME='myfile.ps', /LANDSCAPE

; Make a simple plot to the PostScript file:
PLOT, FINDGEN(10)

; Close the PostScript file:
DEVICE, /CLOSE

; Return plotting to the original device:
SET_PLOT, mydevice
```

Version History

Introduced: Original

CMYK keyword added: 6.1

DFPMIN

The DFPMIN procedure minimizes a user-written function *Func* of two or more independent variables using the Broyden-Fletcher-Goldfarb-Shanno variant of the Davidon-Fletcher-Powell method, using its gradient as calculated by a user-written function *Dfunc*.

DFPMIN is based on the routine `dfuncmin` described in section 10.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
DFPMIN, X, Gtol, Fmin, Func, Dfunc [, /DOUBLE] [, EPS=value]
      [, ITER=variable] [, ITMAX=value] [, STEPMAX=value] [, TOLX=value]
```

Arguments

X

On input, *X* is an *n*-element vector specifying the starting point. On output, it is replaced with the location of the minimum.

Gtol

An input value specifying the convergence requirement on zeroing the gradient.

Fmin

On output, *Fmin* contains the value at the minimum-point *X* of the user-supplied function specified by *Func*.

Func

A scalar string specifying the name of a user-supplied IDL function of two or more independent variables to be minimized. This function must accept a vector argument *X* and return a scalar result.

For example, suppose we wish to find the minimum value of the function

$$y = (x_0 - 3)^4 + (x_1 - 2)^2$$

To evaluate this expression, we define an IDL function named MINIMUM:

```

FUNCTION minimum, X
  RETURN, (X[0] - 3.0)^4 + (X[1] - 2.0)^2
END

```

Dfunc

A scalar string specifying the name of a user-supplied IDL function that calculates the gradient of the function specified by *Func*. This function must accept a vector argument *X* and return a vector result.

For example, the gradient of the above function is defined by the partial derivatives:

$$\frac{\partial y}{\partial x_0} = 4(x_0 - 3)^3, \quad \frac{\partial y}{\partial x_1} = 2(x_1 - 2)$$

We can write a function `GRAD` to express these relationships in the IDL language:

```

FUNCTION grad, X
  RETURN, [4.0*(X[0] - 3.0)^3, 2.0*(X[1] - 2.0)]
END

```

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EPS

Use this keyword to specify a number close to the machine precision. For single-precision calculations, the default value is 3.0×10^{-8} . For double-precision calculations, the default value is 3.0×10^{-16} .

ITER

Use this keyword to specify a named variable which returns the number of iterations performed.

ITMAX

Use this keyword to specify the maximum number of iterations allowed. The default value is 200.

STEPMAX

Use this keyword to specify the scaled maximum step length allowed in line searches. The default value is 100.0

TOLX

Use this keyword to specify the convergence criterion on X values. The default value is $4 \times \text{EPS}$.

Examples

To minimize the function MINIMUM:

```

PRO example_dfpmn

    ; Make an initial guess (the algorithm's starting point):
    X = [1.0, 1.0]

    ; Set the convergence requirement on the gradient:
    Gtol = 1.0e-7

    ; Find the minimizing value:
    DFPMIN, X, Gtol, Fmin, 'minimum', 'grad'

    ; Print the minimizing value:
    PRINT, X

END

FUNCTION minimum, X
    RETURN, (X[0] - 3.0)^4 + (X[1] - 2.0)^2
END

FUNCTION grad, X
    RETURN, [4.0*(X[0] - 3.0)^3, 2.0*(X[1] - 2.0)]
END

```

IDL prints:

```
3.00175  2.00000
```

Version History

Introduced: 4.0

See Also

[AMOEBA](#), [POWELL](#), [SIMPLEX](#)

DIAG_MATRIX

The `DIAG_MATRIX` function constructs a diagonal matrix from an input vector, or if given a matrix, then `DIAG_MATRIX` will extract a diagonal vector.

Syntax

$$\text{Result} = \text{DIAG_MATRIX}(A [, \text{Diag}])$$

Return Value

- If given an input vector with n values, the result is an n -by- n array of the same type. The `DIAG_MATRIX` function may also be used to construct subdiagonal or superdiagonal arrays.
- If given an input n -by- m array, the result is a vector with $\text{MIN}(n,m)$ elements containing the diagonal elements. The `DIAG_MATRIX` function may also be used to extract subdiagonals or superdiagonals.

Arguments

A

Either an n -element input vector to convert to a diagonal matrix, or a n -by- m input array to extract a diagonal. A may be any numeric type.

Diag

An optional argument that specifies the subdiagonal ($\text{Diag} < 0$) or superdiagonal ($\text{Diag} > 0$) to fill or extract. The default is $\text{Diag}=0$ which puts or extracts the values along the diagonal. If A is a vector with the m elements, then the result is an n -by- n array, where $n = m + \text{ABS}(\text{Diag})$. If A is an array, then the result is a vector whose length depends upon the number of elements remaining along the subdiagonal or superdiagonal.

Tip

The *Diag* argument may be used to easily construct tridiagonal arrays. For example, the expression,

$$\text{DIAG_MATRIX}(VL, -1) + \text{DIAG_MATRIX}(V) + \text{DIAG_MATRIX}(VU, 1)$$

will create an n -by- n array, where VL is an $(n - 1)$ -element vector containing the

subdiagonal values, V is an n -element vector containing the diagonal values, and VU is an $(n - 1)$ -element vector containing the superdiagonal values.

Keywords

None.

Examples

Create a tridiagonal matrix and extract the diagonal using the following program:

```

PRO ExDiagMatrix
; Convert three input vectors to a tridiagonal matrix:
diag = [1, -2, 3, -4]
sub = [5, 10, 15]
super = [3, 6, 9]
array = DIAG_MATRIX(diag) + $
DIAG_MATRIX(super, 1) + DIAG_MATRIX(sub, -1)
PRINT, 'DIAG_MATRIX array:'
PRINT, array

; Extract the diagonal:
PRINT, 'DIAG_MATRIX diagonal:'
PRINT, DIAG_MATRIX(array)
END

```

When this program is compiled and run, IDL prints:

```

DIAG_MATRIX array:
1      3      0      0
5     -2      6      0
0     10      3      9
0      0     15     -4
DIAG_MATRIX diagonal:
1     -2      3     -4

```

Version History

Introduced: 5.6

See Also

[IDENTITY](#), [MATRIX_MULTIPLY](#), “[Multiplying Arrays](#)” in Chapter 22 of the *Using IDL* manual.

DIALOG_MESSAGE

The `DIALOG_MESSAGE` function creates a modal (blocking) dialog box that can be used to display information for the user. The dialog must be dismissed, by clicking on one of its option buttons, before execution of the widget program can continue.

This function differs from widgets in a number of ways. The `DIALOG_MESSAGE` dialog does not exist as part of a widget tree, has no children, does not exist in an unrealized state, and generates no events. Instead, the dialog is displayed whenever this function is called. While the `DIALOG_MESSAGE` dialog is displayed, widget activity is limited because the dialog is modal. The function does not return to its caller until the user selects one of the dialog's buttons. Once a button has been selected, the dialog disappears.

There are four basic dialogs that can be displayed. The default type is “Warning”. Other types can be selected by setting one of the keywords described below. Each dialog type displays different buttons. Additionally any dialog can be made to show a “Cancel” button by setting the `CANCEL` keyword. The four types of dialogs are described in the table below:

Dialog Type	Default Buttons
Error	OK
Warning	OK
Question	Yes, No
Information	OK

Table 3-36: Types of `DIALOG_MESSAGE` Dialogs

Syntax

```
Result = DIALOG_MESSAGE( Message_Text [, /CANCEL
    [, /DEFAULT_CANCEL | , /DEFAULT_NO] [, DIALOG_PARENT=widget_id]
    [, DISPLAY_NAME=string] [, /ERROR | , /INFORMATION | , /QUESTION]
    [, RESOURCE_NAME=string] [, TITLE=string] )
```

Return Value

`DIALOG_MESSAGE` returns a text string containing the text of the button selected by the user. Possible returned values are “Yes”, “No”, “OK”, and “Cancel”.

Arguments

Message_Text

A scalar string or string array that contains the text of the message to be displayed. If this argument is set to an array of strings, each array element is displayed as a separate line of text.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Keywords

CANCEL

Set this keyword to add a “Cancel” button to the dialog.

DEFAULT_CANCEL

Set this keyword to make the “Cancel” button the default selection for the dialog. The default selection is the button that is selected when the user presses the default keystroke (usually Space or Return depending on the platform). Setting DEFAULT_CANCEL implies that the CANCEL keyword is also set.

DEFAULT_NO

Set this keyword to make the “No” button the default selection for “Question” dialogs. Normally, the default is “Yes”.

DIALOG_PARENT

Set this keyword to the widget ID of a widget over which the message dialog should be positioned. When displayed, the DIALOG_MESSAGE dialog will be positioned over the specified widget. Dialogs are often related to a non-dialog widget tree. The ID of the widget in that tree to which the dialog is most closely related should be specified.

DISPLAY_NAME

Set this keyword equal to a string indicating the name of the X Windows display on which the dialog is to appear. This keyword is ignored if the `DIALOG_PARENT` keyword is specified. This keyword is also ignored on Microsoft Windows platforms.

ERROR

Set this keyword to create an “Error” dialog. The default dialog type is “Warning”.

INFORMATION

Set this keyword to create an “Information” dialog. The default dialog type is “Warning”.

QUESTION

Set this keyword to create a “Question” dialog. The default dialog type is “Warning”.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the dialog. See [“RESOURCE_NAME”](#) on page 2288 for a complete discussion of this keyword.

TITLE

Set this keyword to a scalar string that contains the text of a title to be displayed in the dialog frame. If this keyword is not specified, the dialog has the dialog type as its title as shown in the table under [DIALOG_MESSAGE](#).

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Version History

Introduced: 5.0

See Also

[XDISPLAYFILE](#)

DIALOG_PICKFILE

The `DIALOG_PICKFILE` function allows you to interactively select a file, or multiple files, using the platform's own native graphical file selection dialog. You can also enter the name of a file that does not exist (see the description of the `WRITE` keyword). When the `DIRECTORY` keyword is set, you can select a directory (or create and select a directory on Windows) using the Browse for Folder dialog.

Entering File Names Manually

Once the file selection dialog has been displayed, you can enter a file name manually by typing the name in the **File name** field (Windows) or **Selection** field (Motif).

Windows

On Microsoft Windows systems, file names can be entered as they appear in the Windows Explorer, including spaces. If the `MULTIPLE_FILES` keyword is set, each file name should be surrounded by double quotes, and the file names should be separated by spaces. For example, to select two files named `First File.txt` and `Second File.txt`, you would enter the following in the **File name** field:

```
"First File.txt" "Second File.txt"
```

Motif

To enter a file name that includes a space character on a Motif system, escape the space character with the `"\"` character. Similarly, if the file name includes the `"\"` character, escape the literal `"\"` with another `"\"` character. If the `MULTIPLE_FILES` keyword is set, file names should be separated by spaces. For example, to select two files named `First File.txt` and `Back\slash File.txt`, you would enter the following in the **Selection** field:

```
First\ File.txt Back\\slash\ File.txt
```

Syntax

```
Result = DIALOG_PICKFILE( [, DEFAULT_EXTENSION=string]
  [, DIRECTORY] [, DIALOG_PARENT=widget_id]
  [, DISPLAY_NAME=string] [, FILE=string] [, FILTER=string/string array]
  [, FIX_FILTER] [, GET_PATH=variable] [, GROUP=widget_id]
  [, MULTIPLE_FILES] [, MUST_EXIST] [, OVERWRITE_PROMPT]
  [, PATH=string] [, READ | , WRITE] [, RESOURCE_NAME=string]
  [, TITLE=string] )
```

Return Value

When the `DIRECTORY` keyword is not specified, `DIALOG_PICKFILE` returns a string array that contains the full path name of the selected file or files. If no file is selected, `DIALOG_PICKFILE` returns a null string.

When the `DIRECTORY` keyword is specified, `DIALOG_PICKFILE` returns a string array that contains the full path of the selected directory.

Keywords

DEFAULT_EXTENSION

Set this keyword to a scalar string representing the default extension to be appended onto the returned file name or names. If the returned file name already has an extension, then the value set for this keyword is not appended. The string value set for this keyword should not include a period (.).

Note

This keyword only applies to file names typed into the dialog. This keyword does not apply to files selected within the dialog.

DIALOG_PARENT

Set this keyword to the widget ID of a widget to be used as the parent of this dialog.

DIRECTORY

Set this keyword to display only the existing directories in the directory specified by the `PATH` keyword. If a path is not specified, setting this keyword shows directories in the current directory. The Browse for Folder dialog is opened to the directory specified by the path definition. Individual files are not displayed. The return value contains the path of the directory selected, or directory created and selected by the user.

DISPLAY_NAME

Set this keyword equal to a string that specifies the name of the X Windows display on which the dialog should be displayed. This keyword is ignored on Microsoft Windows platforms.

FILE

Set this keyword to a scalar string that contains the name of the initial file selection. This keyword is useful for specifying a default filename.

On Windows, this keyword also has the effect of filtering the file list if a wildcard is used, but this keyword should be used to specify a specific filename. To list only files of a certain type, use the `FILTER` keyword.

FILTER

Set this keyword to a string value or an array of strings specifying the file types to be displayed in the file list. This keyword is used to reduce the number of files displayed in the file list. The user can modify the filter unless the `FIX_FILTER` keyword is set. If the value contains a vector of strings, multiple filters are used to filter the files. The filter `*.*` is automatically added to any filter you specify.

For example, to display only files of type `.jpg`, `.tif`, or `.png` in the file selection window, you could use the following code:

```
filters = ['*.jpg', '*.tif', '*.png']
file = DIALOG_PICKFILE(/READ, FILTER = filters)
```

The filter list shown above is displayed as five options in the dialog:

```
*.jpg, *.tif, *.png
*.jpg
*.tif
*.png
*.*
```

Multiple file types can be included in a single filter by providing a semicolon-separated list of types within the string. For example, to account for different extensions used for similar file types, you could use the following code:

```
filters = ['*.jpg;*.jpeg', '*.tif;*.tiff', '*.png']
file = DIALOG_PICKFILE(/READ, FILTER = filters)
```

The filter list shown above is displayed as five options in the dialog:

```
*.jpg, *.jpeg, *.tif, *.tiff, *.png
*.jpg, *.jpeg
*.tif, *.tiff
*.png
*.*
```

The `FILTER` keyword can optionally be set equal to an $n \times 2$ array. In this case, the first vector contains the file types, and the second vector contains a list of descriptions that are displayed in the dialog in place of the file type strings. For example:

```

filters = [['*.jpg;*.jpeg', '*.tif;*.tiff', '*.png', '*.*'], $
          ['JPEG', 'TIFF', 'Bitmap', 'All files']]
file = DIALOG_PICKFILE(/READ, FILTER = filters)

```

The filter list shown above is displayed as four options in the dialog:

```

JPEG
TIFF
Bitmap
All files

```

Note

When an $n \times 2$ array is provided, the `*.*` filter is not automatically added to the list. If you want this filter included in the list, you must include it explicitly.

Under Microsoft Windows, the user cannot modify the displayed filter. The user can enter a filter string in the Filename field to interactively update the list of files displayed. For example, entering `*.pro` in the Filename field causes only `.pro` files to be displayed.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

FIX_FILTER

When this keyword is set, only files that satisfy the filter can be selected. The user has no ability to modify the filter and the filter is not shown.

Under Microsoft Windows, the filter string can never be modified, but the user can enter a filter string in the Filename field to interactively update the list of files displayed even if `FIX_FILTER` is set.

GET_PATH

Set this keyword to a named variable in which the path of the selection is returned.

GROUP

This keyword is obsolete and has been replaced by the `DIALOG_PARENT` keyword. Code that uses the `GROUP` keyword will continue to function as before, but we suggest that all new code use `DIALOG_PARENT`.

MULTIPLE_FILES

Set this keyword to allow for multiple file selection in the file-selection dialog. When you set this keyword, the user can select multiple files using the platform-specific selection method. The currently selected files appear in the selection text field of the dialog. With this keyword set, `DIALOG_PICKFILE` can return a string array that contains the full path name of the selected file or files.

MUST_EXIST

Set this keyword to allow only files that already exist to be selected.

OVERWRITE_PROMPT

If this keyword is set along with the `WRITE` keyword and the user selects a file that already exists, then a dialog will be displayed asking if the user wants to replace the existing file or not. For multiple selections, the user is prompted separately for each file. If the user selects **No** the file selection dialog is displayed again; if the user selects **Yes** then the selection is allowed. This keyword has no effect unless the `WRITE` keyword is also set.

PATH

Set this keyword to a string that contains the initial path from which to select files. If this keyword is not set, the current working directory is used.

READ

Set this keyword to make the title of the dialog “Select File to Read”.

RESOURCE_NAME

Set this keyword equal to a string containing an X Window System resource name to be applied to the dialog.

TITLE

Set this keyword to a scalar string to be used for the dialog title. If it is not specified, the default title is “Please Select a File”.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

WRITE

Set this keyword to make the title of the dialog “Select File to Write”.

Examples

Create a `DIALOG_PICKFILE` dialog that lets users select only files with the extension ‘pro’. Use the ‘Select File to Read’ title and store the name of the selected file in the variable `file`. Enter:

```
file = DIALOG_PICKFILE(/READ, FILTER = '*.pro')
```

Create a `DIALOG_PICKFILE` dialog that allows the user to select a DICOM file. The following code sets the initial path to the `data` directory contained in the `examples` directory of the IDL installation directory.

```
; Allow the user to select a DICOM file.
sFile = DIALOG_PICKFILE(PATH='examples\data', $
    TITLE='Select DICOM Patient File', FILTER='*.dcm')
```

When the `DIRECTORY` keyword is not set, as in the previous example, the user may select one or more files of the `FILTER` type from the directory specified by `PATH`. The `TITLE` keyword defines the dialog title when `DIRECTORY` is not specified as shown in a subsection of the dialog in the following figure.

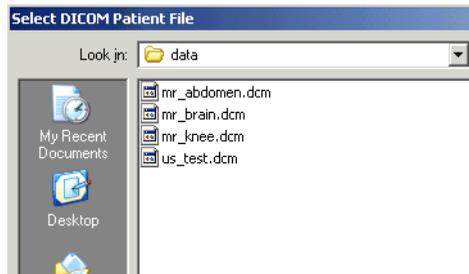


Figure 3-18: Filtered File Selection

Use the `DIRECTORY` keyword to `DIALOG_PICKFILE` to open the “Browse for Folder” dialog. The `PATH` keyword specifies the initial directory selection and the `TITLE` keyword provides additional instruction to the user in the body of the dialog.

```
sFolder = DIALOG_PICKFILE(PATH='examples\data', /DIRECTORY, $
    TITLE="Choose directory containing DICOM patient data.")
```

The following figure shows the results of issuing this statement on the Windows platform.

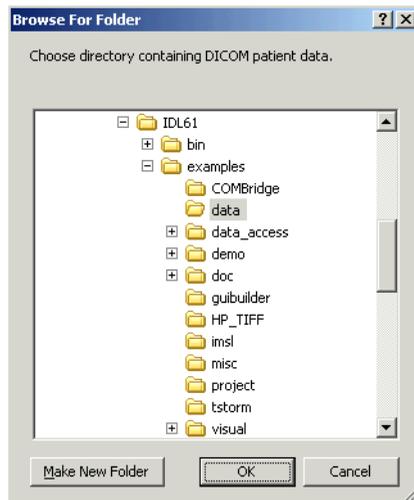


Figure 3-19: `DIALOG_PICKFILE` with `DIRECTORY` Keyword Set

Version History

Introduced: 5.0

`DEFAULT_EXTENSION` and `OVERWRITE_PROMPT` keywords added: 6.0

Browse for Folder dialog updated to enable horizontal scrolling (when needed), and to allow for new folder creation on Windows: 6.1

See Also

[FILEPATH](#)

DIALOG_PRINTERSETUP

The `DIALOG_PRINTERSETUP` function opens a native dialog for setting the applicable properties for a particular printer.

Syntax

```
Result = DIALOG_PRINTERSETUP( [PrintDestination]  
    [, DIALOG_PARENT=widget_id] [, DISPLAY_NAME=string]  
    [, RESOURCE_NAME=string] [, TITLE=string] )
```

Return Value

`DIALOG_PRINTERSETUP` returns a nonzero value if the user pressed the “OK” button in the dialog, or zero otherwise. You can programmatically begin printing based on the value returned by this function.

Arguments

PrintDestination

An instance of the `IDLgrPrinter` object for which setup properties are to be set. If no *PrintDestination* is specified, the printer used by the IDL Direct Graphics `printer` device is modified.

Keywords

DIALOG_PARENT

Set this keyword to the widget ID of a widget to be used as the parent of this dialog.

DISPLAY_NAME

Set this keyword equal to a string indicating the name of the X Windows display on which the dialog is to appear. This keyword is ignored if the `DIALOG_PARENT` keyword is specified. This keyword is also ignored on Microsoft Windows platforms.

RESOURCE_NAME

Set this keyword equal to a string containing an X Window System resource name to be applied to the dialog.

TITLE

Set this keyword equal to a string to be displayed on the dialog frame. This keyword is ignored on Microsoft Windows platforms.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Version History

Introduced: 5.0

See Also

[DIALOG_PRINTJOB](#), “The Printer Device” on page 4301

DIALOG_PRINTJOB

The `DIALOG_PRINTJOB` function opens a native dialog that allows you to set parameters for a printing job (number of copies to print, for example).

Syntax

```
Result = DIALOG_PRINTJOB( [PrintDestination]  
    [, DIALOG_PARENT=widget_id] [, DISPLAY_NAME=string]  
    [, RESOURCE_NAME=string] [, TITLE=string] )
```

Return Value

`DIALOG_PRINTJOB` returns a nonzero value if the user pressed the “OK” button in the dialog, or zero otherwise. You can use the result of this function to programmatically begin printing.

Arguments

PrintDestination

An instance of the `IDLgrPrinter` object for which a printing job is to be initiated. If no *PrintDestination* is specified, the printer used by the IDL Direct Graphics `printer` device is modified.

Keywords

DIALOG_PARENT

Set this keyword to the widget ID of a widget to be used as the parent of this dialog.

DISPLAY_NAME

Set this keyword to a string indicating the name of the X Windows display on which the dialog is to appear. This keyword is ignored if the `DIALOG_PARENT` keyword is specified. This keyword is also ignored on Microsoft Windows platforms.

RESOURCE_NAME

Set this keyword to a string containing an X Window System resource name to be applied to the dialog.

TITLE

Set this keyword to a string to be displayed on the dialog frame. This keyword is ignored on Microsoft Windows platforms.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Version History

Introduced: 5.0

See Also

[DIALOG_PRINTERSETUP](#), “The Printer Device” on page 4301

DIALOG_READ_IMAGE

The `DIALOG_READ_IMAGE` function is a graphical interface used for reading image files. The interface is created as a modal dialog with an optional parent widget.

Syntax

```
Result = DIALOG_READ_IMAGE ( [Filename] [, BLUE=variable]  
    [, DIALOG_PARENT=widget_id] [, FILE=variable] [, FILTER_TYPE=string]  
    [, /FIX_FILTER] [, GET_PATH=variable] [, GREEN=variable]  
    [, IMAGE=variable] [, PATH=string] [, QUERY=variable] [, RED=variable]  
    [, TITLE=string] )
```

Return Value

This function returns 1 if the “Open” button was clicked, and 0 if the “Cancel” button was clicked.

Arguments

Filename

An optional scalar string containing the full pathname of the file to be highlighted.

Keywords

BLUE

Set this keyword to a named variable that will contain the blue channel vector (if any).

DIALOG_PARENT

The widget ID of a widget that calls `DIALOG_READ_IMAGE`. When this ID is specified, a death of the caller results in the death of the `DIALOG_READ_IMAGE` dialog. If `DIALOG_PARENT` is not specified, then the interface is created as a modal, top-level widget.

FILE

Set this keyword to a named variable that will contain the selected filename with full path when the dialog is created.

FILTER_TYPE

Set this keyword to a scalar string containing the format type the dialog filter should begin with. The default is “Image Files”. The user cannot modify the filter if the `FIX_FILTER` keyword is set. Valid values are obtained from the list of supported image types returned from `QUERY_IMAGE`. In addition, there is also the “All Files” type. If set to “All Files”, queries will only happen on filename clicks, making the dialog much more efficient.

Example:

```
FILTER=' .jpg, .tiff'
```

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

FIX_FILTER

When this keyword is set, only files that satisfy the filter can be selected. The user has no ability to modify the filter.

GET_PATH

Set this keyword to a named variable in which the path of the selection is returned.

GREEN

Set this keyword to a named variable that will contain the green channel vector (if any).

IMAGE

Set this keyword to a named variable that will contain the image array read. If Cancel was clicked, no action is taken.

PATH

Set this keyword to a string that contains the initial path from which to select files. If this keyword is not set, the current working directory is used.

QUERY

Set this keyword to a named variable that will return the QUERY_IMAGE structure associated with the returned image. If the “Cancel” button was pressed, the variable set to this keyword is not changed. If an error occurred during the read, the FILENAME field of the structure will be a null string.

RED

Set this keyword to a named variable that will contain the red channel vector (if any).

TITLE

Set this keyword to a scalar string to be used for the dialog title. If it is not specified, the default title is “Select Image File”.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Version History

Introduced: 5.3

See Also

[DIALOG_WRITE_IMAGE](#)

DIALOG_WRITE_IMAGE

The `DIALOG_WRITE_IMAGE` function is a graphical user interface used for writing image files. The interface is created as a modal dialog with an optional parent widget.

Syntax

```
Result = DIALOG_WRITE_IMAGE ( Image [, R, G, B]
    [, DIALOG_PARENT=widget_id] [, FILE=string] [, /FIX_TYPE] [, /NOWRITE]
    [, OPTIONS=variable] [, PATH=string] [, TITLE=string] [, TYPE=variable]
    [, /WARN_EXIST] )
```

Return Value

This routine returns 1 if the “Save” button was clicked, and 0 if the “Cancel” button was clicked.

Arguments

Image

The array to be written to the image file.

R, G, B

These are optional arguments defining the Red, Green, and Blue color tables to be associated with the image array.

Keywords

DIALOG_PARENT

The widget ID of a widget that calls `DIALOG_WRITE_IMAGE`. When this ID is specified, a death of the caller results in the death of the `DIALOG_WRITE_IMAGE` dialog. If `DIALOG_PARENT` is not specified, then the interface is created as a modal, top-level widget.

FILE

Set this keyword to a scalar string that contains the name of the initial file selection. This keyword is useful for specifying a default filename.

FIX_TYPE

When this keyword is set, only files that satisfy the type can be selected. The user has no ability to modify the type.

NOWRITE

Set this keyword to prevent the dialog from writing the file when “Save” is clicked. No data conversions will take place when the save type is chosen.

OPTIONS

Set this keyword to a named variable to contain a structure of the chosen options by the user, including the filename and image type chosen.

PATH

Set this keyword to a string that contains the initial path from which to select files. If this keyword is not set, the current working directory is used.

TITLE

Set this keyword to a scalar string to be used for the dialog title. If it is not specified, the default title is “Save Image File”.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

TYPE

Set this keyword to a scalar string containing the format type the “Save as type” field should begin with. The default is “TIFF”. The user can modify the type unless the `FIX_TYPE` keyword is set. Valid values are obtained from the list of supported image types returned from `QUERY_IMAGE`. The “Save as type” field will reflect the type of the selected file (if one is selected).

WARN_EXIST

Set this keyword to produce a question dialog if the user selects a file that already exists. The default is to quietly overwrite the file.

Version History

Introduced: 5.3

See Also

[DIALOG_READ_IMAGE](#)

DIGITAL_FILTER

The `DIGITAL_FILTER` function returns the coefficients of a non-recursive, digital filter for evenly spaced data points. Frequencies are expressed in terms of the Nyquist frequency, $1/2T$, where T is the time between data samples. Highpass, lowpass, bandpass and bandstop filters may be constructed with this function.

This routine is written in the IDL language. Its source code can be found in the file `digital_filter.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = DIGITAL_FILTER( Flow, Fhigh, A, Nterms [, /DOUBLE] )
```

Return Value

This function returns a vector of coefficients with $(2 \times Nterms + 1)$ elements.

Arguments

Flow

The lower frequency of the filter as a fraction of the Nyquist frequency

Fhigh

The upper frequency of the filter as a fraction of the Nyquist frequency. The following conditions are necessary for various types of filters:

- No Filtering: $Flow = 0, Fhigh = 1$
- Low Pass: $Flow = 0, 0 < Fhigh < 1$
- High Pass: $0 < Flow < 1, Fhigh = 1$
- Band Pass: $0 < Flow < Fhigh < 1$
- Band Stop: $0 < Fhigh < Flow < 1$

A

The filter power relative to the Gibbs phenomenon wiggles in decibels. 50 is a good choice.

Nterms

The number of terms used to construct the filter.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision result. Set `DOUBLE=0` to use single-precision for computations and to return a single-precision result. The default is `/DOUBLE` if the Flow input is double precision, otherwise the default is `DOUBLE=0`.

Examples

```
; Get coefficients:
Coeff = DIGITAL_FILTER(Flow, Fhigh, A, Nterms)
; Apply the filter:
Yout = CONVOL(Yin, Coeff)
```

Version History

Introduced: Original

DOUBLE keyword added: 5.6

See Also

[CONVOL](#), [LEEFILT](#), [MEDIAN](#), [SMOOTH](#)

DILATE

The DILATE function implements the morphologic dilation operator on both binary and grayscale images. For details on using DILATE, see [“Using DILATE”](#) on page 569.

Using DILATE

Mathematical morphology is a method of processing digital images on the basis of shape. A discussion of this topic is beyond the scope of this manual. A suggested reference is: Haralick, Sternberg, and Zhuang, “Image Analysis Using Mathematical Morphology,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, No. 4, July, 1987, pp. 532-550. Much of this discussion is taken from that article.

Briefly, the DILATE function returns the dilation of *Image* by the structuring element *Structure*. This operator is commonly known as “fill”, “expand”, or “grow.” It can be used to fill “holes” of a size equal to or smaller than the structuring element.

Used with binary images, where each pixel is either 1 or 0, dilation is similar to convolution. Over each pixel of the image, the origin of the structuring element is overlaid. If the image pixel is nonzero, each pixel of the structuring element is added to the result using the “or” operator.

Letting $A \oplus B$ represent the dilation of an image A by structuring element B , dilation can be defined as:

$$C = A \oplus B = \bigcup_{b \in B} (A)_b$$

where $(A)_b$ represents the translation of A by b . Intuitively, for each nonzero element $b_{i,j}$ of B , A is translated by i,j and summed into C using the “or” operator. For example:

$$\begin{array}{r} 0100 \\ 0100 \\ 0110 \oplus 11 = 0111 \\ 1000 \\ 0000 \end{array} \quad \begin{array}{r} 0110 \\ 0110 \\ 1100 \\ 0000 \end{array}$$

In this example, the origin of the structuring element is at (0,0).

Used with grayscale images, which are always converted to byte type, the DILATE function is accomplished by taking the maximum of a set of sums. It can be used to conveniently implement the neighborhood maximum operator with the shape of the neighborhood given by the structuring element.

Openings and Closings

The *opening* of image B by structuring element K is defined as $(B \otimes K) \oplus K$. The *closing* of image B by K is defined as $(B \oplus K) \otimes K$ where the \otimes symbol represents the erosion operator implemented by the IDL ERODE function.

As stated by Haralick *et al*, the result of iteratively applied dilations and erosions is an elimination of specific image detail smaller than the structuring element without the global geometric distortion of unsuppressed features. For example, opening an image with a disk structuring element smooths the contour, breaks narrow isthmuses, and eliminates small islands and sharp peaks or capes.

Closing an image with a disk structuring element smooths the contours, fuses narrow breaks and long thin gulfs, eliminates small holes, and fills gaps on the contours.

Note

MORPH_OPEN and MORPH_CLOSE can also be used to perform these tasks.

Syntax

```
Result = DILATE( Image, Structure [, X0 [, Y0 [, Z0]]] [, /CONSTRAINED
  [, BACKGROUND=value]] [, /GRAY [, /PRESERVE_TYPE |, /UINT |
  , /ULONG]] [, VALUES=array] )
```

Return Value

The DILATE function returns the dilation of *Image* by the structuring element *Structure*.

Arguments

Image

A one-, two-, or three-dimensional array upon which the dilation is to be performed. If the parameter is not of byte type, a temporary byte copy is obtained. If neither of

the keywords `GRAY` or `VALUES` is present, the image is treated as a binary image with all nonzero pixels considered as 1.

Structure

A one-, two-, or three-dimensional array that represents the structuring element. Elements are interpreted as binary: values are either zero or nonzero. This argument must have the same number of dimensions as *Image*.

X_0, Y_0, Z_0

Optional parameters specifying the one-, two-, or three-dimensional coordinate of the structuring element's origin. If omitted, the origin is set to the center, $([N_x/2], [N_y/2], [N_z/2])$, where N_x , N_y , and N_z are the dimensions of the structuring element array. The origin need not be within the structuring element.

Keywords

BACKGROUND

Set this keyword to the pixel value that is to be considered the background when dilation is being performed in constrained mode. The default value is 0.

CONSTRAINED

If this keyword is set and grayscale dilation has been selected, the dilation algorithm will operate in constrained mode. In this mode, a pixel is set to the value determined by normal grayscale dilation rules in the output image only if the current value destination pixel value matches the `BACKGROUND` pixel value. Once a pixel in the output image has been set to a value other than the `BACKGROUND` value, it cannot change.

GRAY

Set this keyword to perform grayscale, rather than binary, dilation. The nonzero elements of the *Structure* parameter determine the shape of the structuring element (neighborhood). If `VALUES` is not present, all elements of the structuring element are 0, yielding the neighborhood maximum operator.

PRESERVE_TYPE

Set this keyword to return the same type as the input array. This keyword only applies if the `GRAY` keyword is set.

UINT

Set this keyword to return an unsigned integer array. This keyword only applies if the GRAY keyword is set.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword only applies if the GRAY keyword is set.

VALUES

An array with the same dimensions as *Structure* providing the values of the structuring element. The presence of this parameter implies grayscale dilation. Each pixel of the result is the maximum of the sum of the corresponding elements of VALUE and the *Image* pixel value. If the resulting sum is greater than 255, the return value is 255.

Examples

Example 1

This example thresholds a gray scale image at the value of 100, producing a binary image. The result is then “opened” with a 3 pixel by 3 pixel square shape operator, using the DILATE and ERODE operators. The effect is to remove holes, islands, and peninsula smaller than the shape operator:

```

; Threshold and make binary image:
B = A GE 100

; Create the shape operator:
S = REPLICATE(1, 3, 3)

; "Opening" operator:
C = DILATE(ERODE(B, S), S)

; Show the result:
TVSCL, C

```

Example 2

For grayscale images, DILATE takes the neighborhood maximum, where the shape of the neighborhood is given by the structuring element. Elements for which the structuring element extends off the array are indeterminate. For example, assume you have the following image and structuring element:

```
image = BYTE([2,1,3,3,3,3,1,2])
s = [1,1]
```

If the origin of the structuring element is not specified in the call to DILATE, the origin defaults to one half the width of the structuring element, which is 1 in this case. Therefore, for the first element in the image array, the structuring element is aligned with the image as depicted below:

```
  [2,1,3,3,3,3,1,2]
   ↑
  [1,1]
```

This will cause an indeterminate value for the first element in the DILATE result. If edge values are important, you must pad the image with as many zeros as there are elements in the structuring element that extend off the array, in all dimensions. In this case, you would need to pad the image with a single leading zero. If the structuring element were `s=[1,1,1,1]`, and you specified an origin of 2, the structuring element would align with the image as follows:

```
  [2,1,3,3,3,3,1,2]
   ↑           ↑
 [1,1,1,1]    [1,1,1,1]
```

Therefore, you would need to pad the image with at least two leading zeros and at least one trailing zero. You would then perform the dilation operation on the padded image, and remove the padding from the result.

The following code illustrates this method:

```
image = BYTE([2,1,3,3,3,3,1,2])
s = [1,1] ; Structuring element
PRINT, 'Image: '
PRINT, image

PRINT, 'Dilation using no padding: '
PRINT, DILATE(image, s, /GRAY)

result = DILATE([0, image], s, /GRAY)
PRINT, 'Dilation using padding: '
PRINT, result[1:N_ELEMENTS(image)]
```

IDL prints:

```
Image:
  2  1  3  3  3  3  1  2
Dilation using no padding:
  1  3  3  3  3  3  2  2
Dilation using padding:
  2  3  3  3  3  3  2  2
```

Version History

Introduced: Pre 4.0

See Also

[ERODE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#), [MORPH_GRADIENT](#),
[MORPH_HITORMISS](#), [MORPH_OPEN](#), [MORPH_THIN](#), [MORPH_TOPHAT](#)

DINDGEN

The DINDGEN function creates a double-precision, floating-point array with the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = DINDGEN(D_1 [, \dots, D_8])$$

Return Value

Returns a double-precision, floating-point array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create D, a 100-element, double-precision, floating-point array with each element set to the value of its subscript, enter:

```
D = DINDGEN(100)
```

Version History

Introduced: Original

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [FINDGEN](#), [INDGEN](#), [LINDGEN](#),
[SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

DISSOLVE

The DISSOLVE procedure provides a digital “dissolve” effect for images. The routine copies pixels from the image (arranged into square tiles) to the display in pseudo-random order. This routine is written in the IDL language. Its source code can be found in the file `dissolve.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
DISSOLVE, Image [, DELAY=seconds] [, /ORDER] [, SIZ=pixels] [, X0, Y0=pixels,  
pixels]
```

Arguments

Image

The image to be displayed. It is assumed that the image is already scaled. Byte-valued images display most rapidly.

Keywords

DELAY

The wait between displaying tiles. The default is 0.01 second.

ORDER

The Image display order: 0 = bottom up (the default), 1 = top-down.

SIZ

Size of square tile. The default is 32 x 32 pixels.

X0, Y0

The X and Y offsets of the lower-left corner of the image on screen, in pixels.

Examples

Display an image using 16 x 16 pixel tiles:

```
DISSOLVE, DIST(200), SIZ=16
```

Version History

Introduced: Pre 4.0

See Also

[ERASE, TV](#)

DIST

The DIST function creates an array in which each array element value is proportional to its frequency. This array may be used for a variety of purposes, including frequency-domain filtering.

This routine is written in the IDL language. Its source code can be found in the file `dist.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$Result = DIST(N [, M])$$

Return Value

Returns a rectangular array in which the value of each element is proportional to its frequency.

Arguments

N

The number of columns in the resulting array.

M

The number of rows in the resulting array. If *M* is omitted, the resulting array will be *N* by *N*.

Keywords

None.

Examples

```
; Display the results of DIST as an image:  
TVSCL, DIST(100)
```

Version History

Introduced: Original

See Also

[FFT](#)

DISTANCE_MEASURE

The `DISTANCE_MEASURE` function computes the pairwise distance between a set of items or observations. The `DISTANCE_MEASURE` function is designed to be used with the `CLUSTER_TREE` function.

This routine is written in the IDL language. Its source code can be found in the file `distance_measure.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = DISTANCE_MEASURE( Array [, /DOUBLE] [, /MATRIX]  
    [, MEASURE=value] [, POWER_MEASURE=value] )
```

Return Value

The *Result* is a vector of $m*(m-1)/2$ elements containing the distance matrix in compact form. Given a distance between two items, $D_{i,j}$, the distances within *Result* are returned in the order: $[D_{0,1}, D_{0,2}, \dots, D_{0,m-1}, D_{1,2}, \dots, D_{m-2,m-1}]$.

If `MATRIX` is set, then the *Result* is an m -by- m symmetric array containing the full distance matrix, with zeroes down the diagonal.

Arguments

Array

An n -by- m array representing the coordinates (in an n -dimensional space) of m items. For example, a set of m points in a two-dimensional Cartesian space would be passed in as a 2-by- m array.

Keywords

DOUBLE

Set this keyword to perform computations using double-precision arithmetic and to return a double-precision result. Set `DOUBLE=0` to use single-precision for computations and to return a single-precision result. The default is `/DOUBLE` if *Array* is double precision, otherwise the default is `DOUBLE=0`.

MATRIX

Set this keyword to return the distance matrix as an m -by- m symmetric array. If this keyword is not set then the distance matrix is returned in compact vector form.

MEASURE

Set this keyword to an integer giving the distance measure (the metric) to use. Possible values are:

Value	Distance
0	(Default): Euclidean distance. The Euclidean distance is defined as: $\sqrt{\sum x_i - y_i ^2}$
1	CityBlock (Manhattan) distance. The CityBlock distance is defined as: $\sum x_i - y_i $
2	Chebyshev distance. The Chebyshev distance is defined as: $\text{Max}(x_i - y_i)$
3	Correlative distance. The correlative distance, where r is the correlation coefficient between two items, is defined as: $\sqrt{(1 - r)/2}$
4	Percent disagreement. This distance is useful for categorical data and is defined as: $(\text{Number of } x_i \neq y_i)/n$

Table 4: MEASURE Values

Note

This keyword is ignored if POWER_MEASURE is set.

POWER_MEASURE

Set this keyword to a scalar or a two-element vector giving the parameters p and r to be used in the power distance, defined as:

$$(\sum |x_i - y_i|^p)^{1/r}$$

If POWER_MEASURE is a scalar then the same value is used for both p and r (this is also known as the Minkowski distance).

Note

POWER_MEASURE=1 is the same as the CityBlock distance, while POWER_MEASURE=2 is the same as Euclidean distance.

Example

```

; Given a set of points in two-dimensional space.

DATA = [ $
[1, 1], $
[1, 3], $
[2, 2.2], $
[4, 1.75], $
[4, 4], $
[5, 1], $
[5.5, 3]]

; Compute the Euclidean distance between each point.
DISTANCE = DISTANCE_MEASURE(data)

i1 = [0,0,0,0,0,0, 1,1,1,1,1, 2,2,2,2, 3,3,3, 4,4, 5]
i2 = [1,2,3,4,5,6, 2,3,4,5,6, 3,4,5,6, 4,5,6, 5,6, 6]
PRINT, 'Item# Item# Distance'
PRINT, TRANSPOSE([[i1],[i2],[distance]]), $
    FORMAT = '(I3, I7, F10.2)'

PLOT, data[0,*], data[1,*], PSYM = 6, SYMSIZE = 2, $
    XRANGE = [0,6], YRANGE = [0,5], $
    TITLE='Distance between each point'

FOR i = 0,N_ELEMENTS(distance)-1 DO $
    PLOTS, data[*], [i1[i], i2[i]], linestyle = 1

; Midpoint of each pair.

AVG = 0.5*(data[*], i1] + data[*], i2])

```

```
XYOUTS, avg[0,*], avg[1,*], ALIGN = 0.5, $  
STRTRIM(STRING(distance, format = '(F7.2)'),2)
```

When this code is run, IDL prints:

Item#	Item#	Distance
0	1	2.00
0	2	1.56
0	3	3.09
0	4	4.24
0	5	4.00
0	6	4.92
1	2	1.28
1	3	3.25
1	4	3.16
1	5	4.47
1	6	4.50
2	3	2.05
2	4	2.69
2	5	3.23
2	6	3.59
3	4	2.25
3	5	1.25
3	6	1.95
4	5	3.16
4	6	1.80
5	6	2.06

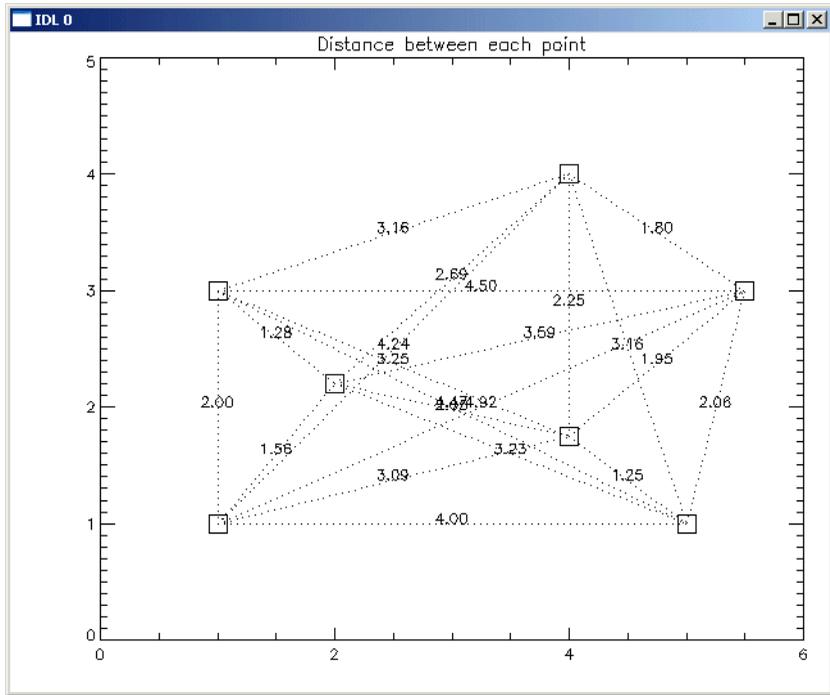


Figure 3-20: Distance Between Each Point Using `DISTANCE_MEASURE`

Version History

Introduced: 6.1

See Also

[CLUSTER_TREE](#), [DENDROGRAM](#), [DENDRO_PLOT](#)

DLM_LOAD

Normally, IDL system routines that reside in dynamically loadable modules (DLMs) are automatically loaded on demand when a routine from a DLM is called. The `DLM_LOAD` procedure can be used to explicitly cause a DLM to be loaded.

Syntax

```
DLM_LOAD, DLMNameStr1 [, DLMNameStr2, ..., DLMNameStrn]
```

Arguments

*DLMNameStr*_{*n*}

A string giving the name of the DLM to be loaded. `DLM_LOAD` causes each named DLM to be immediately loaded.

Keywords

None

Example

Force the JPEG DLM to be loaded:

```
DLM_LOAD, 'jpeg'
```

IDL prints:

```
% Loaded DLM: JPEG.
```

Version History

Introduced: 5.1

DLM_REGISTER

The DLM_REGISTER procedure registers a Dynamically Loadable Module (DLM) in IDL that was not registered when starting IDL. This allows you to create DLMs using the MAKE_DLL procedure and register them in your current session without having to exit and restart IDL.

Warning

DLM_REGISTER intended as a convenience to be used when testing and debugging DLMs or when running demonstration code included in the IDL distribution. It is *not* the recommended way to make Dynamically Loadable Modules known to your IDL session. By design, DLMs are registered when IDL starts; among other things, this allows programs written in the IDL language that call the routines in a DLM to be compiled before the DLM itself is loaded into IDL. The mechanism involved is described in “[Dynamically Loadable Modules](#)” in Chapter 21 of the *External Development Guide* manual; you should read and understand this section before deciding to use DLM_REGISTER.

Syntax

DLM_REGISTER, *DLMDefFilePath*₁ [, *DLMDefFilePath*₂, ..., *DLMDefFilePath*_{*n*}]

Arguments

*DLMDefFilePath*_{*n*}

The name of the DLM module definition file to read.

Keywords

None.

Version History

Introduced: 5.4

See Also

“[Dynamically Loadable Modules](#)” in Chapter 21 of the *External Development Guide* manual

DOC_LIBRARY

The `DOC_LIBRARY` procedure extracts documentation headers from one or more IDL programs (procedures or functions). This command provides a standard interface to the operating-system specific `DL_DOS` and `DL_UNIX` procedures.

The documentation header of the `.pro` file in question must have the following format:

- The first line of the documentation block contains only the characters `;` `+`, starting in column 1.
- The last line of the documentation block contains only the characters `;` `-`, starting in column 1.
- All other lines in the documentation block contain a `;` in column 1.

The file `template.pro` in the `general` subdirectory of the `examples` subdirectory of the IDL distribution contains a template for creating your own documentation headers.

This routine is supplied for users to view online documentation from their own IDL programs. Though it could be used to view documentation headers from the `lib` subdirectory of the IDL distribution, we do not recommend doing so. The documentation headers on the files in the `lib` directory are used for historical purposes—most do not contain the most current or accurate documentation for those routines. The most current documentation for IDL's built-in and library routines is found in IDL's online help system (enter `?` at the IDL prompt).

This routine is written in the IDL language. Its source code can be found in the file `doc_library.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

`DOC_LIBRARY` [, *Name*] [, `/PRINT`]

UNIX-Only keywords: [, `DIRECTORY=string`] [, `/MULTI`]

Arguments

Name

A string containing the name of the IDL routine in question. Under Windows or UNIX, *Name* can be `"*"` to get information on all routines.

Keywords

DIRECTORY

This keyword is only available on UNIX platforms.

A string containing the name of the directory to search. If omitted, the current directory and !PATH are used.

MULTI

This keyword is only available on UNIX platforms.

Set this keyword to allow printing of more than one file if the requested module exists in more than one directory.

PRINT

Set this keyword to send the output of DOC_LIBRARY to the default printer. Under UNIX, if PRINT is a string, it is interpreted as a shell command used for output with the documentation from DOC_LIBRARY providing standard input (i.e., PRINT="cat > junk").

Obsolete Keywords

The following keywords are obsolete:

- FILE
- PATH
- OUTPUTS

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

To view the documentation header for the library function DIST, enter:

```
DOC_LIBRARY, 'DIST'
```

Version History

Introduced: Original

See Also

[MK_HTML_HELP](#)

DOUBLE

The DOUBLE function converts *Expression* into a double-precision floating-point value.

Syntax

$$Result = \text{DOUBLE}(Expression[, Offset [, D_1 [, \dots, D_8]])]$$

Return Value

Returns a double-precision floating-point value or array of the specified dimensions. If *Expression* is a complex number, DOUBLE returns the real part.

Arguments

Expression

The expression to be converted to double-precision, floating-point.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as double-precision, floating-point data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid floating-point value and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Suppose that A contains the integer value 45. A double-precision, floating-point version of A can be stored in B by entering:

```
B = DOUBLE(A)
PRINT, B
```

IDL prints:

```
45.000000
```

Version History

Introduced: Original

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [FIX](#), [FLOAT](#), [LONG](#), [LONG64](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

DRAW_ROI

The DRAW_ROI procedure draws a region or group of regions to the current Direct Graphics device. The primitives used to draw each ROI are based on the TYPE property of the given IDLanROI object. The TYPE property selects between points, polylines, and filled polygons.

Syntax

```
DRAW_ROI, oROI [, /LINE_FILL] [, SPACING=value]
```

Graphics Keywords: [, CLIP=[*X₀*, *Y₀*, *X₁*, *Y₁*] [, COLOR=*value*] [, /DATA | , /DEVICE | , /NORMAL] [, LINestyle={0 | 1 | 2 | 3 | 4 | 5}] [, /NOCLIP] [, ORIENTATION=*ccw_degrees_from_horiz*] [, PSYM=*integer*{0 to 10}] [, SYMSIZE=*value*] [, /T3D] [, THICK=*value*]

Arguments

oROI

A reference to an IDLanROI object to be drawn.

Keywords

LINE_FILL

Set this keyword to indicate that polygonal regions are to be filled with parallel lines, rather than using the default solid fill. When using a line fill, the thickness, linestyle, orientation, and spacing of the lines may be specified by keywords.

SPACING

The spacing, in centimeters, between the parallel lines used to fill polygons.

Graphics Keywords Accepted

CLIP, COLOR, DATA, DEVICE, LINSTYLE, NOCLIP, NORMAL, ORIENTATION, PSYM, SYMSIZE, T3D, THICK

Examples

The following example displays an image and collects data for a region of interest. The resulting ROI is displayed as a filled polygon.

```

PRO roi_ex
; Load and display an image.
img=READ_DICOM(FILEPATH('mr_knee.dcm',SUBDIR=['examples','data']))
TV, img

; Create a polygon region object.
oROI = OBJ_NEW('IDLanROI', TYPE=2)

; Print instructions.
PRINT,'To create a region:'
PRINT,' Left mouse: select points for the region.'
PRINT,' Right mouse: finish the region.'

; Collect first vertex for the region.
CURSOR, xOrig, yOrig, /UP, /DEVICE
oROI->AppendData, xOrig, yOrig
PLOTS, xOrig, yOrig, PSYM=1, /DEVICE

;Continue to collect vertices for region until right mouse button.
x1 = xOrig
y1 = yOrig
while !MOUSE.BUTTON ne 4 do begin
    x0 = x1
    y0 = y1
    CURSOR, x1, y1, /UP, /DEVICE
    PLOTS, [x0,x1], [y0,y1], /DEVICE
    oROI->AppendData, x1, y1
endwhile
PLOTS, [x1,xOrig], [y1,yOrig], /DEVICE

; Draw the the region with a line fill.
DRAW_ROI, oROI, /LINE_FILL, SPACING=0.2, ORIENTATION=45, /DEVICE
END

```

Version History

Introduced: 5.3

EFONT

The EFONT procedure provides a simple widget-based vector font editor and display. Use this procedure to read and/or modify a local copy of the file `herشل.chr`, located in the `resource/fonts` subdirectory of the main IDL directory, which contains the vector fonts used by IDL in plotting. This is a very rudimentary editor. Click the “Help” button on the EFONT main menu for more information.

This routine is written in the IDL language. Its source code can be found in the file `efont.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
EFONT [, Init_Font] [, /BLOCK] [, GROUP=widget_id]
```

Arguments

Init_Font

The initial font index, from 3 to 29. The default is 3.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have EFONT block, any earlier calls to XMANAGER must have been called with the [NO_BLOCK](#) keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

The widget ID of the widget that calls EFONT. If GROUP is set, the death of the caller results in the death of EFONT.

Version History

Introduced: Pre 4.0

See Also

[SHOWFONT](#), [XFONT](#)

EIGENQL

The EIGENQL function computes the eigenvalues and eigenvectors of an n -by- n real, symmetric array using Householder reductions and the QL method with implicit shifts.

This routine is written in the IDL language. Its source code can be found in the file `eigenql.pro` in the `lib` subdirectory of the IDL distribution.

Note

If you are working with complex inputs, instead use the `LA_EIGENQL` function.

Syntax

```
Result = EIGENQL( A [, /ABSOLUTE] [, /ASCENDING] [, /DOUBLE]
  [, EIGENVECTORS=variable] [, /OVERWRITE | , RESIDUAL=variable] )
```

Return Value

This function returns an n -element vector containing the eigenvalues.

Arguments

A

An n -by- n symmetric single- or double-precision floating-point array.

Keywords

ABSOLUTE

Set this keyword to sort the eigenvalues by their absolute value (their magnitude) rather than by their signed value.

ASCENDING

Set this keyword to return eigenvalues in ascending order (smallest to largest). If not set or set to zero, eigenvalues are returned in descending order (largest to smallest). The eigenvectors are correspondingly reordered.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EIGENVECTORS

Set this keyword equal to a named variable that will contain the computed eigenvectors in an n -by- n array. The i^{th} row of the returned array contains the i^{th} eigenvalue. If no variable is supplied, the array will not be computed.

OVERWRITE

Set this keyword to use the input array for internal storage and to overwrite its previous contents.

RESIDUAL

Use this keyword to specify a named variable that will contain the residuals for each eigenvalue/eigenvector (λ/x) pair. The residual is based on the definition $Ax - (\lambda)x = 0$ and is an array of the same size as A and the same type as *Result*. The rows of this array correspond to the residuals for each eigenvalue/eigenvector pair.

Note

If the **OVERWRITE** keyword is set, the **RESIDUAL** keyword has no effect.

Examples

```

; Define an n-by-n real, symmetric array:
A = [[ 5.0,  4.0,  0.0, -3.0], $
     [ 4.0,  5.0,  0.0, -3.0], $
     [ 0.0,  0.0,  5.0, -3.0], $
     [-3.0, -3.0, -3.0,  5.0]]

; Compute the eigenvalues and eigenvectors:
eigenvalues = EIGENQL(A, EIGENVECTORS = evecs, $
                    RESIDUAL = residual)

; Print the eigenvalues and eigenvectors:
PRINT, 'Eigenvalues: '
PRINT, eigenvalues
PRINT, 'Eigenvectors: '
PRINT, evecs

```

IDL prints:

```
Eigenvalues:
12.0915      6.18662      1.00000      0.721870

Eigenvectors:
-0.554531    -0.554531    -0.241745     0.571446
-0.342981    -0.342981     0.813186    -0.321646
 0.707107    -0.707107    -6.13503e-008 -6.46503e-008
 0.273605     0.273605     0.529422     0.754979
```

The accuracy of each eigenvalue/eigenvector (λ/x) pair may be checked by printing the residual array:

```
PRINT, residual
```

The RESIDUAL array has the same dimensions as the input array and the same type as the result. The residuals are contained in the rows of the RESIDUAL array. All residual values should be floating-point zeros.

Version History

Introduced: 5.0

See Also

[EIGENVEC](#), [LA_EIGENQL](#), [TRIQL](#)

EIGENVEC

The EIGENVEC function computes the eigenvectors of an n -by- n real, non-symmetric array using Inverse Subspace Iteration. Use ELMHES and HQR to find the eigenvalues of an n -by- n real, nonsymmetric array.

This routine is written in the IDL language. Its source code can be found in the file `eigenvec.pro` in the `lib` subdirectory of the IDL distribution.

Note

If you are working with complex inputs, instead use the LA_EIGENVEC function.

Syntax

```
Result = EIGENVEC( A, Eval [, /DOUBLE] [, ITMAX=value]  
                [, RESIDUAL=variable] )
```

Return Value

This function returns a complex array with a column dimension equal to n and a row dimension equal to the number of eigenvalues.

Arguments

A

An n -by- n nonsymmetric, single- or double-precision floating-point array.

EVAL

An n -element complex vector of eigenvalues.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ITMAX

The maximum number of iterations allowed in the computation of each eigenvector. The default value is 4.

RESIDUAL

Use this keyword to specify a named variable that will contain the residuals for each eigenvalue/eigenvector (λ/x) pair. The residual is based on the definition $Ax - \lambda x = 0$ and is an array of the same size and type as that returned by the function. The rows of this array correspond to the residuals for each eigenvalue/eigenvector pair.

Examples

```

; Define an n-by-n real, nonsymmetric array:
A = [[1.0, -2.0, -4.0,  1.0], $
      [0.0, -2.0,  3.0,  4.0], $
      [2.0, -6.0, -1.0,  4.0], $
      [3.0, -3.0,  1.0, -2.0]]

; Compute the eigenvalues of A using double-precision
; complex arithmetic and print the result:
eval = HQR(ELMHES(A), /DOUBLE)
PRINT, 'Eigenvalues: '
PRINT, eval

evec = EIGENVEC(A, eval, RESIDUAL = residual)

; Eigenvectors are not generally unique.
; Multiply each eigenvector by a complex scaling
; factor to force the initial term to be real.
; This normalization ensures a unique solution.
FOR i=0,3 DO evec[* ,i] *= ABS(evec[0,i])/evec[0,i]

PRINT, 'Eigenvectors:'
PRINT, evec, FORMAT='(4("(",f8.5,",",f8.5,") " ))'

PRINT, 'Residuals:'
FOR i=0,3 DO print, A ## evec[* ,i] - eval[i]*evec[* ,i], $
  FORMAT = '(4(" ",g9.2,",",g9.2,") " )'

```

IDL prints:

```

Eigenvalues:
(0.26366252,-6.1925900)(0.26366252,6.1925900)
(-4.9384492,0.0000000)(0.411124050.0000000)
% Compiled module: EIGENVEC.
Eigenvectors:
( 0.42919, 0.00000) (-0.32241, 0.41235)
( 0.29827, 0.54013) ( 0.23222, 0.32739)
( 0.42919, 0.00000) (-0.32241,-0.41235)
( 0.29827,-0.54013) ( 0.23222,-0.32739)
( 0.54966, 0.00000) ( 0.18401, 0.00000)

```

```

( 0.58125, 0.00000) (-0.57111, 0.00000)
( 0.79297, 0.00000) ( 0.50289, 0.00000)
(-0.04962, 0.00000) ( 0.34035, 0.00000)
Residuals:
( 3.1e-008, 2.9e-008) (-5.1e-008, 7.8e-009)
(-1.5e-008, 5.9e-008) (-5.5e-009, 3.9e-008)
( 3.1e-008,-2.9e-008) (-5.1e-008,-7.8e-009)
(-1.5e-008,-5.9e-008) (-5.5e-009,-3.9e-008)
(-1.1e-007,      0.00) ( 7.9e-009,      0.00)
( 2.5e-008,      0.00) (-2.4e-008,      0.00)
(-5.0e-008,      0.00) (-6.1e-008,      0.00)
( 1.5e-010,      0.00) (-4.2e-008,      0.00)

```

You can check the accuracy of each eigenvalue/eigenvector (λ/x) pair by printing the residual array. All residual values should be floating-point zeros.

Note

Different machines may produce slightly different results.

Version History

Introduced: 4.0

See Also

[ELMHES](#), [HQR](#), [LA_EIGENVEC](#), [TRIQL](#), [TRIRED](#)

ELMHES

The ELMHES function reduces a real, nonsymmetric n by n array A to upper Hessenberg form. ELMHES is based on the routine `elmhes` described in section 11.5 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the `LA_ELMHES` function.

Syntax

Result = ELMHES(A [, /COLUMN] [, /DOUBLE] [, /NO_BALANCE])

Return Value

The result is an upper Hessenberg array with eigenvalues that are identical to those of the original array A . The Hessenberg array is stored in the upper triangle and the first subdiagonal. Elements below the subdiagonal should be ignored but are not automatically set to zero.

Arguments

A

An n by n real, nonsymmetric array.

Keywords

COLUMN

Set this keyword if the input array A is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

NO_BALANCE

Set this keyword to disable balancing. By default, a balancing algorithm is applied to A . Balancing a nonsymmetric array is recommended to reduce the sensitivity of eigenvalues to rounding errors.

Examples

See the description of the [HQR](#) for an example using this function.

Version History

Introduced: 4.0

See Also

[EIGENVEC](#), [HQR](#), [LA_ELMHES](#), [TRIQL](#), [TRIRED](#)

EMPTY

The EMPTY procedure causes all buffered output for the current graphics device to be written. IDL uses buffered output on many display devices for reasons of efficiency. This buffering leads to rare occasions where a program needs to be certain that data are not waiting in a buffer, but have actually been output. EMPTY is a low-level graphics routine. IDL graphics routines generally handle flushing of buffered data transparently to the user, so the need for EMPTY is very rare.

Syntax

EMPTY

Arguments

None.

Keywords

None.

Version History

Introduced: Original

See Also

[FLUSH](#)

ENABLE_SYSRTN

The `ENABLE_SYSRTN` procedure enables/disables IDL system routines. This procedure is intended for use by runtime and callable IDL applications, and is not generally useful for interactive use.

Special Cases

The following is a list of cases in which `ENABLE_SYSRTN` is unable to enable or disable a requested routine. All such attempts are simply ignored without issuing an error, allowing the application to run without error in different IDL environments:

- Attempts to enable/disable non-existent system routines.
- Attempts to enable a system routine disabled due to the mode in which IDL is licensed, as opposed to being disabled via `ENABLE_SYSRTN`, are quietly ignored (e.g. demo mode).
- The routines `CALL_FUNCTION`, `CALL_METHOD`, `CALL_PROCEDURE`, and `EXECUTE` cannot be disabled via `ENABLE_SYSRTN`. However, anything that can be called from them *can* be disabled, so this is not a significant drawback.

Syntax

```
ENABLE_SYSRTN [, Routines ] [, /DISABLE ] [, /EXCLUSIVE ] [, /FUNCTIONS ]
```

Arguments

Routines

A string scalar or array giving the names of routines to be enabled or disabled. By default, these are procedures, but this can be changed by setting the `FUNCTIONS` keyword.

Keywords

DISABLE

By default, the Routines are enabled. Setting this keyword causes them to be disabled instead.

EXCLUSIVE

By default, `ENABLE_SYSRTN` does not alter routines not listed in `Routines`. If `EXCLUSIVE` is set, the specified routines are taken to be the only routines that should be enabled or disabled, and all other routines have the opposite action applied.

Therefore, setting `EXCLUSIVE` and not `DISABLE` means that the routines in the `Routines` argument are enabled and all other system routines of the same type (function or procedure) are disabled. Setting `EXCLUSIVE` and `DISABLE` means that all listed routines are disabled and all others are enabled.

FUNCTIONS

Normally, the `Routines` argument specifies the names of procedures. Set the `FUNCTIONS` keyword to manipulate functions instead.

Examples

To disable the `PRINT` procedure:

```
ENABLE_SYSRTN, /DISABLE, 'PRINT'
```

To enable the `PRINT` procedure and disable all other procedures:

```
ENABLE_SYSRTN, /EXCLUSIVE, 'PRINT'
```

To ensure all possible functions are enabled:

```
ENABLE_SYSRTN, /DISABLE, /EXCLUSIVE, /FUNCTIONS
```

In the last example, all named functions should be disabled and all other functions should be enabled. Since no *Routines* argument is provided, this means that all routines become enabled.

Version History

Introduced: 5.2.1

EOF

The EOF function tests the specified file unit for the end-of-file condition.

Note

The EOF function cannot be used with files opened with the RAWIO keyword to the OPEN routines. Many of the devices commonly used with RAWIO signal their end-of-file by returning a zero transfer count to the I/O operation that encounters the end-of-file.

Syntax

Result = EOF(*Unit*)

Return Value

If the file pointer is positioned at the end of the file, EOF returns true (1), otherwise false (0) is returned.

Arguments

Unit

The file unit to test for end-of-file.

Keywords

None.

Examples

If file unit number 1 is open, the end-of-file condition can be checked by examining the value of the expression EOF(1). For example, the following IDL code reads and prints a text file:

```
; Open the file test.lis:
OPENR, 1, 'test.lis'
; Define a string variable:
A = ''
; Loop until EOF is found:
WHILE ~ EOF(1) DO BEGIN
```

```
        ; Read a line of text:  
        READF, 1, A  
        ; Print the line:  
        PRINT,  
    ENDWHILE  
    ; Close the file:  
    CLOSE, 1
```

Version History

Introduced: Original

See Also

[POINT_LUN](#)

EOS_* Routines

For information, see [Chapter 5, “HDF-EOS”](#) in the *IDL Scientific Data Formats* manual.

ERASE

The ERASE procedure erases the screen of the currently selected graphics device (or starts a new page if the device is a printer). The device is reset to alphanumeric mode if it has such a mode (e.g., Tektronix terminals).

Syntax

```
ERASE [, Background_Color] [, CHANNEL=value] [, COLOR=value]
```

Arguments

Background_Color

The color index for the screen to be erased to. If this argument is omitted, ERASE resets the screen to the default background color (normally 0) stored in the system variable !P.BACKGROUND. Providing a value for *Background_Color* overrides the default.

Warning

Not all devices support this feature.

Keywords

CHANNEL

The channel or channel mask for the erase operation. This parameter has meaning only when used with devices that support TrueColor or multiple-display channels. The default value is !P.CHANNEL.

COLOR

Specifies the background color. Using this keyword is analogous to using the *Background_Color* argument.

Examples

```
; Display a simple image in the current window:
TV, DIST(255)

; Erase the image from the window:
ERASE
```

Version History

Introduced: Original

See Also

[SET_PLOT](#), [WINDOW](#), [WSET](#)

ERF

The ERF function returns the value of the error function:

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

For real input, the error function is computed using rational functions, as described in “Rational Chebyshev approximations for the error function,” W. J. Cody, *Math. Comp.*, 1969, pp. 631-638. For complex input, the error function is computed as $\text{Sign} \times \text{IGAMMA}(0.5, Z^2)$, where *Sign* is taken from the real part of Z .

Syntax

Result = ERF(*Z*)

Return Value

The result is double-precision if the argument is double-precision, otherwise the result is floating-point. The result always has the same structure as Z . The ERF function also accepts complex arguments.

Arguments

Z

The expression for which the error function is to be evaluated. Z may be complex.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

To find the error function of 0.4 and print the result, enter:

```
PRINT, ERF(0.4D)
```

IDL prints:

```
0.42839236
```

Version History

Introduced: Pre 4.0

Z argument accepts complex input: 5.6

See Also

[ERFC](#), [ERFCX](#), [EXPINT](#), [GAMMA](#), [IGAMMA](#)

ERFC

The ERFC function returns the value of the complementary error function:

$$\operatorname{erfc}(z) = 1 - \operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_z^{\infty} e^{-t^2} dt$$

For real input, ERFC is computed using rational functions, as described in “Rational Chebyshev approximations for the error function,” W. J. Cody, *Math. Comp.*, 1969, pp. 631-638. For complex input, ERFC is computed as $1 - \operatorname{ERF}(Z)$.

Syntax

Result = ERFC(*Z*)

Return Value

The result is double-precision if the argument is double-precision, otherwise the result is floating-point. The result always has the same structure as *Z*. The ERFC function also accepts complex arguments.

Arguments

Z

The expression for which the complementary error function is to be evaluated. *Z* may be complex.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

To find the complementary error function of 0.4 and print the result, enter:

```
PRINT, ERFC(0.4D)
```

IDL prints:

```
0.57160764
```

Version History

Introduced: Pre 4.0

Z argument accepts complex input: 5.6

See Also

[ERFC](#), [ERFCX](#)

ERFCX

The ERFCX function returns the value of the scaled complementary error function:

$$\text{erfcx}(z) = e^{z^2} \text{erfc}(z)$$

For real input, ERFCX is computed using rational functions, as described in “Rational Chebyshev approximations for the error function,” W. J. Cody, *Math. Comp.*, 1969, pp. 631-638. For complex input, ERFCX is computed as $\text{EXP}(Z^2)\text{ERFC}(Z)$.

Syntax

Result = ERFCX(*Z*)

Return Value

The result is double-precision if the argument is double-precision, otherwise the result is floating-point. The result always has the same structure as *Z*. The ERFCX function also accepts complex arguments.

Arguments

Z

The expression for which the scaled complementary error function is to be evaluated. *Z* may be complex.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

To find the scaled complementary error function of 0.4 and print the result, enter:

```
PRINT, ERFCX(0.4D)
```

IDL prints:

```
0.67078779
```

Version History

Introduced: 5.5

Z argument accepts complex input: 5.6

See Also

[ERF](#), [ERFC](#)

ERODE

The ERODE function implements the erosion operator on binary and grayscale images and vectors. This operator is commonly known as “shrink” or “reduce”.

Using ERODE

See the description of the [DILATE](#) function for background on morphological operators. Erosion is the dual of dilation. It does to the background what dilation does to the foreground. Briefly, given an *Image* and a structuring element, *Structure*, the ERODE function can be used to remove islands smaller than the structuring element.

Over each pixel of the image, the origin of the structuring element is overlaid. If each nonzero element of the structuring element is contained in the image, the output pixel is set to one. Letting $A \otimes B$ represent the erosion of an image A by structuring element B , erosion can be defined as:

$$C = A \otimes B = \bigcap_{b \in B} (A)_{-b}$$

where $(A)_{-b}$ represents the translation of A by b . The structuring element B can be visualized as a probe that slides across image A , testing the spatial nature of A at each point. If B translated by i,j can be contained in A (by placing the origin of B at i,j), then i,j belongs to the erosion of A by B . For example:

$$\begin{array}{r}
 0100 \quad 0000 \\
 0100 \quad 0000 \\
 1110 \otimes 11 = 1100 \\
 1000 \quad 0000 \\
 0000 \quad 0000
 \end{array}$$

In this example, the origin of the structuring element is at (0, 0).

Used with grayscale images, which are always converted to byte type, the ERODE function is accomplished by taking the minimum of a set of differences. It can be used to conveniently implement the neighborhood minimum operator with the shape of the neighborhood given by the structuring element.

Syntax

```
Result = ERODE( Image, Structure [, X0 [, Y0 [, Z0]]] [, /GRAY
  [, /PRESERVE_TYPE | , /UINT | , /ULONG]] [, VALUES=array] )
```

Return Value

The ERODE function returns the erosion of *Image* by the structuring element *Structure*.

Arguments

Image

A one-, two-, or three-dimensional array upon which the erosion is to be performed. If this parameter is not of byte type, a temporary byte copy is obtained. If neither of the keywords GRAY or VALUES is present, the image is treated as a binary image with all nonzero pixels considered as 1.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values—either zero or nonzero. The structuring element must have the same number of dimensions as *Image*.

X₀, Y₀, Z₀

Optional parameters specifying the one-, two-, or three-dimensional coordinate of the structuring element's origin. If omitted, the origin is set to the center, ($[N_x/2]$, $[N_y/2]$, $[N_z/2]$), where N_x , N_y , and N_z are the dimensions of the structuring element array. The origin need not be within the structuring element.

Keywords

GRAY

Set this keyword to perform grayscale, rather than binary, erosion. Nonzero elements of the *Structure* parameter determine the shape of the structuring element (neighborhood). If *VALUES* is not present, all elements of the structuring element are 0, yielding the neighborhood minimum operator.

PRESERVE_TYPE

Set this keyword to return the same type as the input array. This keyword only applies if the *GRAY* keyword is set.

UINT

Set this keyword to return an unsigned integer array. This keyword only applies if the *GRAY* keyword is set.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword only applies if the *GRAY* keyword is set.

VALUES

An array of the same dimensions as *Structure* providing the values of the structuring element. The presence of this keyword implies grayscale erosion. Each pixel of the result is the minimum of Image less the corresponding elements of *VALUE*. If the resulting difference is less than zero, the return value will be zero.

Examples

Example 1

This example thresholds a grayscale image at the value of 100, producing a binary image. The result is then “opened” with a 3 pixel by 3 pixel square shape operator, using the *ERODE* and *DILATE* operators. The effect is to remove holes, islands, and peninsula smaller than the shape operator:

```
; Threshold and make binary image:
B = A GE 100

; Create the shape operator:
S = REPLICATE(1, 3, 3)
```

```
; "Opening" operator:  
C = DILATE(ERODE(B, S), S)  
  
; Show the result:  
TVSCL, C
```

Example 2

For grayscale images, `ERODE` takes the neighborhood minimum, where the shape of the neighborhood is given by the structuring element. Elements for which the structuring element extends off the array are indeterminate. For example, assume you have the following image and structuring element:

```
image = BYTE([2,1,3,3,3,3,1,2])
s = [1,1]
```

If the origin of the structuring element is not specified in the call to ERODE, the origin defaults to one half the width of the structuring element, which is 1 in this case. Therefore, for the first element in the image array, the structuring element is aligned with the image as depicted below:

```
  [2,1,3,3,3,3,1,2]
  ↑
 [1,1]
```

This will cause an indeterminate value for the first element in the ERODE result. If edge values are important, you must pad the image with as many elements as there are elements in the structuring element that extend off the array, in all dimensions. The value of the padding elements must be the maximum value in the image, since ERODE calculates a neighborhood minimum. In this case, you would need to pad the image with a single leading 3. If the structuring element were `s=[1,1,1,1]`, and you specified an origin of 2, the structuring element would align with the image as follows:

```
  [2,1,3,3,3,3,1,2]
  ↑           ↑
 [1,1,1,1]   [1,1,1,1]
```

Therefore, you would need to pad the image with at least two leading 3s and at least one trailing 3. You would then perform the erosion operation on the padded image, and remove the padding from the result.

The following code illustrates this method:

```
image = BYTE([2,1,3,3,3,3,1,2])
s = [1,1] ; Structuring element
PRINT, 'Image: '
PRINT, image

PRINT, 'Erosion using no padding: '
PRINT, ERODE(image, s, /GRAY)

result = ERODE([MAX(image), image], s, /GRAY)
PRINT, 'Erosion using padding: '
PRINT, result[1:N_ELEMENTS(image)]
```

IDL prints:

```
Image:
  2  1  3  3  3  3  1  2
Erosion using no padding:
  0  1  1  3  3  3  1  1
```

Erosion using padding:

2 1 1 3 3 3 1 1

Version History

Introduced: Pre 4.0

See Also

[DILATE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#), [MORPH_GRADIENT](#),
[MORPH_HITORMISS](#), [MORPH_OPEN](#), [MORPH_THIN](#), [MORPH_TOPHAT](#)

ERRPLOT

The ERRPLOT procedure plots error bars over a previously drawn plot.

This routine is written in the IDL language. Its source code can be found in the file `errplot.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
ERRPLOT, [ X, ] Low, High [, WIDTH=value]
```

Arguments

X

A vector containing the abscissa values at which the error bars are to be plotted. *X* only needs to be provided if the abscissa values are not the same as the index numbers of the plotted points.

Low

A vector of lower estimates, equal to data - error.

High

A vector of upper estimates, equal to data + error.

Keywords

WIDTH

The width of the error bars. The default is 1% of plot width.

Examples

To plot symmetrical error bars where *Y* is a vector of data values and *ERR* is a symmetrical error estimate, enter:

```
; Plot data:
PLOT, Y

; Overplot error bars:
ERRPLOT, Y-ERR, Y+ERR
```

If error estimates are non-symmetrical, provide actual error estimates in the *upper* and *lower* arguments.

```
; Plot data:  
PLOT, Y
```

```
; Provide custom lower and upper bounds:  
ERRPLOT, lower, upper
```

To plot Y versus a vector of abscissas:

```
; Plot data (X versus Y):  
PLOT, X, Y
```

```
; Overplot error estimates:  
ERRPLOT, X, Y-ERR, Y+ERR
```

Version History

Introduced: Original

See Also

[OPLOTERR](#), [PLOT](#), [PLOTERR](#)

EXECUTE

The EXECUTE function compiles and executes one or more IDL statements contained in a string at run-time.

Warning

Use of the EXECUTE function is not permitted when IDL is in Virtual Machine mode.

Like the CALL_PROCEDURE and CALL_FUNCTION routines, calls to EXECUTE can be nested. However, compiling the string at run-time is inefficient. CALL_FUNCTION and CALL_PROCEDURE provide much of the functionality of EXECUTE without imposing this limitation, and should be used instead of EXECUTE whenever possible.

Syntax

Result = EXECUTE(*String* [, *QuietCompile*] [, *QuietExecution*])

Return Value

EXECUTE returns *true* (1) if the string was successfully compiled and executed. If an error occurs during either phase, the result is *false* (0).

Arguments

String

A string containing the command(s) to be compiled and executed.

QuietCompile

If this argument is set to a non-zero value, EXECUTE will not print the compiler generated error messages (such as syntax errors). If *QuietCompile* is omitted or set to 0, EXECUTE will output such errors.

QuietExecution

If this argument is set to a non-zero value, EXECUTE will not print error messages generated during execution. If *QuietExecution* is omitted or set to 0, EXECUTE will output such errors.

Keywords

None.

Examples

Create a string that holds a valid IDL command and execute the command by entering:

```
com = 'PLOT, [0,1]'  
void = EXECUTE(com)
```

Version History

Introduced: Original

QuietCompile argument: IDL 5.2

QuietExecution argument: IDL 6.1

Execute the contents of the string by entering:

```
R = EXECUTE (com)
```

A plot should appear. You can confirm that the string was successfully compiled and executed by checking that the value of R is 1.

Version History

Introduced: Original

See Also

[CALL_FUNCTION](#), [CALL_METHOD](#), [CALL_PROCEDURE](#)

EXIT

The EXIT procedure quits IDL and exits back to the operating system. All buffers are flushed and open files are closed. The values of all variables that were not saved are lost.

Syntax

```
EXIT [, /NO_CONFIRM] [, STATUS=code]
```

Arguments

None.

Keywords

NO_CONFIRM

Set this keyword to suppress any confirmation dialog that would otherwise be displayed in a GUI version of IDL such as the IDL Development Environment.

STATUS

Set this keyword equal to an exit status code that will be returned when IDL exits. For example, on a UNIX system using the Bourne shell:

Start IDL:

```
$ idl
```

Exit IDL specifying exit status 45:

```
IDL> exit, status=45
```

Display last exit status code:

```
$ echo $?
```

The following displays:

```
45
```

Version History

Introduced: Original

See Also

[CLOSE](#), [FLUSH](#), [STOP](#), [WAIT](#)

EXP

The EXP function returns the natural exponential function of *Expression*.

Syntax

Result = EXP(*Expression*)

Return Value

Returns the natural exponential function of the given *Expression*.

Arguments

Expression

The expression to be evaluated. If *Expression* is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. The definition of the exponential function for complex arguments is:

$$\text{EXP}(x) = \text{COMPLEX}(e^R \cos I, e^R \sin I)$$

where:

R = real part of x , and I = imaginary part of x . If *Expression* is an array, the result has the same structure, with each element containing the result for the corresponding element of *Expression*.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Plot a Gaussian with a 1/e width of 10 and a center of 50 by entering:

```
PLOT, EXP(-(FINDGEN(100)/10. - 5.0)^2)
```

Version History

Introduced: Original

See Also

[ALOG](#)

EXPAND

The EXPAND procedure shrinks or expands a two-dimensional array, using bilinear interpolation. It is similar to the CONGRID and REBIN routines.

This routine is written in the IDL language. Its source code can be found in the file `expand.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
EXPAND, A, Nx, Ny, Result [, FILLVAL=value] [, MAXVAL=value]
```

Arguments

A

A two-dimensional array to be magnified.

Nx

Desired size of the X dimension, in pixels.

Ny

Desired size of the Y dimension, in pixels.

Result

A named variable that will contain the magnified array.

Keywords

FILLVAL

Set this keyword equal to the value to use when elements larger than MAXVAL are encountered. The default is -1.

MAXVAL

Set this keyword equal to the largest desired value. Elements greater than this value are set equal to the value of the FILLVAL keyword.

Version History

Introduced: Pre 4.0

See Also

[CONGRID](#), [REBIN](#)

EXPAND_PATH

The EXPAND_PATH function is used to expand a simple path-definition string into a full path listing. The returned path can be useful for setting the !PATH, and !HELP_PATH system variables, for example.

- !PATH is a list of locations where IDL searches for currently undefined procedures and functions.
- !HELP_PATH is a list of locations where IDL searches for online help files when the online help facility is used.

Note

The mechanism used by EXPAND_PATH to expand the path-definition string is the same as that is used to expand the contents of the environment variables IDL_PATH, IDL_DLM_PATH, and IDL_HELP_PATH at startup. See [“The Path Definition String”](#) below for more information.

The Path Definition String

EXPAND_PATH accepts a single argument, a scalar string that contains a simple path-definition string. EXPAND_PATH expands the path-definition string into a list of directories that can be assigned to the !PATH, !DLM_PATH, or !HELP_PATH system variables.

Note

The syntax of the path definition string describe here can also be used when setting the IDL_PATH, IDL_DLM_PATH, and IDL_HELP_PATH environment variables. When IDL reads the environment variable at startup, it will treat the contents of the environment variable in the same way EXPAND_PATH treats the path definition string.

IDL supports the following special notations within the path definition string:

- **Using “+”** — When IDL encounters a “+” in front of a directory name, it searches the directory and all of its subdirectories for files of the appropriate type for the given path:
 - IDL program files (.pro or .sav) if neither the DLM nor the HELP keywords to EXPAND_PATH are present, or in the IDL_PATH environment variable.

- Dynamically Loadable Module files (.dlm) if the DLM keyword to EXPAND_PATH is present, or in the IDL_DLM_PATH environment variable.
- Files that can be used by IDL's online help system, if the HELP keyword to EXPAND_PATH is present, or in the IDL_HELP_PATH environment variable. On UNIX platforms, help files are Adobe Portable Document Format (.pdf) files, HTML format (.html or .htm) files, or have the file extension .help. On Windows systems, help files can be either HTML Help (.chm), WinHelp (.hlp), PDF (.pdf), or HTML (.html or .htm) files.

Any directory containing one or more of the appropriate type of file is added to the path.

If the “+” is *not* present, the specified directory is added to the path regardless of its contents.

Order of Expanded Directories

When expanding a path segment starting with “+”, IDL ensures that all directories containing the appropriate type files are placed in the path string. The order in which the directories in such an expanded path segment appear is completely unspecified, and does not necessarily correspond to any specific order (such as top-down alphabetized). This allows IDL to construct the path in the fastest possible way and speeds the process of loading paths from environment variables at startup. This is only a problem if two subdirectories in such a hierarchy contain a file with the same name.

If the order in which “+” expands directories is a problem for your application, you should add the directories to the path explicitly and not use “+”. Only the order of the files within a given “+” entry are determined by IDL. It never reorders !PATH (or !DLM_PATH or !HELP_PATH) in any other way. You can therefore obtain any search order you desire by writing the path explicitly.

- **Using “<IDL_DEFAULT>”** — IDL replaces any occurrence of the token <IDL_DEFAULT> in a path definition string with the default value IDL would have used if no environment variable or preference were set. The actual value of this placeholder depends on where IDL has been installed. Hence, to view IDL's default path:

```
PRINT, EXPAND_PATH(' <IDL_DEFAULT>')
```

To append your own directory after IDL's default DLM path using the IDL_DLM_PATH environment variable (under UNIX):

```
% setenv IDL_DLM_PATH "<IDL_DEFAULT>:/your/path/here"
```

(Setting the Windows environment variable `IDL_DLM_PATH` to a similar string would produce the same result on a Windows system.) This substitution allows you to set up your paths without having to hard-code IDL's defaults into your startup scripts or environment variables.

Note that the actual path that the token `<IDL_DEFAULT>` expands to depends on the context in which it is used. The default path for `.pro` and `.sav` files is different from the default path for `.dlm` files or help files. To see this, enter the following statements into IDL:

```
PRINT, EXPAND_PATH( '<IDL_DEFAULT>' )
PRINT, EXPAND_PATH( '<IDL_DEFAULT>', /DLM )
PRINT, EXPAND_PATH( '<IDL_DEFAULT>', /HELP )
```

Three variations of the `<IDL_DEFAULT>` token eliminate this dependence on context:

- `<IDL_DEFAULT_PATH>` defines the default path to `.pro` and `.sav` files.
- `<IDL_DEFAULT_DLM>` defines the default path to `.dlm` files.
- `<IDL_DEFAULT_HELP>` defines the default path to help (`.pdf`, `.html`, and `.chm`) files.
- **Using “`<IDL_BIN_DIRNAME>`”** — IDL replaces any occurrence of the token `<IDL_BIN_DIRNAME>` in a path definition string with the name of the subdirectory within the installed IDL distribution where binaries for the current system are kept. This feature is useful for distributing packages of DLMs (Dynamically Loadable Modules) with support for multiple operating system and hardware combinations.
- **Using “`<IDL_VERSION_DIRNAME>`”** — IDL replaces any occurrence of the token `<IDL_VERSION_DIRNAME>` in a path definition string with a unique name for the IDL version that is currently running. This feature can be combined with `<IDL_BIN_DIRNAME>` to easily distribute packages of DLMs with support for multiple IDL versions, operating systems, and hardware platforms.

Note

See “[!DLM_PATH](#)” on page 4365 for examples using the `<IDL_BIN_DIRNAME>` and `<IDL_VERSION_DIRNAME>` tokens.

Syntax

```
Result = EXPAND_PATH( String [, /ALL_DIRS] [, /ARRAY] [, COUNT=variable]
[, /DLM] [, /HELP] )
```

Return Value

Returns a list of directories that can be assigned to the `!PATH`, `!DLM_PATH`, or `!HELP_PATH` system variables given a string path to be expanded.

Arguments

String

A scalar string containing the path-definition string to be expanded. See [“The Path Definition String”](#) for details.

Keywords

ALL_DIRS

Set this keyword to return all directories without concern for their contents, otherwise, `EXPAND_PATH` only returns those directories that contain `.pro` or `.sav` files.

ARRAY

Set this keyword to return the result as a string array with each element containing one path segment. In this case, there is no need for a separator character and none is supplied. Normally, the result is a string array with the path segments separated with the correct special delimiter character for the current operating system.

COUNT

Set this keyword to a named variable which returns the number of path segments contained in the result.

DLM

Set this keyword to return those directories that contain IDL Dynamically Loadable Module (`.dlm`) description files.

HELP

Set this keyword to return directories that contain help files. On UNIX platforms, help files are in Adobe Portable Document Format (`.pdf`), HTML format (`.html` or `.htm`), or have the file extension `.help`. On Windows systems, help files can be either HTML Help (`.chm`), WinHelp (`.hlp`), PDF (`.pdf`), or HTML (`.html` or `.htm`) files.

Examples

Example 1

Assume you have the following directory structure:

```
/home
  myfile.txt
  /programs
    /pro
      myfile.pro
```

Search the `/home` directory and all its subdirectories, and return the directories containing `.pro` and `.sav` files:

```
PRINT, EXPAND_PATH( '+/home' )
```

IDL prints:

```
/home/programs/pro
```

Example 2

Search the same directory, but this time return all directories, not just those containing `.pro` and `.sav` files:

```
PRINT, EXPAND_PATH( '+home', /ALL_DIRS )
```

IDL prints:

```
/home/programs/pro:/home/programs
```

Example 3

Print the default value of the `!DLM_PATH` system variable:

```
PRINT, EXPAND_PATH( '<IDL_DEFAULT>', /DLM )
```

Version History

Introduced: Pre 4.0

Modified to use the `<IDL_*_PATH>` syntax: 5.6

See Also

“[Running IDL Programs](#)” in Chapter 9 of the *Using IDL* manual and “[IDL Environment System Variables](#)” on page 4363.

EXPINT

The EXPINT function returns the value of the exponential integral $E_n(x)$.

EXPINT is based on the routine `expint` described in section 6.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = EXPINT( N, X [, /DOUBLE] [, EPS=value] [, ITER=variable]
               [, ITMAX=value] )
```

Return Value

Returns the exponential integral $E_n(x)$.

Arguments

N

An integer specifying the order of $E_n(x)$. N can be either a scalar or an array.

X

The value at which $E_n(x)$ is evaluated. X can be either a scalar or an array.

Note: If an array is specified for both N and X , then EXPINT evaluates $E_n(x)$ for each N_i and X_i . If either N or X is a scalar and the other an array, the scalar is paired with each array element in turn.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic. Set this keyword to zero to return a single-precision result.

Note

All internal computations are done using double-precision arithmetic.

EPS

Use this keyword to specify a number close to the desired relative error. The default value is 3.0×10^{-12} .

ITER

Set this keyword equal to a named variable that will contain the actual number of iterations performed.

ITMAX

An input integer specifying the maximum allowed number of iterations. The default value is 100000.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To compute the value of the exponential integral at the following X values:

```
; Define the parametric X values:
X = [1.00, 1.05, 1.27, 1.34, 1.38, 1.50]

; Compute the exponential integral of order 1:
result = EXPINT(1, X)

; Print the result:
PRINT, result
```

IDL prints:

```
0.219384 0.201873 0.141911 0.127354 0.119803 0.100020
```

This is the exact solution vector to six-decimal accuracy.

Version History

Introduced: 4.0

ITER keyword added: 5.6

See Also

[ERF](#)

EXTRAC

The EXTRAC function returns a defined portion of an array or vector. The main advantage to EXTRAC is that, when parts of the specified subsection lie outside the bounds of the array, zeros are returned for these outlying elements. It is usually more efficient to use the array subscript ranges (the “:” operator; see “[Subscript Ranges](#)” in Chapter 7 of the *Building IDL Applications* manual) to perform such operations.

EXTRAC was originally a built-in system procedure in the PDP-11 version of IDL, and was retained in that form in the original VAX/VMS IDL for compatibility. Most applications of the EXTRAC function are more concisely written using subscript ranges (e.g., X(10:15)). EXTRAC has been rewritten as a library function that provides the same interface as the previous versions.

Note

If you know that the subarray will never lie beyond the edges of the array, it is more efficient to use array subscript ranges (the “:” operator) to extract the data instead of EXTRAC.

This routine is written in the IDL language. Its source code can be found in the file `extrac.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$Result = EXTRAC(Array, C_1, C_2, \dots, C_n, S_1, S_2, \dots, S_n)$$

Return Value

Returns any rectangular sub-matrix or portion of the parameter array.

Arguments

Array

The array from which the subarray will be copied.

C_i

The starting subscript in *Array* for the subarray. There should be one C_i for each dimension of *Array*. These arguments must be integers.

S_i

The size of each dimension. The result will have dimensions of (S_1, S_2, \dots, S_n) . There should be one S_i for each dimension of *Array*. These arguments must be non-negative.

Keywords

None.

Examples

Extracting elements from a vector:

```
; Create a 1000 element floating-point vector with each element set
; to the value of its subscript:
A = FINDGEN(1000)
; Extract 300 points starting at A[200] and extending to A[499]:
B = EXTRAC(A, 200, 300)
```

In the next example, the first 50 elements extracted — B[0] to B[49] — lie outside the bounds of the vector and are set to 0. The value of B[50] is the same as the value of A[0], and the value of B[51] is equal to A[1] which is 1. Enter:

```
; Create a 1000 element vector:
A = FINDGEN(1000)
; Extract 50 elements, 49 of which lie outside the bounds of A:
B = EXTRAC(A, -50, 100)
```

The following commands illustrate the use of EXTRAC with multi-dimensional arrays:

```
; Make a 64 by 64 array:
A = INTARR(64,64)
; Extract a 32 by 32 portion starting at A(20,30):
B = EXTRAC(A, 20, 30, 32, 32)
```

As suggested in the discussion above, a better way to perform the same operation as the previous line is:

```
; Use the array subscript operator instead of EXTRAC:
B = A(20:51, 30:61)
```

Extract the 20th column and 32nd row of A:

```
; Extract 20th column of A:
B = EXTRAC(A, 19, 0, 1, 64)
; Extract 32nd row of A:
B = EXTRAC(A, 0, 31, 64, 1)
```

Take a 32 BY 32 matrix from A starting at A(40,50):

```
; Note that those points beyond the boundaries of A are set to 0:  
B = EXTRAC(A, 40, 50, 32, 32)
```

Version History

Introduced: Pre 4.0

See Also

[“Subscript Ranges”](#) in Chapter 7 of the *Building IDL Applications* manual.

EXTRACT_SLICE

The `EXTRACT_SLICE` function extracts a specified planar slice of volumetric data. This function allows for a rotation or vector form of the slice equation. In the vector form, the slice plane is governed by the plane equation ($ax+by+cz+d=0$) and a single vector which defines the x direction. This form is more common throughout the IDL polygon interface. In the rotation form, the slicing plane can be oriented at any angle and can pass through any desired location in the volume.

This function allows for a vertex grid to be generated without sampling the data. In this form, the vertices could be used to sample additional datasets or used to form polygonal meshes. It would also be useful to return the planar mesh connectivity in this case.

Support for anisotropic data volumes is included via an `ANISOTROPY` keyword. This is an important feature in the proper interpolation of common medical imaging data.

This routine is written in the IDL language. Its source code can be found in the file `extract_slice.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = EXTRACT_SLICE( Vol, Xsize, Ysize, Xcenter, Ycenter, Zcenter, Xrot, Yrot,
  Zrot [, ANISOTROPY=[xspacing, yspacing, zspacing]] [, OUT_VAL=value]
  [, /RADIANS] [, /SAMPLE] [, VERTICES=variable] )
```

or

```
Result = EXTRACT_SLICE( Vol, Xsize, Ysize, Xcenter, Ycenter, Zcenter,
  PlaneNormal, Xvec [, ANISOTROPY=[xspacing, yspacing, zspacing]]
  [, OUT_VAL=value] [, /RADIANS] [, /SAMPLE] [, VERTICES=variable] )
```

Return Value

Returns a two-dimensional planar slice extracted from 3-D volumetric data or returns a vertex grid in the form of a [3,n] array of vertices.

Arguments

PlaneNormal

Set this input argument to a 3 element array. The values are interpreted as the normal of the slice plane.

Xvec

Set this input argument to a 3 element array. The three values are interpreted as the 0 dimension directional vector. This should be a unit vector.

Vol

The volume of data to slice. This argument is a three-dimensional array of any type except string or structure. The planar slice returned by `EXTRACT_SLICE` has the same data type as *Vol*.

Xsize

The desired X size (dimension 0) of the returned slice. To preserve the correct aspect ratio of the data, Xsize should equal Ysize. For optimal results, set Xsize and Ysize to be greater than or equal to the largest of the three dimensions of *Vol*.

Ysize

The desired Ysize (dimension 1) of the returned slice. To preserve the correct aspect ratio of the data, Ysize should equal Xsize. For optimal results, set Xsize and Ysize to be greater than or equal to the largest of the three dimensions of *Vol*.

Xcenter

The X coordinate (index) of the point within the volume that the slicing plane passes through. The center of the slicing plane passes through *Vol* at the coordinate (*Xcenter*, *Ycenter*, *Zcenter*).

Ycenter

The Y coordinate (index) of the point within the volume that the slicing plane passes through. The center of the slicing plane passes through *Vol* at the coordinate (*Xcenter*, *Ycenter*, *Zcenter*).

Zcenter

The Z coordinate (index) of the point within the volume that the slicing plane passes through. The center of the slicing plane passes through *Vol* at the coordinate (*Xcenter*, *Ycenter*, *Zcenter*).

Xrot

The X-axis rotation of the slicing plane, in degrees. Before transformation, the slicing plane is parallel to the X-Y plane. The slicing plane transformations are performed in the following order:

- Rotate *Z_rot* degrees about the Z axis.
- Rotate *Y_rot* degrees about the Y axis.
- Rotate *X_rot* degrees about the X axis.
- Translate the center of the plane to *Xcenter*, *Ycenter*, *Zcenter*.

Yrot

The Y-axis rotation of the slicing plane, in degrees.

Zrot

The orientation Z-axis rotation of the slicing plane, in degrees.

Keywords

ANISOTROPY

Set this keyword to a three-element array. This array specifies the spacing between the planes of the input volume in grid units of the (isotropic) output image.

OUT_VAL

Set this keyword to a value that will be assigned to elements of the returned slice that lie outside of the original volume.

RADIANS

Set this keyword to indicate that *Xrot*, *Yrot*, and *Zrot* are in radians. The default is degrees.

SAMPLE

Set this keyword to perform nearest neighbor sampling when computing the returned slice. The default is to use bilinear interpolation. A small reduction in execution time results when SAMPLE is set and the OUT_VAL keyword is *not* used.

VERTICES

Set this output keyword to a named variable in which to return a [3,Xsize,Ysize] floating point array. This is an array of the x, y, z sample locations for each pixel in the normal output.

Obsolete Keywords

The following keywords are obsolete:

- CUBIC

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

Display an oblique slice through volumetric data:

```

; Create some data:
vol = RANDOMU(s, 40, 40, 40)

; Smooth the data:
FOR i=0, 10 DO vol = SMOOTH(vol, 3)

; Scale the smoothed part into the range of bytes:
vol = BYTACL(vol(3:37, 3:37, 3:37))

; Extract a slice:
slice = EXTRACT_SLICE(vol, 40, 40, 17, 17, 17, 30.0, 30.0, 0.0, $
    OUT_VAL=0B)

; Display the 2D slice as a magnified image:
TVACL, REBIN(slice, 400, 400)

```

Version History

Introduced: Pre 4.0

See Also

[SLICER3](#)

F_CVF

The `F_CVF` function computes the cutoff value V in an F distribution with degrees of freedom in the numerator and degrees of freedom in the denominator such that the probability that a random variable X is greater than V is equal to a user-supplied probability P .

This routine is written in the IDL language. Its source code can be found in the file `f_cvf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$Result = F_CVF(P, Dfn, Dfd)$$

Return Value

Returns the cutoff value V , given a distribution F , with specified degrees of freedom in the numerator and denominator.

Arguments

P

A non-negative single- or double-precision floating-point scalar, in the interval $[0.0, 1.0]$, that specifies the probability of occurrence or success.

Dfn

A positive integer, single- or double-precision floating-point scalar that specifies the number of degrees of freedom of the F distribution numerator.

Dfd

A positive integer, single- or double-precision floating-point scalar that specifies the number of degrees of freedom of the F distribution denominator.

Keywords

None.

Examples

Use the following command to compute the cutoff value in an F distribution with ten degrees of freedom in the numerator and six degrees of freedom in the denominator such that the probability that a random variable X is greater than the cutoff value is 0.01. The result should be 7.87413:

```
PRINT, F_CVF(0.01, 10, 6)
```

Version History

Introduced: 4.0

See Also

[CHISQR_CVF](#), [F_PDF](#), [GAUSS_CVF](#), [T_CVF](#)

F_PDF

The `F_PDF` function computes the probability P that, in an F distribution with defined degrees of freedom in the numerator and denominator, a random variable X is less than or equal to a user-specified cutoff value V .

This routine is written in the IDL language. Its source code can be found in the file `f_pdf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$\text{Result} = \text{F_PDF}(V, Dfn, Dfd)$$

Return Value

If all arguments are scalar, the function returns a scalar. If all arguments are arrays, the function matches up the corresponding elements of V , Dfn , and Dfd , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other arguments are arrays, the function uses the scalar value with each element of the arrays, and returns an array with the same dimensions as the smallest input array.

If any of the arguments are double-precision, the result is double-precision, otherwise the result is single-precision.

Arguments

V

A scalar or array that specifies the cutoff value(s).

Dfn

A positive scalar or array that specifies the number of degrees of freedom of the F distribution numerator.

Dfd

A positive scalar or array that specifies the number of degrees of freedom of the F distribution denominator.

Keywords

None.

Examples

Use the following command to compute the probability that a random variable X , from the F distribution with five degrees of freedom in the numerator and 24 degrees of freedom in the denominator, is less than or equal to 3.90. The result should be 0.990059:

```
PRINT, F_PDF(3.90, 5, 24)
```

Version History

Introduced: 4.0

See Also

[BINOMIAL](#), [CHISQR_PDF](#), [F_CVF](#), [GAUSS_PDF](#), [T_PDF](#)

FACTORIAL

The FACTORIAL function computes the factorial $N!$ For integers, the factorial is computed as $(N) \cdot (N - 1) \cdot (N - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$. For non-integers the factorial is computed using $\text{GAMMA}(N+1)$.

This routine is written in the IDL language. Its source code can be found in the file `factorial.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = FACTORIAL(*N* [, /STIRLING] [, /UL64])

Return Value

Returns the product of the non-negative scalar value or array of values.

Arguments

N

A non-negative scalar or array of values.

Note

Large values of N will cause floating-point overflow errors. The maximum size of N varies with machine architecture. On machines that support the IEEE standard for floating-point arithmetic, the maximum value of N is 170. See [MACHAR](#) for a discussion of machine-specific parameters affecting floating-point arithmetic.

Keywords

STIRLING

Set this keyword to use Stirling's asymptotic formula to approximate $N!$:

$$N! = \sqrt{2\pi N} \left[\frac{N}{e} \right]^N$$

where e is the base of the natural logarithm.

UL64

Set this keyword to return the results as unsigned 64-bit integers. This keyword is ignored if `STIRLING` is set.

Note

Unsigned 64-bit integers will overflow for values of N greater than 20.

Examples

Compute 20!:

```
PRINT, FACTORIAL(20)
```

IDL prints:

```
2.4329020e+18
```

Version History

Introduced: Pre 4.0

See Also

[BINOMIAL](#), [TOTAL](#)

FFT

The FFT function returns a result equal to the complex, discrete Fourier transform of *Array*. The result of this function is a single- or double-precision complex array.

The discrete Fourier transform, $F(u)$, of an N -element, one-dimensional function, $f(x)$, is defined as:

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) \exp[-j2\pi ux/N]$$

And the inverse transform, (*Direction* > 0), is defined as:

$$f(x) = \sum_{u=0}^{N-1} F(u) \exp[j2\pi ux/N]$$

If the keyword **OVERWRITE** is set, the transform is performed in-place, and the result overwrites the original contents of the array.

Running Time

For a one-dimensional FFT, running time is roughly proportional to the total number of points in *Array* times the sum of its prime factors. Let N be the total number of elements in *Array*, and decompose N into its prime factors:

$$N = 2^{K_2} \cdot 3^{K_3} \cdot 5^{K_5} \dots$$

Running time is proportional to:

$$T_0 + N(T_1 + 2K_2T_2 + T_3(3K_3 + 5K_5 + \dots))$$

where $T_3 \sim 4T_2$. For example, the running time of a 263 point FFT is approximately 10 times longer than that of a 264 point FFT, even though there are fewer points. The sum of the prime factors of 263 is 264 (1 + 263), while the sum of the prime factors of 264 is 20 (2 + 2 + 2 + 3 + 11).

Syntax

```
Result = FFT( Array [, Direction] [, DIMENSION=vector] [, /DOUBLE]
             [, /INVERSE] [, /OVERWRITE] )
```

Return Value

FFT returns a complex array that has the same dimensions as the input array. The output array is ordered in the same manner as almost all discrete Fourier transforms. Element 0 contains the zero frequency component, F_0 . The array element F_1 contains the smallest, nonzero positive frequency, which is equal to $1/(N_i T_i)$, where N_i is the number of elements and T_i is the sampling interval of the i^{th} dimension. F_2 corresponds to a frequency of $2/(N_i T_i)$. Negative frequencies are stored in the reverse order of positive frequencies, ranging from the highest to lowest negative frequencies.

Note

The FFT function can be performed on functions of up to eight (8) dimensions. If a function has n dimensions, IDL performs a transform in each dimension separately, starting with the first dimension and progressing sequentially to dimension n . For example, if the function has two dimensions, IDL first does the FFT row by row, and then column by column.

For an even number of points in the i^{th} dimension, the frequencies corresponding to the returned complex values are:

$$0, 1/(N_i T_i), 2/(N_i T_i), \dots, (N_i/2-1)/(N_i T_i), 1/(2T_i), -(N_i/2-1)/(N_i T_i), \dots, -1/(N_i T_i)$$

where $1/(2T_i)$ is the Nyquist critical frequency.

For an odd number of points in the i^{th} dimension, the frequencies corresponding to the returned complex values are:

$$0, 1/(N_i T_i), 2/(N_i T_i), \dots, (N_i/2-0.5)/(N_i T_i), -(N_i/2-0.5)/(N_i T_i), \dots, -1/(N_i T_i)$$

Arguments

Array

The array to which the Fast Fourier Transform should be applied. If *Array* is not of complex type, it is converted to complex type. The dimensions of the result are identical to those of *Array*. The size of each dimension may be any integer value and does not necessarily have to be an integer power of 2, although powers of 2 are certainly the most efficient.

Direction

Direction is a scalar indicating the direction of the transform, which is negative by convention for the forward transform, and positive for the inverse transform. If *Direction* is not specified, the forward transform is performed.

A normalization factor of $1/N$, where N is the number of points, is applied during the forward transform.

Note

When transforming from a real vector to complex and back, it is slightly faster to set *Direction* to 1 in the real to complex FFT.

Note also that the value of *Direction* is ignored if the INVERSE keyword is set.

Keywords

DIMENSION

Set this keyword to the dimension across which to calculate the FFT. If this keyword is not present or is zero, then the FFT is computed across all dimensions of the input array. If this keyword is present, then the FFT is only calculated only across a single dimension. For example, if the dimensions of *Array* are N1, N2, N3, and DIMENSION is 2, the FFT is calculated only across the second dimension.

DOUBLE

Set this keyword to a value other than zero to force the computation to be done in double-precision arithmetic, and to give a result of double-precision complex type. If DOUBLE is set equal to zero, computation is done in single-precision arithmetic and the result is single-precision complex. If DOUBLE is not specified, the data type of the result will match the data type of *Array*.

INVERSE

Set this keyword to perform an inverse transform. Setting this keyword is equivalent to setting the *Direction* argument to a positive value. Note, however, that setting INVERSE results in an inverse transform even if *Direction* is specified as negative.

OVERWRITE

If this keyword is set, and the *Array* parameter is a variable of complex type, the transform is done “in-place”. The result overwrites the previous contents of the variable. For example, to perform a forward, in-place FFT on the variable *a*:

```
a = FFT(a, -1, /OVERWRITE)
```

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Note

Specifically, FFT will use the thread pool to overlap the inner loops of the computation when used on data with dimensions which have factors of 2, 3, 4, or 5. The prime-number DFT does not use the thread pool, as doing so would yield a relatively small benefit for the complexity it would introduce. Our experience shows that the improvement in performance from using the thread pool for FFT is highly dependent upon many factors (data length and dimensions, single vs. double precision, operating system, and hardware) and can vary between platforms.

Examples

Display the log of the power spectrum of a 100-element index array by entering:

```
PLOT, /YLOG, ABS(FFT(FINDGEN(100), -1))
```

As a more complex example, display the power spectrum of a 100-element vector sampled at a rate of 0.1 seconds per point. Show the 0 frequency component at the center of the plot and label the abscissa with frequency:

```
; Define the number of points and the interval:
N = 100
T = 0.1

; Midpoint+1 is the most negative frequency subscript:
N21 = N/2 + 1

; The array of subscripts:
F = INDGEN(N)
; Insert negative frequencies in elements F(N/2 +1), ..., F(N-1):
F[N21] = N21 -N + FINDGEN(N21-2)

; Compute T0 frequency:
F = F/(N*T)
```

```

; Shift so that the most negative frequency is plotted first:
PLOT, /YLOG, SHIFT(F, -N21), SHIFT(ABS(FFT(F, -1)), -N21)

```

Compute the FFT of a two-dimensional image by entering:

```

; Create a cosine wave damped by an exponential.
n = 256
x = FINDGEN(n)
y = COS(x*!PI/6)*EXP(-((x - n/2)/30)^2/2)

; Construct a two-dimensional image of the wave.
z = REBIN(y, n, n)
; Add two different rotations to simulate a crystal structure.
z = ROT(z, 10) + ROT(z, -45)
WINDOW, XSIZE=540, YSIZE=540
LOADCT, 39
TVSCL, z, 10, 270

; Compute the two-dimensional FFT.
f = FFT(z)
logpower = ALOG10(ABS(f)^2) ; log of Fourier power spectrum.
TVSCL, logpower, 270, 270

; Compute the FFT only along the first dimension.
f = FFT(z, DIMENSION=1)
logpower = ALOG10(ABS(f)^2) ; log of Fourier power spectrum.
TVSCL, logpower, 10, 10

; Compute the FFT only along the second dimension.
f = FFT(z, DIMENSION=2)
logpower = ALOG10(ABS(f)^2) ; log of Fourier power spectrum.
TVSCL, logpower, 270, 10

```

Version History

Introduced: Original

See Also

[HILBERT](#)

FILE_BASENAME

The FILE_BASENAME function returns the *basename* of a *file path*. A file path is a string containing one or more segments consisting of names separated by directory delimiter characters (slash (/) under UNIX, or backslash (\) under Microsoft Windows). The basename is the final rightmost segment of the file path; it is usually a file, but can also be a directory name. See “Rules used by FILE_BASENAME” on page 662 for additional information.

Note

FILE_BASENAME operates on strings based strictly on their syntax. The *Path* argument need not refer to actual or existing files.

FILE_BASENAME is based on the standard UNIX `basename(1)` utility.

Note

To retrieve the leftmost portion of the file path (the *dirname*), use the [FILE_DIRNAME](#) function.

Syntax

Result = FILE_BASENAME(*Path* [, *RemoveSuffix*] [, /[FOLD_CASE](#)])

Return Value

A scalar string or string array containing the basename for each element of the *Path* argument.

Arguments

Path

A scalar string or string array containing paths for which the basename is desired.

Note

Under Microsoft Windows, the backslash (\) character is used to separate directories within a path. For compatibility with UNIX, and general convenience, the forward slash (/) character is also accepted as a directory separator in the *Path* argument.

RemoveSuffix

An optional scalar string or 1-element string array specifying a filename suffix to be removed from the end of the basename, if present.

Note

If the entire basename string matches the suffix, the suffix is *not* removed.

Keywords

FOLD_CASE

By default, FILE_BASENAME follows the case sensitivity policy of the underlying operating system when attempting to match a string specified by the *RemoveSuffix* argument. By default, matches are case sensitive on UNIX platforms, and case insensitive on Microsoft Windows platforms. The FOLD_CASE keyword is used to change this behavior. Set it to a non-zero value to cause FILE_BASENAME to do all string matching case insensitively. Explicitly set FOLD_CASE equal to zero to cause all string matching to be case sensitive.

Note

The value of the FOLD_CASE keyword is ignored if the *RemoveSuffix* argument is not present.

Rules used by FILE_BASENAME

FILE_BASENAME makes a copy of the input file path string, then modifies the copy according to the following rules:

- If *Path* is a NULL string, then FILE_BASENAME returns a NULL string.
- If *Path* consists entirely of directory delimiter characters, the result of FILE_BASENAME is a single directory delimiter character.
- If there are any trailing directory delimiter characters, they are removed.
- Under Microsoft Windows, remove any of the following, if present:
 - The drive letter and colon (for file paths of the form `c:\directory\file`).
 - The initial double-backslash and host name (for UNC file paths of the form `\\host\share\directory\file`).

- If any directory delimiter characters remain, all characters up to and including the last directory delimiter are removed.
- If the *RemoveSuffix* argument is present, is not identical to the characters remaining, and matches the *suffix* of the characters remaining, the suffix is removed. Otherwise, the *Result* is not modified by this step. The case sensitivity of the string comparison used in this step is controlled by the `FOLD_CASE` keyword.

Examples

The following command prints the basename of an IDL `.pro` file, removing the `.pro` suffix:

```
PRINT, FILE_BASENAME('/usr/local/rsi/idl/lib/dist.pro', '.pro')
```

IDL prints:

```
dist
```

Similarly, the following command prints the basenames of all `.pro` files in the `lib` subdirectory of the IDL distribution that begin with the letter “I,” performing a case insensitive match for the suffix:

```
PRINT, FILE_BASENAME(FILE_SEARCH(FILEPATH('lib')+ '/i*.pro'),  
                      '.pro', /FOLD_CASE)
```

Version History

Introduced: 6.0

See Also

[FILE_DIRNAME](#), [PATH_SEP](#), [STREGEX](#), [STRMID](#), [STRPOS](#), [STRSPLIT](#)

FILE_CHMOD

The FILE_CHMOD procedure allows you to change the current access permissions (sometimes known as modes on UNIX platforms) associated with a file or directory. File modes are specified using the standard Posix convention of three protection classes (user, group, other), each containing three attributes (read, write, execute). These permissions can be specified as an octal bitmask in which desired permissions have their associated bit set and unwanted ones have their bits cleared. This is the same format familiar to users of the UNIX `chmod(1)` command).

Keywords are available to specify permissions without the requirement to specify a bitmask, providing a simpler way to handle many situations. All of the keywords share a similar behavior: Setting them to a non-zero value adds the specified permission to the *Mode* argument. Setting the keyword to 0 removes that permission.

To find the current protection settings for a given file, you can use the GET_MODE keyword to the FILE_TEST function.

Syntax

```
FILE_CHMOD, File [, Mode] [, /A_EXECUTE | /A_READ | /A_WRITE]
  [, /G_EXECUTE | /G_READ | /G_WRITE] [, /NOEXPAND_PATH]
  [, /O_EXECUTE | /O_READ | /O_WRITE]
  [, /U_EXECUTE | /U_READ | /U_WRITE]
```

UNIX-Only Keywords: [, /SETGID] [, /SETUID] [, /STICKY_BIT]

Arguments

File

A scalar or array of file or directory names for which protection modes will be changed.

Mode

An optional bit mask specifying the absolute protection settings to be applied to the files. If *Mode* is not supplied, FILE_CHMOD looks up the current modes for the file and uses it instead. Any additional modes specified via keywords are applied relative to the value in *Mode*. Setting a keyword adds the necessary mode bits to *Mode*, and clearing it by explicitly setting a keyword to 0 removes those bits from *Mode*.

The values of the bits in these masks correspond to those used by the UNIX `chmod(2)` system call and `chmod(1)` user command, and are given in the following table. Since these bits are usually manipulated in groups of three, octal notation is commonly used when referring to them. When constructing a mode, the following platform specific considerations should be kept in mind:

- The `setuid`, `setgid`, and sticky bits are specific to the UNIX operating system, and have no meaning elsewhere. `FILE_CHMOD` ignores them on non-UNIX systems. The UNIX kernel may quietly refuse to set the sticky bit if you are not the root user. Consult the `chmod(2)` man page for details.
- The Microsoft Windows operating system does not have 3 permission classes like UNIX does. Therefore, setting for all three classes are combined into a single request.
- The Microsoft Windows operating system always allows read access to any files visible to a program. `FILE_CHMOD` therefore ignores any requests to remove read access.
- The Microsoft Windows operating system does not maintain an execute bit for files, but instead uses the file suffix to decide if a file is executable. `FILE_CHMOD` cannot change the execution status of a file in the Windows environment; such requests are quietly ignored.

Bit	Octal Mask	Meaning
12	'4000' o	Setuid: Set user ID on execution.
11	'2000' o	Setgid: Set group ID on execution.
10	'1000' o	Turn on sticky bit. See the UNIX documentation on <code>chmod(2)</code> for details.
9	'0400' o	Allow read by owner.
8	'0200' o	Allow write by owner.
7	'0100' o	Allow execute by owner.
6	'0040' o	Allow read by group.
5	'0020' o	Allow write by group.
4	'0010' o	Allow execute by group.
3	'0004' o	Allow read by others.

Table 3-37: UNIX `chmod(2)` mode bits

Bit	Octal Mask	Meaning
2	'0002' o	Allow write by others.
1	'0001' o	Allow execute by others.

Table 3-37: UNIX `chmod(2)` mode bits (Continued)

Keywords

A_EXECUTE

Execute access for all three (user, group, other) categories.

A_READ

Read access for all three (user, group, other) categories.

A_WRITE

Write access for all three (user, group, other) categories.

G_EXECUTE

Execute access for the group category.

G_READ

Read access for the group category.

G_WRITE

Write access for the group category.

NOEXPAND_PATH

Set this keyword to cause `FILE_CHMOD` to use the *File* argument exactly as specified, without applying the usual file path expansion.

O_EXECUTE

Execute access for the other category.

O_READ

Read access for the other category.

O_WRITE

Write access for the other category.

SETGID

This keyword is only available on UNIX platforms.

The Set Group ID bit.

SETUID

This keyword is only available on UNIX platforms.

The Set User ID bit.

STICKY_BIT

This keyword is only available on UNIX platforms.

Sets the sticky bit.

U_EXECUTE

Execute access for the user category.

U_READ

Read access for the user category.

U_WRITE

Write access for the user category.

Examples

In the first example, we make the file `moose.dat` read only to everyone except the owner of the file, but not change any other settings:

```
FILE_CHMOD, 'moose.dat', /U_WRITE, G_WRITE=0, O_WRITE=0
```

In the next example, we make the file readable and writable to the owner and group, but read-only to anyone else, and remove any other modes:

```
FILE_CHMOD, 'moose.dat', '664'o
```

Version History

Introduced: 5.4

FILE_COPY

The FILE_COPY procedure copies files, or directories of files, to a new location. The copies retain the permission settings of the original files, and belong to the user that performed the copy. See “Rules Used By FILE_COPY” on page 671 for additional information.

FILE_COPY copies files based on their names. To copy data between open files, see the COPY_LUN procedure.

Syntax

```
FILE_COPY, SourcePath, DestPath [, /ALLOW_SAME] [, /NOEXPAND_PATH]
    [, /OVERWRITE] [, /RECURSIVE] [, /REQUIRE_DIRECTORY]
    [, /VERBOSE]
```

```
UNIX-Only Keywords: [, /COPY_NAMED_PIPE] [, /COPY_SYMLINK]
    [, /FORCE]
```

Arguments

SourcePath

A scalar string or string array containing the names of the files or directories to be copied.

Note

If *SourcePath* contains a directory, the RECURSIVE keyword must be set.

DestPath

A scalar string or string array containing the names of the destinations to which the files and directories specified by *SourcePath* are to be copied. If more than one file is to be copied to a given destination, that destination must exist and be a directory.

Keywords

ALLOW_SAME

Attempting to copy a file on top of itself by specifying the same file for *SourcePath* and *DestPath* is usually considered to be an error. If the ALLOW_SAME keyword is set, no copying is done and the operation is considered successful.

COPY_NAMED_PIPE

This keyword is only available on UNIX platforms.

When `FILE_COPY` encounters a UNIX *named pipe* (also called a *fifo*) in *SourcePath*, it usually opens it as a regular file and attempts to copy data from it to the destination file. If `COPY_NAMED_PIPE` is set, `FILE_COPY` will instead replicate the pipe, creating a new named pipe at the destination using the system `mkfifo()` function.

COPY_SYMLINK

This keyword is only available on UNIX platforms.

When `FILE_COPY` encounters a UNIX *symbolic link* in *SourcePath*, it attempts to copy the file or directory pointed to by the link. If `COPY_SYMLINK` is set, `FILE_COPY` will instead create a symbolic link at the destination with the same name as the source symbolic link, and pointing to the same path as the source.

FORCE

This keyword is only available on UNIX platforms.

Even if the `OVERWRITE` keyword is set, `FILE_COPY` does not overwrite files that have their file permissions set to prevent it. If the `FORCE` keyword is set, such files are quietly removed to make way for the overwrite operation to succeed.

Note

`FORCE` does not imply `OVERWRITE`; both must be specified to overwrite a protected file.

NOEXPAND_PATH

Set this keyword to cause `FILE_COPY` to use *SourcePath* and *DestPath* exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See [FILE_SEARCH](#) for details on path expansion.

OVERWRITE

Set this keyword to allow `FILE_COPY` to overwrite an existing file.

RECURSIVE

Set this keyword to cause directories specified by *SourcePath* to be copied to *DestPath* recursively, preserving the hierarchy and names of the files from the source.

If *SourcePath* includes one or more directories, the `RECURSIVE` keyword *must* be set.

Note

On a UNIX system, when performing a recursive copy on a directory hierarchy that includes files that are links to other files, the destination files will be copies, not links. Setting the `COPY_SYMLINK` keyword will cause files that are *symbolic* links to be copied as symbolic links, but `FILE_COPY` does not include a similar facility for copying *hard* links. See the description of the [FILE_LINK](#) for more information on UNIX file links.

REQUIRE_DIRECTORY

Set this keyword to cause `FILE_COPY` to require that *DestPath* exist and be a directory.

VERBOSE

Set this keyword to cause `FILE_COPY` to issue an informative message for every file copy operation it carries out.

Rules Used By FILE_COPY

The following rules govern how `FILE_COPY` operates:

- The arguments to `FILE_COPY` can be scalar or array. If both arguments are arrays, the arrays must contain the same number of elements; in this case, the files are copied pairwise, with each file from *SourcePath* being copied to the corresponding file in the *DestPath*. If *SourcePath* is an array and *DestPath* is a scalar, all files in *SourcePath* are copied to the single location given by *DestPath*, which must exist and be a directory.
- Elements of *SourcePath* may use wildcard characters (as accepted by the [FILE_SEARCH](#) function) to specify multiple files. All the files matched for a given element of *SourcePath* are copied to the location specified by the corresponding element of *DestPath*. If multiple files are copied to a single element of *DestPath*, that element must exist and be a directory.
- If a file specified in *DestPath* does not exist, the corresponding file from *SourcePath* is copied using the name specified by *DestPath*. Any parent directories to the file specified by *DestPath* must already exist.
- If *DestPath* names an existing regular file, `FILE_COPY` will not overwrite it, unless the `OVERWRITE` keyword is specified.

- If *DestPath* names an existing directory and *SourcePath* names a regular (non-directory) file, then `FILE_COPY` creates a file with the same name as the file given by *SourcePath* within the *DestPath* directory.
- If *DestPath* specifies an existing directory and *SourcePath* also names a directory, and the `RECURSIVE` keyword is set, `FILE_COPY` checks for the existence of a subdirectory of *DestPath* with the same name as the source directory. If this subdirectory does not exist, it is created using the same permissions as the directory being copied. Then, all the files and directories underneath the source directory are copied to this subdirectory. `FILE_COPY` will refuse to overwrite existing files within the destination subdirectory unless the `OVERWRITE` keyword is in effect.

Examples

Make a backup copy of a file named `myroutine.pro` in the current working directory:

```
FILE_COPY, 'myroutine.pro', 'myroutine.pro.backup'
```

Create a subdirectory named `BACKUP` in the current working directory and copy all `.pro` files, `makefile`, and `mydata.dat` into it:

```
FILE_MKDIR, 'BACKUP'  
FILE_COPY, ['*.pro', 'makefile', 'mydata.dat'], 'BACKUP'
```

Version History

Introduced: 5.6

See Also

[COPY_LUN](#), [FILE_LINK](#), [FILE_MOVE](#)

FILE_DELETE

The FILE_DELETE procedure deletes a file or empty directory, if the process has the necessary permissions to remove the file as defined by the current operating system. FILE_CHMOD can be used to change file protection settings.

Note

On UNIX, if a file to be deleted is a symbolic link, FILE_DELETE deletes the link itself, and not the file that the link points to.

Operating System Syntax

The syntax used to specify directories for removal depends on the operating system in use, and is in general the same as you would use when issuing commands to the operating system command interpreter.

Microsoft Windows users must be careful to not specify a trailing backslash at the end of a specification. For example:

```
FILE_DELETE, 'c:\mydir\myfile'
```

and not:

```
FILE_DELETE, 'c:\mydir\myfile\'
```

Syntax

```
FILE_DELETE, File1 [, ... Filen] [, /ALLOW_NONEXISTENT]  
[, /NOEXPAND_PATH] [, /QUIET] [, /RECURSIVE] [, /VERBOSE]
```

Arguments

File_{*j*}

A scalar or array of file or directory names to be deleted, one name per string element. Directories must be specified in the native syntax for the current operating system. See “Operating System Syntax” below for additional details.

Keywords

ALLOW_NONEXISTENT

If set, `FILE_DELETE` will quietly ignore attempts to delete a non-existent file. Other errors will still be reported. The `QUIET` keyword can be used instead to suppress all errors.

NOEXPAND_PATH

Set this keyword to cause `FILE_DELETE` to use the *File* argument exactly as specified, without applying the usual file path expansion.

QUIET

`FILE_DELETE` will normally issue an error if it is unable to remove a requested file or directory. If `QUIET` is set, no error is issued and `FILE_DELETE` simply moves on to the next requested item.

RECURSIVE

By default, `FILE_DELETE` will refuse to delete directories that are not empty. If `RECURSIVE` is set, `FILE_DELETE` will instead quietly delete all files contained within that directory and any subdirectories below it, and then remove the directory itself.

Warning

Recursive delete is a very powerful and useful operation. However, it is a relatively dangerous command with the ability to rapidly destroy a great deal of data. Once deleted, files cannot be recovered unless you have a separate backup, so a mistaken recursive delete can be very damaging. Be very careful to specify correct arguments to `FILE_DELETE` when using the `RECURSIVE` keyword.

VERBOSE

The `VERBOSE` keyword causes `FILE_DELETE` to issue an informative message for every file it deletes.

Examples

In this example, we remove an empty directory named `moose`:

```
FILE_DELETE, 'moose'
```

Version History

Introduced: 5.4

ALLOW_NONEXISTENT and VERBOSE keywords added: 5.6

FILE_DIRNAME

The FILE_DIRNAME function returns the *dirname* of a *file path*. A file path is a string containing one or more segments consisting of names separated by directory delimiter characters (slash (/) under UNIX, or backslash (\) under Microsoft Windows). The dirname is all of the file path except for the final rightmost segment, which is usually a file name, but can also be a directory name. See “[Rules use by FILE_DIRNAME](#)” on page 677 for additional information.

Note

FILE_DIRNAME operates on strings based strictly on their syntax. The *Path* argument need not refer to actual or existing files.

FILE_DIRNAME is based on the standard Unix `dirname(1)` utility.

Note

To retrieve the rightmost portion of the file path (the *basename*), use the [FILE_BASENAME](#) function.

Syntax

Result = FILE_DIRNAME(*Path* [, /[MARK_DIRECTORY](#)])

Return Value

A scalar string or string array containing the *dirname* for each element of the *Path* argument.

Note

By default, the *dirname* does not include a final directory separator character; this behavior can be changed using the [MARK_DIRECTORY](#) keyword.

Note

On Windows platforms, the string returned by FILE_DIRNAME always uses the backslash (\) as the directory separator character, even if the slash (/) was used in the *Path* argument.

Arguments

Path

A scalar string or string array containing paths for which the dirname is desired.

Note

Under Microsoft Windows, the backslash (\) character is used to separate directories within a path. For compatibility with UNIX, and general convenience, the forward slash (/) character is also accepted as a directory separator in the *Path* argument. However, all *results* produced by FILE_DIRNAME on Windows platforms use the standard backslash for this purpose, regardless of the separator character used in the input *Path* argument.

Keywords

MARK_DIRECTORY

Set this keyword to include a directory separator character at the end of the returned directory name string. Including the directory character allows you to concatenate a file name to the end of the directory name string without having to supply the separator character manually. This is convenient for cross platform programming, as the separator characters differ between operating systems.

Rules use by FILE_DIRNAME

FILE_DIRNAME makes a copy of the input path string, and then modifies the copy according to the following rules:

- If *Path* is a NULL string, then FILE_DIRNAME returns a single dot (.) character, representing the current working directory of the IDL process.
- Under Microsoft Windows, a file path can start with either of the following:
 - A drive letter and a colon (for file paths of the form `c:\directory\file`).
 - An initial double-backslash and a host name (for UNC file paths of the form `\\host\share\directory\file`).

If either of these are present in *Path*, they are considered to be part of the dirname, and are copied to the result *without interpretation by the remaining steps below*.

- If *Path* consists entirely of directory delimiter characters, the result of `FILE_DIRNAME` is a single directory delimiter character (prefixed by a Windows drive letter and colon or a UNC prefix, if necessary).
- All characters to the right of the rightmost directory delimiter character are removed.
- All trailing directory delimiter characters are removed.
- If the `MARK_DIRECTORY` keyword is set, a single directory delimiter character is appended to the end.

Examples

The following statements print the directory in which IDL locates the file `dist.pro` when it needs a definition for the `DIST` function. (`DIST` is part of the standard IDL user library, included with IDL):

```
temp = DIST(4) ; Ensure that DIST is compiled
PRINT, FILE_DIRNAME((ROUTINE_INFO('DIST', $
    /FUNCTION, /SCOURE)).path)
```

Depending on the platform and location where IDL is installed, IDL prints something like:

```
/usr/local/rsi/idl/lib
```

Version History

Introduced: 6.0

See Also

[FILE_BASENAME](#), [PATH_SEP](#), [STREGEX](#), [STRMID](#), [STRPOS](#), [STRSPLIT](#)

FILE_EXPAND_PATH

The FILE_EXPAND_PATH function expands a given file or partial directory name to its fully qualified name regardless of the current working directory.

Note

This routine should be used only to make sure that file paths are fully qualified, but not to expand wildcard characters (e.g. *). The behavior of FILE_EXPAND_PATH when it encounters a wildcard is platform dependent, and should not be depended on. These differences are due to the underlying operating system, and are beyond IDL's control. To expand wildcards and obtain fully qualified paths, use the FILE_SEARCH function with the FULLY_QUALIFY_PATH keyword:

```
A = FILE_SEARCH( '*.pro', /FULLY_QUALIFY_PATH)
```

Syntax

Result = FILE_EXPAND_PATH (*Path*)

Return Value

FILE_EXPAND_PATH returns a fully qualified file path that completely specifies the location of *Path* without the need to consider the user's current working directory. If *Path* is a NULL string, FILE_EXPAND_PATH returns the fully qualified path to the current working directory.

Arguments

Path

A scalar or array of file or directory names to be fully qualified.

Keywords

None.

Examples

In this example, we change directories to the IDL lib directory and expand the file path for the DIST function:

```
cd, FILEPATH('', SUBDIRECTORY=['lib'])  
print, FILE_EXPAND_PATH('dist.pro')
```

This results in the following if run on a UNIX system:

```
/usr/local/rsi/idl_6.1/lib/dist.pro
```

Version History

Introduced: 5.4

See Also

[FILE_SEARCH](#)

FILE_INFO

The FILE_INFO function returns status information about a specified file.

Syntax

Result = FILE_INFO(*Path* [, /NOEXPAND_PATH])

Return Value

The FILE_INFO function returns a structure expression of type FILE_INFO containing status information about the specified file or files. The result will contain one structure for each element in the *Path* argument.

Fields of the FILE_INFO Structure

The FILE_INFO structure consists of the following fields:

Field Name	Meaning
NAME	The name of the file.
EXISTS	True (1) if the file exists. False (0) if it does not exist.
READ	True (1) if the file exists and is readable by the user. False (0) if it is not readable.
WRITE	True (1) if the file exists and is writable by the user. False (0) if it is not writable.
EXECUTE	True (1) if the file exists and is executable by the user. False (0) if it is not executable. The source of this information differs between operating systems: UNIX: IDL checks the execute bit maintained by the operating system. Microsoft Windows: The determination is made on the basis of the file name extension (e.g. .exe).

Table 3-38: Fields of the FILE_INFO Structure

Field Name	Meaning
REGULAR	True (1) if the file exists and is a regular disk file and not a directory, pipe, socket, or other special file type. False (0) if it is not a regular disk file (it maybe a directory, pipe, socket, or other special file type).
DIRECTORY	True (1) if the file exists and is a directory. False (0) if it is not a directory.
BLOCK_SPECIAL	True (1) if the file exists and is a UNIX block special device. On non-UNIX operating systems, this field will always be False (0).
CHARACTER_SPECIAL	True (1) if the file exists and is a UNIX character special device. On non-UNIX operating systems, this field will always be False (0).
NAMED_PIPE	True (1) if the file exists and is a UNIX named pipe (fifo) device. On non-UNIX operating systems, this field will always be False (0).
SETGID	True (1) if the file exists and has its Set-Group-ID bit set. On non-UNIX operating systems, this field will always be False (0).
SETUID	True (1) if the file exists and has its Set-User-ID bit set. On non-UNIX operating systems, this field will always be False (0).
SOCKET	True (1) if the file exists and is a UNIX domain socket. On non-UNIX operating systems, this field will always be False (0).
STICKY_BIT	True (1) if the file exists and has its sticky bit set. On non-UNIX operating systems, this field will always be False (0).
SYMLINK	True (1) if the file exists and is a UNIX symbolic link. On non-UNIX operating systems, this field will always be False (0).

Table 3-38: Fields of the FILE_INFO Structure (Continued)

Field Name	Meaning
DANGLING_SYMLINK	True (1) if the file exists and is a UNIX symbolic link that points at a non-existent file. On non-UNIX operating systems, this field will always be False (0).
ATIME, CTIME, MTIME	The date of last access, date of creation, and date of last modification given in seconds since 1 January 1970 UTC. Use the SYSTIME function to convert these dates into a textual representation. Note that some file systems do not maintain all of these dates (e.g. MS DOS FAT file systems), and may return 0. On some non-UNIX operating systems, access time is not maintained, and ATIME and MTIME will always return the same date.
SIZE	The current length of the file in bytes. If <i>Path</i> is not to a regular file (possibly to a directory, pipe, socket, or other special file type), the value of SIZE will not contain any useful information.

Table 3-38: Fields of the FILE_INFO Structure (Continued)

Arguments

Path

A string or string array containing the path or paths to the file or files about which information is required.

Keywords

NOEXPAND_PATH

If specified, FILE_INFO uses *Path* exactly as specified, without applying the usual file path expansion.

Examples

To get information on the file `dist.pro` within the IDL User Library:

```
HELP,/STRUCTURE, FILE_INFO(FILEPATH('dist.pro', $
    SUBDIRECTORY = 'lib'))
```

Executing the above command will produce output similar to:

```
** Structure FILE_INFO, 21 tags, length=72:
NAME                STRING      '/usr/local/rsi/idl/lib/dist.pro'
EXISTS              BYTE        1
READ                BYTE        1
WRITE               BYTE        0
EXECUTE             BYTE        0
REGULAR             BYTE        1
DIRECTORY           BYTE        0
BLOCK_SPECIAL       BYTE        0
CHARACTER_SPECIAL   BYTE        0
NAMED_PIPE          BYTE        0
SETGID              BYTE        0
SETUID              BYTE        0
SOCKET              BYTE        0
STICKY_BIT          BYTE        0
SYMLINK             BYTE        0
DANGLING_SYMLINK   BYTE        0
MODE                LONG        420
ATIME               LONG64     970241431
CTIME               LONG64     970241595
MTIME               LONG64     969980845
SIZE                LONG64     1717
```

Version History

Introduced: 5.5

See Also

[FILE_TEST](#), [FSTAT](#)

FILE_LINES

The `FILE_LINES` function reports the number of lines of text contained within the specified file or files.

Text files containing data are very common. To read such a file usually requires knowing how many lines of text it contains. Under UNIX and Windows, there is no special text file type, and it is not possible to tell how many lines are contained in a file from basic file attributes. Rather, lines are encoded using a special character or characters at the end of each line:

- UNIX operating systems use an ASCII linefeed (LF) character at the end of each line.
- Older Macintosh systems (prior to the UNIX-based Mac OS X) use a carriage return (CR).
- Microsoft Windows uses a two character CR/LF sequence.

The only way to determine the number of lines of text contained within a file is to open it and count lines while reading and skipping over them until the end of the file is encountered. Since files are often copied from one type of system to another without going through the proper line termination conversion, portable software needs to be able to recognize any of these terminations, regardless of the system being used. `FILE_LINES` performs this operation in an efficient and portable manner, handling all three of the line termination conventions listed above.

This routine works by opening the file and reading the data contained within. It is therefore only suitable for regular disk files, and only when access to that file is fast enough to justify reading it more than once. For other types of files, other approaches are necessary, such as:

- Reading the file once, using an adaptive (expandable) data structure, counting the number of lines as they are input, and growing the data structure as necessary.
- Building a header into your file format that includes the necessary information, or somehow embedding the number of lines into the file data.
- Maintaining file information in a separate file associated with each file.
- Using a self describing data format that avoids these issues.

This routine assumes that the specified file or files contain only lines of text. It is unable to correctly count lines in files that contain binary data, or which do not use the standard line termination characters. Results are undefined for such files.

Note that `FILE_LINES` is equivalent to the following IDL code:

```

FUNCTION file_lines, filename
  OPENR, unit, filename, /GET_LUN
  str = ''
  count = 0ll
  WHILE ~ EOF(unit) DO BEGIN
    READF, unit, str
    count = count + 1
  ENDWHILE
  FREE_LUN, unit
  RETURN, count
END

```

The primary advantage of `FILE_LINES` over the IDL version shown here is efficiency. `FILE_LINES` is able to avoid the overhead of the `WHILE` loop as well as not having to create an IDL string for each line of the file.

Syntax

Result = `FILE_LINES(Path [,/COMPRESS] [,/NOEXPAND_PATH])`

Return Value

Returns the number of lines of text contained within the specified file or files. If an array of file names is specified via the *Path* parameter, the return value is an array with the same number of elements as *Path*, with each element containing the number of lines in the corresponding file.

Arguments

Path

A scalar string or string array containing the names of the text files for which the number of lines is desired.

Keywords

COMPRESS

If this keyword is set, `FILE_LINES` assumes that the files specified in *Path* contain data compressed in the standard GZIP format, and decompresses the data in order to count the number of lines. See the description of the `COMPRESS` keyword to the `OPEN` procedure for additional information.

NOEXPAND_PATH

If this keyword is set, `FILE_LINES` uses *Path* exactly as specified, without expanding any wildcard characters or environment variable names included in the path. See [FILE_SEARCH](#) for details on path expansion.

Examples

Read the contents of the text file `mydata.dat` into a string array.

```
nlines = FILE_LINES('mydata.dat')
sarr = STRARR(nlines)
OPENR, unit, 'mydata.dat', /GET_LUN
READF, unit, sarr
FREE_LUN, unit
```

Version History

Introduced: 5.6

COMPRESS keyword added: 6.0

See Also

[OPEN](#), [READ/READF](#)

FILE_LINK

The FILE_LINK procedure creates UNIX file links, both regular (hard) and symbolic. FILE_LINK is available only under UNIX.

A hard link is a directory entry that references a file. UNIX allows multiple such links to exist simultaneously, meaning that a given file can be referenced by multiple names. All such links are fully equivalent references to the same file (there are no concepts of primary and secondary names). All files carry a reference count that contains the number of hard links that point to them; deleting a link to a file does not remove the actual file from the filesystem until the last hard link to the file is removed. The following limitations on hard links are enforced by the operating system:

- Hard links may not span filesystems, as hard linking is only possible within a single filesystem.
- Hard links may not be created between directories, as doing so has the potential to create infinite circular loops within the hierarchical UNIX filesystem. Such loops will confuse many system utilities, and can even cause filesystem damage.

A symbolic link is an indirect pointer to a file; its directory entry contains the name of the file to which it is linked. Symbolic links may span filesystems and may refer to directories.

Many users find symbolic links easier to understand and use. Due to their generality and lack of restriction, RSI recommends their use over hard links for most purposes. FILE_LINK creates symbolic links by default.

See “[Rules Used by FILE_LINK](#)” on page 689 for information on how FILE_LINK interprets its arguments.

Syntax

```
FILE_LINK, SourcePath, DestPath [, /ALLOW_SAME] [, /HARDLINK]
      [, /NOEXPAND_PATH] [, /VERBOSE]
```

Arguments

SourcePath

A scalar string or string array containing the names of the files or directories to be linked.

DestPath

A scalar string or string array containing the names of the destinations to which the files and directories given by *SourcePath* are to be linked. If more than one file is to be linked to a given destination, that destination must exist and be a directory.

Keywords

ALLOW_SAME

Attempting to link a file to itself by specifying the same file for *SourcePath* and *DestPath* is usually considered to be an error. If the `ALLOW_SAME` keyword is set, no link is created and the operation is considered to be successful.

HARDLINK

Set this keyword to create hard links. By default, `FILE_LINK` creates symbolic links.

NOEXPAND_PATH

Set this keyword to cause `FILE_LINK` to use *SourcePath* and *DestPath* exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See [FILE_SEARCH](#) for details on path expansion.

VERBOSE

Set this keyword to cause `FILE_LINK` to issue an informative message for every file link operation it carries out.

Rules Used by FILE_LINK

The following rules govern how `FILE_LINK` operates:

- The arguments to `FILE_LINK` can be scalar or array. If both arguments are arrays, they must contain the same number of elements, and the files are paired, with each file from *SourcePath* being linked to the corresponding file in the *DestPath*. If *SourcePath* is an array and *DestPath* is a scalar, all links are created in the single location given by *DestPath*, which must exist and be a directory.
- Elements of *SourcePath* may use wildcard characters (as accepted by the [FILE_SEARCH](#) function) to specify multiple files. All the files matched for a given element of *SourcePath* are linked to the corresponding element of

DestPath. If multiple files are linked to a single element of *DestPath*, that element must exist and be a directory.

- If a file specified in *DestPath* does not exist, the corresponding file from *SourcePath* is linked using the name specified by *DestPath*. Any parent directories to the filename specified by *DestPath* must already exist.
- If *DestPath* names an existing regular file, `FILE_LINK` will not overwrite it.
- If *DestPath* names an existing directory, a link with the same name as the source file is created in the directory. This is primarily of interest with hard links.

Examples

Create a symbolic link named `current.dat` in the current working directory, pointing to the file `/master/data/saturn7.dat`:

```
FILE_LINK, '/master/data/saturn7.dat', 'current.dat'
```

Version History

Introduced: 5.6

See Also

[COPY_LUN](#), [FILE_COPY](#), [FILE_MOVE](#), [FILE_READLINK](#)

FILE_MKDIR

The FILE_MKDIR procedure creates a new directory, or directories, with the default access permissions for the current process.

Note

Use the FILE_CHMOD procedure to alter access permissions.

If a specified directory has non-existent parent directories, FILE_MKDIR automatically creates all the intermediate directories as well.

Syntax

```
FILE_MKDIR, File1 [... FileN] [, /NOEXPAND_PATH]
```

Arguments

FileN

A scalar or array of directory names to be created, one name per string element. Directories must be specified in the native syntax for the current operating system.

Keywords

NOEXPAND_PATH

Set this keyword to cause FILE_MKDIR to use the *File* argument exactly as specified, without applying the usual file path expansion.

Examples

To create a subdirectory named `moose` in the current working directory:

```
FILE_MKDIR, 'moose'
```

Version History

Introduced: 5.4

FILE_MOVE

The `FILE_MOVE` procedure renames files and directories, effectively moving them to a new location. The moved files retain their permission and ownership attributes. Within a given filesystem or volume, `FILE_MOVE` does not copy file data. Rather, it simply changes the file names by updating the directory structure of the filesystem. This operation is fast and safe, but is only possible within a single filesystem. Attempts to move a regular file from one filesystem to another are carried out by copying the file using `FILE_COPY`, and then deleting the original file. It is an error to attempt to use `FILE_MOVE` to move a directory from one filesystem to another.

See “[Rules Used by FILE_MOVE](#)” on page 693 for information on how `FILE_MOVE` interprets its arguments.

Syntax

```
FILE_MOVE, SourcePath, DestPath [, /ALLOW_SAME] [, /NOEXPAND_PATH]
      [, /OVERWRITE] [, /REQUIRE_DIRECTORY] [, /VERBOSE]
```

Arguments

SourcePath

A scalar string or string array containing the names of the files or directories to be moved.

DestPath

A scalar string or string array containing the names of the destinations to which the files and directories specified by *SourcePath* are to be moved. If more than one file is to be moved to a given destination, that destination must exist and be a directory.

Keywords

ALLOW_SAME

Attempting to move a file on top of itself by specifying the same file for *SourcePath* and *DestPath* is usually considered to be an error. If the `ALLOW_SAME` keyword is set, no renaming is done and the operation is considered to be successful.

NOEXPAND_PATH

Set this keyword to cause `FILE_MOVE` to use *SourcePath* and *DestPath* exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See [FILE_SEARCH](#) for details on path expansion.

OVERWRITE

Set this keyword to allow `FILE_MOVE` to overwrite an existing file.

REQUIRE_DIRECTORY

Set this keyword to cause `FILE_MOVE` to require that *DestPath* exist and be a directory.

VERBOSE

Set this keyword to cause `FILE_MOVE` to issue an informative message for every file move operation it carries out.

Rules Used by FILE_MOVE

The following rules govern how `FILE_MOVE` operates:

- The arguments to `FILE_MOVE` can be scalar or array. If both arguments are arrays, they must contain the same number of elements, and the files are moved in pairs, with each file from *SourcePath* being renamed to the corresponding file in the *DestPath*. If *SourcePath* is an array and *DestPath* is a scalar, all files in *SourcePath* are renamed to the single location given by *DestPath*, which must exist and be a directory.
- Elements of *SourcePath* may use wildcard characters (as accepted by the [FILE_SEARCH](#) function) to specify multiple files. All the files matched for that element of *SourcePath* are renamed to the location specified by the corresponding element of *DestPath*. If multiple files are renamed to a single element of *DestPath*, that element must exist and be a directory.
- If a file specified in *DestPath* does not exist, the corresponding file from *SourcePath* is moved using the name specified by *DestPath*. Any parent directories to the filename specified by *DestPath* must already exist.
- If *DestPath* names an existing regular file, `FILE_MOVE` will not overwrite it, unless the `OVERWRITE` keyword is specified.
- If *DestPath* names an existing directory and *SourcePath* names a regular (non-directory) file, the source file is moved into the specified directory.

- If *DestPath* specifies an existing directory and *SourcePath* also names a directory, `FILE_MOVE` checks for the existence of a subdirectory of *DestPath* with the same name as the source directory. If this subdirectory does not exist, the source directory is moved to the specified location. If the subdirectory does exist, an error is issued, and the rename operation is not carried out.

Examples

Rename the file `backup.dat` to `primary.dat` in the current working directory:

```
FILE_MOVE, 'backup.dat', 'primary.dat'
```

Create a subdirectory named `BACKUP` in the current working directory and move all `.pro` files, `makefile`, and `mydata.dat` into it:

```
FILE_MKDIR, 'BACKUP'  
FILE_MOVE, ['*.pro', 'makefile', 'mydata.dat'], 'BACKUP'
```

Version History

Introduced: 5.6

See Also

[COPY_LUN](#), [FILE_COPY](#), [FILE_LINK](#)

FILE_READLINK

The FILE_READLINK function returns the path pointed to by UNIX symbolic links.

Syntax

```
Result = FILE_READLINK(Path [, /ALLOW_NONEXISTENT]  
[, /ALLOW_NONSYMLINK] [, /NOEXPAND_PATH] )
```

Return Value

Returns the path associated with a symbolic link.

Arguments

Path

A scalar string or string array containing the names of the symbolic links to be translated.

Keywords

ALLOW_NONEXISTENT

Set this keyword to return a NULL string rather than throwing an error if *Path* contains a non-existent file.

ALLOW_NONSYMLINK

Set this keyword to return a NULL string rather than throwing an error if *Path* contains a path to a file that is not a symbolic link.

NOEXPAND_PATH

Set this keyword to cause FILE_READLINK to use *Path* exactly as specified, without expanding any wildcard characters or environment variable names included in the path. See [FILE_SEARCH](#) for details on path expansion.

Examples

Under Mac OS X, the `/etc` directory is actually a symbolic link. The following statement reads it and returns the location to which the link points:

```
path = FILE_READLINK('/etc')
```

It is possible to have chains of symbolic links, each pointing to another. The following function uses `FILE_READLINK` to iteratively translate such links until it finds the actual file:

```
FUNCTION RESOLVE_SYMLINK, path

    savepath = path          ; Remember last successful translation
    WHILE (path NE '') DO BEGIN
        path = FILE_READLINK(path, /ALLOW_NONEXISTENT, $
            /ALLOW_NONSYMLINK)
        IF (path NE '') THEN BEGIN
            ; If returned path is not absolute, use it to replace the
            ; last path segment of the previous path.
            IF (STRMID(path, 0, 1) NE '/') THEN BEGIN
                last = STRPOS(savepath, '/', /REVERSE_SEARCH)
                IF (last NE -1) THEN path = STRMID(savepath, 0, last) $
                    + '/' + path
            ENDIF
            savepath = path
        ENDIF
    ENDWHILE

    ; FILE_EXPAND_PATH removes redundant things like ../ from
    ; the result.
    RETURN, FILE_EXPAND_PATH(savepath)

END
```

Version History

Introduced: 5.6

See Also

[FILE_LINK](#)

FILE_SAME

It is common for a given file to be accessible via more than one name. For example, a relative path and a fully-qualified path to the same file are considered different names, since the strings that make up the paths are not lexically identical. In addition, under UNIX, the widespread use of links (hard and symbolic) makes multiple names for the same file very common.

The `FILE_SAME` function is used to determine if two different file names refer to the same underlying file.

The mechanism used to determine whether two names refer to the same file depends on the operating system in use:

UNIX: Under UNIX, all files are uniquely identified by two integer values: the filesystem that contains the file and the *inode number*, which identifies the file within the filesystem. If the input arguments are lexically identical, `FILE_SAME` will return True, regardless of whether the file specified actually exists. Otherwise, `FILE_SAME` compares the device and inode numbers of the two files, and returns True if they are identical, or False otherwise.

Windows: Unlike UNIX, Microsoft Windows identifies files solely by their names. `FILE_SAME` therefore expands the two supplied paths to their fully qualified forms, converts any “8.3 short names” to their real names, and then performs a simple case insensitive string comparison to determine if the paths are identical. (See “[Microsoft Windows “8.3 Short Names”](#)” on page 712 for more information on this subject). This is reliable for local disk files, but can produce incorrect results under some circumstances:

- UNC network paths can expand to different, but equivalent, paths. For example, a network server may be referred to by either a name or an IP address.
- Network attached storage can have mechanisms for giving multiple names to the same file, but to the Windows client system the names will appear to refer to different files. For example, a UNIX server using Samba software to serve files to machines on a Windows network can use symbolic links to produce two names for the same file, but these will appear as two distinct files to a Windows machine.

For these reasons, `FILE_SAME` is primarily of interest on UNIX systems. Under Windows, RSI recommends its use only on local files.

Syntax

```
Result = FILE_SAME(Path1, Path2 [, /NOEXPAND_PATH] )
```

Return Value

FILE_SAME returns True (1) if two filenames refer to the same underlying file, or False (0) otherwise. If either or both of the input arguments are arrays of file names, the result is an array, following the same rules as standard IDL operators.

Arguments

Path1, Path2

Scalar or array string values containing the two file paths to be compared.

Keywords

NOEXPAND_PATH

Set this keyword to cause FILE_SAME to use the *Path* arguments exactly as specified, without expanding any wildcard characters or environment variable names included in the paths. See [FILE_SEARCH](#) for details on path expansion. The utility of doing this depends on the operating system in use:

UNIX: Under UNIX, path expansion is not necessary unless the *Path* arguments use shell meta characters or environment variables.

Windows: By default, FILE_SAME expands the supplied paths to their fully qualified forms in order to be able to compare them. Preventing this path expansion cripples its ability to make a useful comparison, and is not recommended.

Examples

UNIX command shells often provide the HOME environment variable to point at the user's home directory. Many shells also expand the '~' character to point at the home directory. The following IDL statement determines if these two mechanisms refer to the same directory:

```
PRINT, FILE_SAME('~', '$HOME')
```

On a UNIX system, the following statement determines if the current working directory is the same as your home directory:

```
PRINT, FILE_SAME('.', '$HOME')
```

On some BSD-derived UNIX systems, the three commands `/bin/cp`, `/bin/ln`, and `/bin/mv` are actually three hard links to the same binary file. The following statement will print the number 1 if this is true on your system:

```
PRINT, TOTAL(FILE_SAME('/bin/cp', ['/bin/ln', '/bin/mv'])) EQ 2
```

Under Mac OS X, the `/etc` directory is actually a symbolic link to `/private/etc`. As a result, the following lines of code provide a simple test to determine whether Mac OS X is the current platform:

```
IF FILE_SAME('/etc', '/private/etc') THEN $  
  PRINT, 'Running Mac OS X' ELSE $  
  PRINT, 'Not Running Mac OS X'
```

Note

The above lines are shown simply as an example; checking the value of `!VERSION.OS_FAMILY` is a more reliable method of determining which operating system is in use.

Version History

Introduced: 5.6

See Also

[FILE_EXPAND_PATH](#), [FILE_INFO](#), [FILE_SEARCH](#), [FILE_TEST](#)

FILE_SEARCH

The `FILE_SEARCH` function returns a string array containing the names of all files matching the input path specification. Input path specifications may contain wildcard characters, enabling them to match multiple files. `FILE_SEARCH`'s interpretation of wildcard characters in the path specification is described in [“Supported Wildcards and Expansions”](#) on page 709.

A *relative path* is a path specification that can only be unambiguously interpreted by basing it relative to some other known location. Usually, this location is the current working directory for the process. A *fully qualified path* is a complete and unambiguous path that can be interpreted directly. For example, `bin/idl` is a relative path, while `/usr/local/rsi/idl/bin/idl` is a fully qualified path. By default, `FILE_SEARCH` follows the format of the input to decide whether to return relative or fully-qualified paths.

Note

In most cases, the operation of `FILE_SEARCH` is straightforward. There are, however, numerous options available; while these options make the routine more powerful, they may also make its behavior less intuitive. Read the keyword descriptions for additional details.

In addition, there are platform-specific behaviors of which you should be aware, especially if you work in a multiplatform environment. See [“Platform-Specific Filename Matching Issues”](#) on page 711 for details.

Syntax

Result = `FILE_SEARCH(Path_Specification)`

or for recursive searching,

Result = `FILE_SEARCH(Dir_Specification, Recur_Pattern)`

Keywords: `[, COUNT=variable] [, /EXPAND_ENVIRONMENT]`
`[, /EXPAND_TILDE] [, /FOLD_CASE] [, /FULLY_QUALIFY_PATH]`
`[, /ISSUE_ACCESS_ERROR] [, /MARK_DIRECTORY]`
`[, /MATCH_ALL_INITIAL_DOT | /MATCH_INITIAL_DOT] [, /NOSORT]`
`[, /QUOTE] [, /TEST_DIRECTORY] [, /TEST_EXECUTABLE]`
`[, /TEST_READ] [, /TEST_REGULAR] [, /TEST_WRITE]`
`[, /TEST_ZERO_LENGTH] [, /WINDOWS_SHORT_NAMES]`

UNIX-Only Keywords: [, /TEST_BLOCK_SPECIAL]
 [, /TEST_CHARACTER_SPECIAL] [, /TEST_DANGLING_SYMLINK]
 [, /TEST_GROUP] [, /TEST_NAMED_PIPE] [, /TEST_SETGID]
 [, /TEST_SETUID] [, /TEST_SOCKET] [, /TEST_STICKY_BIT]
 [, /TEST_SYMLINK] [, /TEST_USER]

Return Value

Returns all matched filenames in a string array, one file name per array element. If no files exist with names matching the input arguments, a null scalar string is returned instead of a string array.

If the input path is relative, the results will be relative. If the input is fully qualified, the results will also be fully qualified. If you specify the FULLY_QUALIFY_PATH keyword, the results will be fully qualified no matter which form of input is used. FILE_SEARCH returns results based on standard and recursive searches:

- **Standard:** When called with a single *Path_Specification* argument, FILE_SEARCH returns all files that match that specification. This is the same operation, sometimes referred to as *file globbing*, performed by most operating system command interpreters when wildcard characters are used in file specifications.
- **Recursive:** When called with two arguments, FILE_SEARCH performs recursive searching of directory hierarchies. In a recursive search, FILE_SEARCH looks recursively for any and all subdirectories in the file hierarchy rooted at the *Dir_Specification* argument. Within each of these subdirectories, it returns the names of all files that match the pattern in the *Recur_Pattern* argument. This operation is similar to that performed by the UNIX `find(1)` command.

Note

To avoid going into an infinite loop, the FILE_SEARCH routine does not follow symbolic links.

Arguments

Any of the arguments described in this section can contain wildcard characters, as described in “[Supported Wildcards and Expansions](#)” on page 709.

Path_Specification

A scalar or array variable of string type, containing file paths to match. If *Path_Specification* is not supplied, or if it is supplied as a null string, FILE_SEARCH uses a default pattern of ' * ', which matches all files in the current directory.

Dir_Specification

A scalar or array variable of string type, containing directory paths within which FILE_SEARCH will perform recursive searching for files matching the *Recur_Pattern* argument. FILE_SEARCH examines *Dir_Specification*, and any directory found below it, and returns the paths of any files in those directories that match *Recur_Pattern*. If *Dir_Specification* is supplied as a null string, FILE_SEARCH searches the current directory.

Recur_Pattern

A scalar string containing a pattern for files to match in any of the directories specified by the *Dir_Specification* argument. If *Recur_Pattern* is supplied as a null string, FILE_SEARCH uses a default pattern of ' * ', which matches all files in the specified directories.

Keywords

COUNT

A named variable into which the number of files found is placed. If no files are found, a value of 0 (zero) is returned.

EXPAND_ENVIRONMENT

By default, FILE_SEARCH follows the conventions of the underlying operating system to determine whether it should expand environment variable references in input file specification patterns. The default is to do such expansions under UNIX, and not to do them under Microsoft Windows. The EXPAND_ENVIRONMENT keyword is used to change this behavior. Set it to a non-zero value to cause FILE_SEARCH to perform environment variable expansion on all platforms. Set it to zero to disable such expansion.

The syntax for expanding environment variables in an input file pattern is based on that supported by the standard UNIX shell (`/bin/sh`), as described in [“Supported Wildcards and Expansions”](#) on page 709.

EXPAND_TILDE

Users of the UNIX C-shell (`/bin/csh`), and other tools influenced by it, are familiar with the use of a tilde (`~`) character at the beginning of a path to denote a home directory. A tilde by itself at the beginning of the path (e.g. `~/directory/file`) is equivalent to the home directory of the user executing the command, while a tilde followed by the name of a user (e.g. `~user/directory/file`) is expanded to the home directory of the named user.

By default, `FILE_SEARCH` follows the conventions of the underlying operating system in deciding whether to expand a leading tilde or to treat it as a literal character. Hence, the default is to expand the leading tilde under UNIX, and not under Microsoft Windows. The `EXPAND_TILDE` keyword is used to change this behavior.

Set `EXPAND_TILDE` to 0 (zero) to disable tilde expansion on all platforms. Set it to a non-zero value to enable tilde expansion.

Note

Under Microsoft Windows, only the plain form of tilde is recognized. Attempts to use the `~user` form will cause IDL to issue an error. IDL uses the `HOME` and `HOMEPATH` environment variables to obtain a home directory for the current Windows user.

FOLD_CASE

By default, `FILE_SEARCH` follows the case sensitivity policy of the underlying operating system. By default, matches are case sensitive on UNIX platforms, and case insensitive on Microsoft Windows platforms. The `FOLD_CASE` keyword is used to change this behavior. Set it to a non-zero value to cause `FILE_SEARCH` to do all file matching case insensitively. Explicitly set `FOLD_CASE` equal to zero to cause all file matching to be case sensitive.

RSI does not recommend changing the default value of `FOLD_CASE`, for the following reasons:

- Under UNIX, case-insensitive file searching (that is, setting `FOLD_CASE=1`) can lead to confusing behavior, since files with the same name in different combinations of upper- and lower-case letters are actually distinct files that can exist simultaneously in the same directory. However, case insensitivity can be useful under UNIX when combined with wildcards in order to find all instances of a given file type without regard to case. For example, the following will find all files in the current directory that end with a `.dat` extension without regard to the case of the extension:

```
datafiles = FILE_SEARCH('*.dat', /FOLD_CASE)
```

- Under Windows, case-sensitive file searching (that is, setting FOLD_CASE=0) is rarely useful, since files with the same name in different combinations of upper- and lower-case letters cannot exist simultaneously in the same directory, and a case-insensitive search will return any version.

FULLY_QUALIFY_PATH

If set, FILE_SEARCH expands all returned file paths so that they are complete. Under UNIX, this means that all files are specified relative to the root of the file system. On Windows platforms, it means that all files are specified relative to the drive on which they are located. By default, FILE_SEARCH returns fully qualified paths when the input specification is fully qualified, and returns relative paths otherwise. For example:

```
CD, '/usr/local/rsi/idl/bin'
PRINT, FILE_SEARCH('idl')
idl
PRINT, FILE_SEARCH('idl', /FULLY_QUALIFY_PATH)
/usr/local/rsi/idl/bin/idl
```

Under Microsoft Windows, any use of a drive letter colon (:) character implies full qualification, even if the path following the colon does not start with a slash character.

ISSUE_ACCESS_ERROR

If the IDL process lacks the necessary permission to access a directory included in the input specification, FILE_SEARCH will normally skip over it quietly and not include it in the generated results. Set ISSUE_ACCESS_ERROR to cause an error to be issued instead.

MARK_DIRECTORY

If set, all directory paths are returned with a path separator character appended to the end. This allows the caller to concatenate a file name directly to the end without having to supply a separator character first. This is convenient for cross-platform programming, as the separator characters differ between operating systems:

```
PRINT, FILE_SEARCH(!DIR)
/usr/local/rsi/idl
PRINT, FILE_SEARCH(!DIR, /MARK_DIRECTORY)
/usr/local/rsi/idl/
```

MATCH_ALL_INITIAL_DOT

By default, wildcards do not match leading dot (.) characters, and `FILE_SEARCH` does not return the names of files that start with the dot (.) character unless the leading dot is actually contained within the search string. Set `MATCH_ALL_INITIAL_DOT` to change this policy so that wildcards will match all files starting with a dot, including the special “.” (current directory) and “..” (parent directory) entries. RSI recommends the use of the `MATCH_INITIAL_DOT` keyword instead of `MATCH_ALL_INITIAL_DOT` for most purposes.

MATCH_INITIAL_DOT

`MATCH_INITIAL_DOT` serves the same function as `MATCH_ALL_INITIAL_DOT`, except that the special “.” (current directory) and “..” (parent directory) directories are not included.

NOSORT

Normally, `FILE_SEARCH` sorts the list of files returned by the operating system in a case-sensitive manner. If the `NOSORT` keyword is set, `FILE_SEARCH` will not perform the sort, instead returning exactly what is returned by the underlying operating system calls. On some operating systems, this can make `FILE_SEARCH` execute faster. The order of the results returned when `NOSORT` is set depends on the implementation of the operating system, and should not be relied upon.

QUOTE

`FILE_SEARCH` usually treats all wildcards found in the input specification as having the special meanings described in [“Supported Wildcards and Expansions”](#) on page 709. This means that such characters cannot normally be used as plain literal characters in file names. For example, it is not possible to match a file that contains a literal asterisk character in its name because asterisk is interpreted as the “match zero or more characters” wildcard.

If the `QUOTE` keyword is set, the backslash character can be used to escape any character so that it is treated as a plain character with no special meaning. In this mode, `FILE_SEARCH` replaces any two-character sequence starting with a backslash with the second character of the pair. In the process, any special wildcard meaning that character might have had disappears, and the character is treated as a literal.

If `QUOTE` is set, any literal backslash characters in your path must themselves be escaped with a backslash character. This is especially important for Microsoft Windows users, because the directory separator character for that platform is the

backslash. Windows IDL also accepts UNIX-style forward slashes for directory separators, so Windows users have two choices in handling this issue:

```
Result = FILE_SEARCH('C:\\home\\bob\\*.dat', /QUOTE)
Result = FILE_SEARCH('C:/home/bob/*.dat', /QUOTE)
```

On a Windows system, either of these options gives the path to a file named *.dat.

TEST_BLOCK_SPECIAL

This keyword is only available on UNIX platforms.

Only include a matching file if it is a block special device.

TEST_CHARACTER_SPECIAL

This keyword is only available on UNIX platforms.

Only include a matching file if it is a character special device.

TEST_DANGLING_SYMLINK

This keyword is only available on UNIX platforms.

Only include a matching file if it is a symbolic link that points at a non-existent file.

TEST_DIRECTORY

Only include a matching file if it is a directory.

TEST_EXECUTABLE

Only include a matching file if it is executable. The source of this information differs between operating systems:

UNIX: IDL checks the per-file information (the execute bit) maintained by the operating system.

Microsoft Windows: The determination is made on the basis of the file name extension (e.g. .exe).

TEST_GROUP

This keyword is only available on UNIX platforms.

Only include a matching file if it belongs to the same effective group ID (GID) as the IDL process.

TEST_NAMED_PIPE

This keyword is only available on UNIX platforms.

Only include a matching file if it is a named pipe (fifo) device.

TEST_READ

Only include a matching file if it is readable by the user.

Note

This keyword does not support Access Control List (ACL) settings for files.

TEST_REGULAR

Only include a matching file if it is a regular disk file and not a directory, pipe, socket, or other special file type.

TEST_SETGID

This keyword is only available on UNIX platforms.

Only include a matching file if it has its Set-Group-ID bit set.

TEST_SETUID

This keyword is only available on UNIX platforms.

Only include a matching file if it has its Set-User-ID bit set.

TEST_SOCKET

This keyword is only available on UNIX platforms.

Only include a matching file if it is a UNIX domain socket.

TEST_STICKY_BIT

This keyword is only available on UNIX platforms.

Only include a matching file if it has its sticky bit set.

TEST_SYMLINK

This keyword is only available on UNIX platforms.

Only include a matching file if it is a symbolic link that points at an existing file.

TEST_USER

This keyword is only available on UNIX platforms.

Only include a matching file if it belongs to the same effective user ID (UID) as the IDL process.

TEST_WRITE

Only include a matching file if it is writable by the user.

Note

This keyword does not support Access Control List (ACL) settings for files.

TEST_ZERO_LENGTH

Only include a matching file if it has zero length.

Note

The length of a directory is highly system-dependent and does not necessarily correspond to the number of files it contains. In particular, it is possible for an empty directory to report a non-zero length. RSI does not recommend using the `TEST_ZERO_LENGTH` keyword on directories, as the information returned cannot be used in a meaningful way.

TEST_* Keywords

The keywords with names that start with the `TEST_` prefix allow you to filter the list of resulting file paths based on various criteria. If you remove the `TEST_` prefix from these keywords, they correspond directly to the same keywords to the `FILE_TEST` function, and are internally implemented by the same test code. One could therefore use `FILE_TEST` instead of the `TEST_` keywords to `FILE_SEARCH`. For example, the following statement locates all subdirectories of the current directory:

```
Result = FILE_SEARCH(/TEST_DIRECTORY)
```

It is equivalent to the following statements, using `FILE_TEST`:

```
result = FILE_SEARCH()
idx = where(FILE_TEST(result, /DIRECTORY), count)
result = (count eq 0) ? '' : result[idx]
```

The `TEST_*` keywords are more succinct, and can be more efficient in the common case in which `FILE_SEARCH` generates a long list of results, only to have `FILE_TEST` discard most of them.

WINDOWS_SHORT_NAMES

By default, `FILE_SEARCH` ignores Microsoft Windows 8.3 short names when performing file matching and only considers the real file names. Set the `WINDOWS_SHORT_NAMES` keyword to change this policy. If this keyword is set, `FILE_SEARCH` looks at both the real and 8.3 short names associated with each file as it checks for a match. See “Microsoft Windows “8.3 Short Names”” on page 712 for more information on this subject. This keyword is quietly ignored on non-Windows platforms.

You should be aware that turning on short name support can lead to confusing results. Such use should be considered carefully. For example, consider running the following IDL statement in a directory containing a single file named `file_search.html`:

```
PRINT, FILE_SEARCH( '*.htm' )
```

Because this statement does not enable short name support, no files matching the specified pattern are found, and IDL does not print any filenames. This is the result most people would expect. If short name support is enabled, however:

```
PRINT, FILE_SEARCH( '*.htm', /WINDOWS_SHORT_NAMES )
```

IDL prints:

```
file_search.html
```

In this case, IDL checks the short names as well as the real names. The 8.3 short name for `file_search.html` will be similar to `FILE_S~1.HTM`, which matches the *Path Specification*. As such, `FILE_SEARCH` reports the real name for the matched file. This is the correct answer, but probably not the expected result.

Supported Wildcards and Expansions

The wildcards understood by `FILE_SEARCH` are based on those used by the standard UNIX shell `/bin/sh` (the `?`, `*`, `[`, and `]` characters, and environment variables) with some enhancements commonly found in the C-shell `/bin/csh` (the `~`, `{`, and `}` characters). These wildcards are processed identically across all IDL supported platforms. The supported wildcards are shown in the following table:

Wildcard	Description
*	Matches any string, including the null string.
?	Matches any single character.

Table 3-39: Supported Wildcards and Expansions

Wildcard	Description
[...]	Matches any one of the enclosed characters. A pair of characters separated by “-” matches any character lexically between the pair, inclusive. If the first character following the opening bracket ([) is a ! or ^, any character not enclosed is matched.
{ <i>str</i> , <i>str</i> , ...}	Expand to each string (or filename-matching pattern) in the comma-separated list.
~ ~ <i>user</i>	If used at start of input file specification, is replaced with the path to the appropriate home directory. See the description of the EXPAND_TILDE keyword for details.
<i>\$var</i>	Replace with value of the named environment variable. See the description of the EXPAND_ENVIRONMENT keyword for full details.
<i>\${var}</i>	Replace <i>\${var}</i> with the value of the <i>var</i> environment variable. If <i>var</i> is not found in the environment, <i>\${var}</i> is replaced with a null string. This format is useful when the environment variable reference sits directly next to unrelated text, as the use of the { } brackets make it possible for IDL to determine where the environment variable ends and the remaining text starts (e.g. <i>\${mydir}other_text</i>).
<i>\${var:-alttext}</i>	If environment variable <i>var</i> is present in the environment and has a non-NULL value, then substitute that value. If <i>var</i> is not present, or has a NULL value, then substitute the alternative text (<i>alttext</i>) provided instead.
<i>\${var-alttext}</i>	If environment variable <i>var</i> is present in the environment (even if it has a NULL value) then substitute that value. If <i>var</i> is not present, then substitute the alternative text (<i>alttext</i>) provided instead.

Table 3-39: Supported Wildcards and Expansions (Continued)

These wildcards can appear anywhere in an input file specification, with the following exceptions:

Tilde (~)

The tilde character is only considered to be a wildcard if it is the first character in the input file specification and the `EXPAND_TILDE` keyword is set. Otherwise, it is treated as a regular character.

Initial Dot Character

The default is for wildcards not to match the dot (.) character if it occurs as the first character of a directory or file name. This follows the convention of UNIX shells, which treat such names as hidden files. In order to match such files, you can take any of the following actions:

- Explicitly include the dot character at the start of your pattern (e.g. “.*”).
- Specify the `MATCH_INITIAL_DOT` keyword, which changes the dot matching policy so that wildcards will match any names starting with dot (except for the special “.” and “..” directories).
- Specify the `MATCH_ALL_INITIAL_DOT` keyword, which changes the dot matching policy so that wildcards will match any names starting with dot (including the special “.” and “..” directories).

Microsoft Windows UNC Paths

On a local area network, Microsoft Windows offers an alternative to the drive letter syntax for accessing files. The *Universal Naming Convention* (UNC) allows for specification of paths on other hosts using the syntax:

```
\\hostname\sharename\dir\dir\...\file
```

UNC paths are distinguished from normal paths by the use of two initial slashes in the path. `FILE_SEARCH` can process such paths, but wildcard characters are not allowed in the `hostname` or `sharename` segments. Wildcards are allowed for specifying directories and files. For performance reasons, RSI does not recommend using the recursive form of `FILE_SEARCH` with UNC paths on very large directory trees.

Platform-Specific Filename Matching Issues

When using `FILE_SEARCH`, you should be aware of the following platform-specific issues:

File Path Syntax

The syntax allowed for file paths differs between operating systems. `FILE_SEARCH` always processes file paths using the syntax rules for the platform on which the IDL

session is running. As a convenience for Microsoft Windows users, Windows IDL accepts UNIX style forward slashes as well as the usual backslashes as path separators.

Differing Defaults Between Platforms

The different operating systems supported by IDL have some conventions for processing file paths that are inherently incompatible. If `FILE_SEARCH` attempted to force an identical default policy for these features across all platforms, the resulting routine would be inconvenient to use on all platforms. `FILE_SEARCH` resolves this inherent tension between convenience and control in the following way:

- These features are controlled by keywords which are listed in the table below. If a keyword is not explicitly specified, `FILE_SEARCH` will determine an appropriate default for that feature based on the conventions of the underlying operating system. Hence, `FILE_SEARCH` will by default behave in a way that is reasonable on the platform on which it is used.
- If one of these keywords is explicitly specified, `FILE_SEARCH` will use its value to determine support for that feature. Hence, if the keyword is used, `FILE_SEARCH` will behave identically on all platforms. If maximum cross-platform control is desired, you can achieve it by specifying all the relevant keywords.

The keywords that have different defaults on different platforms are listed in the following table:

Wildcard	Keyword	Default UNIX	Default Win
<code>\$var</code> <code>\${var}</code> <code>\${var:-alttext}</code> <code>\${var-alttext}</code>	<code>EXPAND_ENVIRONMENT</code>	yes	no
<code>~</code>	<code>EXPAND_TILDE</code>	yes	no
	<code>FOLD_CASE</code>	no	yes

Table 3-40: `FILE_SEARCH` Defaults that Differ Between Platforms

Microsoft Windows “8.3 Short Names”

Older versions of Microsoft operating systems limited files to very short names: up to eight characters were allowed for the file name, followed by a dot (.), followed by an

extension of up to three characters. This scheme is often referred to as “8.3 short names” or just “8.3”. Newer releases of the Windows operating system have moved past the 8.3 limits, and allow much longer names. In order to allow old programs from older systems to run on newer systems without first being rebuilt, these newer versions of Windows actually maintain two separate and distinct file names for each file. Every file has, in addition to its real (potentially long) name, an automatically generated “8.3 short name”. If the real name fits within the 8.3 limits, the real and short names are the same. If the real name does not fit within the 8.3 limits, the operating system constructs an 8.3 short name for it by applying a set of heuristic rules (See Microsoft’s documentation for more detail on how these names are constructed). For example, the file `file_search.html` will be given a short name that looks something like `FILE_S~1.HTM`.

Note

8.3 short names are strictly a Windows backwards compatibility feature, and are not generally useful in newer software.

8.3 short names are an issue for `FILE_SEARCH` if the *Path_Specification* argument includes them. `FILE_SEARCH` handles this situation using the following rules:

1. Before starting the process of file name matching, `FILE_SEARCH` examines the portion of *Path_Specification* between the first character and the first wildcard character (or the entire string if there are no wildcards) for non-wildcarded short names. Any such names are replaced by their real names.
2. By default, `FILE_SEARCH` only considers the real name during the process of matching *Path_Specification* with files, ignoring the 8.3 short names. The `WINDOWS_SHORT_NAMES` keyword can be set to change this policy. If `WINDOWS_SHORT_NAMES` is set, `FILE_SEARCH` looks at *both* the real and 8.3 short names associated with each file as it checks for a match.
3. If `WINDOWS_SHORT_NAMES` is set and `FILE_SEARCH` matches the 8.3 short name for a file, the real file name is returned. For instance, in the above example if `FILE_SEARCH` matches `FILE_S~1.HTM`, it will return `file_search.html`.

Warning

Windows 8.3 short names can be very confusing to understand. RSI recommends not using them unless absolutely necessary. Most modern applications will not encounter a need to match 8.3 short names.

Examples

Example 1

Find all files in the current working directory:

```
Result = FILE_SEARCH()
```

Example 2

Find all IDL program (*.pro) files in the current working directory:

```
Result = FILE_SEARCH('*.pro')
```

Example 3

Under Microsoft Windows, find all files in the top level directories of all drives other than the floppy drives:

```
Result=FILE_SEARCH('[!ab]:*')
```

This example relies on the following:

- FILE_SEARCH allows wildcards within the drive letter part of an input file specification.
- Drives A and B are always floppies, and are not used by Windows for any other type of drive.

Example 4

Find all files in the user's home directory that start with the letters A-D. Match both upper and lowercase letters:

```
Result = FILE_SEARCH('~/[a-d]*', /EXPAND_TILDE, /FOLD_CASE)
```

Example 5

Find all directories in the user's home directory that start with the letters A-D. Match both upper and lowercase letters:

```
Result = FILE_SEARCH('~/[a-d]*', /EXPAND_TILDE, /FOLD_CASE, $
    /TEST_DIRECTORY)
```

Example 6

Recursively find all subdirectories found underneath the user's home directory that do not start with a dot character:

```
Result = FILE_SEARCH('$HOME', '*', /EXPAND_ENVIRONMENT, $
```

```
/TEST_DIRECTORY)
```

Example 7

Recursively find all subdirectories found underneath the user's home directory, including those that start with a dot character, but excluding the special “.” and “..” directories:

```
Result = FILE_SEARCH('$HOME', '*', /MATCH_INITIAL_DOT, $
    /EXPAND_ENVIRONMENT, /TEST_DIRECTORY)
```

Example 8

Find all .pro and .sav files in an IDL library search path, sorted by directory, in the order IDL searches for them:

```
Result = FILE_SEARCH(STRSPLIT(!PATH, PATH_SEP(/SEARCH_PATH), $
    /EXTRACT) + '/*.{pro,sav}')
```

Colon (:) is the UNIX path separator character, so the call to STRSPLIT breaks the IDL search path into an array of directories. To each directory name, we concatenate the wildcards necessary to match any .pro or .sav files in that directory. When this array is passed to FILE_SEARCH, it locates all files that match these specifications. FILE_SEARCH sorts all of the files found by each input string. The files for each string are then placed into the output array in the order they were searched for.

Example 9

Recursively find all directories in your IDL distribution:

```
Result = FILE_SEARCH(!DIR, '*', /TEST_DIRECTORY)
```

Example 10

Under Microsoft Windows, FILE_SEARCH can be used to convert a path that uses 8.3 short names into an equivalent path that uses the real names. For example, Windows provides a directory for each user where applications are expected to create temporary files. Consider a user of Windows 2000 named Scott. The short name for Scott's temporary directory (which is usually available via the Windows “Temp” environment variable) will typically be something like C:\DOCUME~1\scott\LOCALS~1\Temp. This can be converted to real names using a statement like the following:

```
PRINT, FILE_SEARCH(GETENV('Temp'))
```

IDL prints:

```
C:\Documents and Settings\scott\Local Settings\Temp
```

There are some noteworthy facts about this example:

1. Short names are relatively rare in modern Windows systems. Environment variables are one of the few ways in which they are still seen.
2. It was not necessary to specify the `WINDOWS_SHORT_NAMES` keyword in this example, although it would have been harmless to do so, because the path being converted contains no wildcard characters. `FILE_SEARCH` will always attempt to convert any non-wildcarded path components at the beginning of the path to their long names before it begins searching.
3. The `GETENV` function supports a special variable (`IDL_TMPDIR`) that should be used to obtain the directory where temporary files go. This provides a portable cross platform way to find a good temporary directory. Under Windows, `GETENV` automatically converts the result of translating the `IDL_TMPDIR` environment variable to real names in a manner similar to that shown in this example.
4. People find the 8.3 short names confusing, and you may wish to translate them to real names for display purposes. However, most file handling functions and procedures in IDL will accept either without issue. There is little reason to translate them before using them in programs.

Version History

Introduced: 5.5

`WINDOWS_SHORT_NAMES` keyword added: 6.1

See Also

[FILE_TEST](#), [FILEPATH](#), [GETENV](#)

FILE_TEST

The FILE_TEST function checks files for existence and other attributes without having to first open the file.

Syntax

```
Result = FILE_TEST( File [, /DIRECTORY | , /EXECUTABLE | , /READ |  
    , /REGULAR | , /WRITE | , /ZERO_LENGTH] [, GET_MODE=variable]  
    [, /NOEXPAND_PATH] )
```

UNIX-Only Keywords: [, /BLOCK_SPECIAL | , /CHARACTER_SPECIAL |
 , /DANGLING_SYMLINK | , /GROUP | , /NAMED_PIPE | , /SETGID | , /SETUID |
 , /SOCKET | , /STICKY_BIT | , /SYMLINK | , /USER]

Return Value

FILE_TEST returns 1 (true), if the specified file exists and all of the attributes specified by the keywords are also true. If no keywords are present, a simple test for existence is performed. If the file does not exist or one of the specified attributes is not true, then FILE_TEST returns 0 (false).

Arguments

File

A scalar or array of file names to be tested. The result is of type integer with the same number of elements as *File*.

Keywords

BLOCK_SPECIAL

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and is a block special device.

CHARACTER_SPECIAL

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and is a character special device.

DANGLING_SYMLINK

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* is a symbolic link that points at a non-existent file.

DIRECTORY

Set this keyword to return 1 (true) if *File* exists and is a directory.

EXECUTABLE

Set this keyword to return 1 (true) if *File* exists and is executable. The source of this information differs between operating systems:

- UNIX: IDL checks the per-file information (the execute bit) maintained by the operating system.
- Microsoft Windows: The determination is made on the basis of the file name extension (e.g. `.exe`).

GET_MODE

Set this keyword to a named variable to receive the UNIX style mode (permission) mask for the specified file. The bits in these masks correspond to those used by the UNIX `chmod(2)` system call, and are explained in detail in the description of the *Mode* argument to the [FILE_CHMOD](#) procedure. When interpreting the value returned by this keyword, the following platform specific details should be kept in mind:

- The `setuid`, `setgid`, and sticky bits are specific to the UNIX operating system, and will never be returned on any other platform. Consult the `chmod(2)` man page and/or other UNIX programming documentation for more details.
- The Microsoft Windows operating system does not have 3 permission classes like UNIX does. Therefore, IDL returns the same settings for all three classes.
- The Microsoft Windows operating system does not maintain an execute bit for files, but instead uses the file suffix to decide if a file is executable.

GROUP

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and belongs to the same effective group ID (GID) as the IDL process.

NAMED_PIPE

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and is a named pipe (fifo) device.

NOEXPAND_PATH

Set this keyword to cause FILE_TEST to use the *File* argument exactly as specified, without applying the usual file path expansion.

READ

Set this keyword to return 1 (true) if *File* exists and is readable by the user.

REGULAR

Set this keyword to return 1 (true) if *File* exists and is a regular disk file and not a directory, pipe, socket, or other special file type.

SETGID

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and has its Set-Group-ID bit set.

SETUID

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and has its Set-User-ID bit set.

SOCKET

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and is a UNIX domain socket.

STICKY_BIT

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and has its sticky bit set.

SYMLINK

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and is a symbolic link that points at an existing file.

USER

This keyword is only available on UNIX platforms.

Set this keyword to return 1 (true) if *File* exists and belongs to the same effective user ID (UID) as the IDL process.

WRITE

Set this keyword to return 1 (true) if *File* exists and is writable by the user.

ZERO_LENGTH

Set this keyword to return 1 (true) if *File* exists and has zero length.

Note

The length of a directory is highly system dependent and does not necessarily correspond to the number of files it contains. In particular, it is possible for an empty directory to report a non-zero length. RSI does not recommend using the ZERO_LENGTH keyword on directories, as the information returned cannot be used in a meaningful way.

Examples

Does my IDL distribution support the IRIX operating system?

```
result = FILE_TEST(!DIR + '/bin/bin.sgi', /DIRECTORY)
PRINT, 'IRIX IDL Installed: ', result ? 'yes' : 'no'
```

Version History

Introduced: 5.4

See Also

[FILE_INFO](#), [FILE_SEARCH](#), [FSTAT](#)

FILE_WHICH

The FILE_WHICH function separates a specified file path into its component directories, and searches each directory in turn for a specific file. This command is modeled after the UNIX `which(1)` command.

This routine is written in the IDL language. Its source code can be found in the file `file_which.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = FILE_WHICH( [Path, ] File [, /INCLUDE_CURRENT_DIR] )
```

Return Value

Returns the path for the first file for the given name found by searching the specified path. If FILE_WHICH does not find the desired file, a NULL string is returned.

Arguments

Path

A search path to be searched. If *Path* is not present, the value of the IDL `!PATH` system variable is used.

File

The file to look for in the directories given by *Path*.

Keywords

INCLUDE_CURRENT_DIR

If set, FILE_WHICH looks in the current directory before starting to search *Path* for *File*. When IDL searches for a routine to compile, it looks in the current working directory before searching `!PATH`. The `INCLUDE_CURRENT_DIR` keyword allows FILE_WHICH to mimic this behavior.

Examples

To find the location of this routine:

```
Result = FILE_WHICH('file_which.pro')
```

To find the location of the UNIX `ls` command:

```
Result = FILE_WHICH(getenv('PATH'), 'ls')
```

Version History

Introduced: 5.4

FILEPATH

The FILEPATH function returns the fully-qualified path to a file contained in the IDL distribution. Operating system dependencies are taken into consideration. This routine is used by RSI to make the library routines portable. This routine is written in the IDL language. Its source code can be found in the file `filepath.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = FILEPATH( Filename [, ROOT_DIR=string]
                  [, SUBDIRECTORY=string/string_array] [, /TERMINAL] [, /TMP] )
```

Return Value

Returns the fully-qualified path to a specified file.

Arguments

Filename

A string containing the name of the file to be found. The file should be specified in all lowercase characters. No device or directory information should be included.

Keywords

ROOT_DIR

A string containing the name of the directory from which the resulting path should be based. If not present, the value of `!DIR` is used. This keyword is ignored if `TERMINAL` or `TMP` are specified.

SUBDIRECTORY

The name of the subdirectory in which the file should be found. If this keyword is omitted, the main IDL directory is used. This variable can be either a scalar string or a string array with the name of each level of subdirectory depth represented as an element of the array.

For example, to get a path to the file `filepath.pro` in IDL's `lib` subdirectory, enter:

```
path = FILEPATH('filepath.pro',SUBDIR=['lib'])
```

TERMINAL

Set this keyword to return the filename of the user's terminal.

TMP

Set this keyword to indicate that the specified file is a scratch file. Returns a path to the proper place for temporary files under the current operating system.

Under Microsoft Windows, FILEPATH checks to see if the following environment variables are set—TMP, TEMP, WINDIR—and uses the value of the first one it finds. If none of these environment variables exists, \TMP is used as the temporary directory.

Examples

Open the IDL distribution file `people.dat`:

```
OPENR, 1, FILEPATH('people.dat', SUBDIRECTORY=['examples','data'])
```

Version History

Introduced: Pre 4.0

See Also

[FILE_SEARCH](#), [PATH_SEP](#)

FINDGEN

The FINDGEN function creates a floating-point array of the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = \text{FINDGEN}(D_1 [, \dots, D_8])$$

Return Value

Returns a single-precision, floating-point array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create F, a 6-element vector of single-precision, floating-point values where each element is set to the value of its subscript, enter:

```
F = FINDGEN(6)
```

The value of F[0] is 0.0, F[1] is 1.0, and so on.

Version History

Introduced: Original

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [INDGEN](#), [LINDGEN](#),
[SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

FINITE

The FINITE function identifies whether or not a given argument is finite.

Syntax

```
Result = FINITE( X [, /INFINITY] [, /NAN] [, SIGN=value])
```

Return Value

Returns 1 (True) if its argument is finite. If the argument is infinite or not a defined number (*NaN*), the FINITE function returns 0 (False). The result is a byte expression of the same structure as the argument *X*.

Note

See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.

Arguments

X

A floating-point, double-precision, or complex scalar or array expression. Strings are first converted to floating-point. This function is meaningless for byte, integer, or longword arguments.

Keywords

INFINITY

Set this keyword to cause FINITE to return True if the *X* argument is the IEEE special floating-point value *Infinity* (either positive or negative), or False otherwise.

NAN

Set this keyword to cause FINITE to return True if the *X* argument is the IEEE special floating-point value “Not A Number” (*NaN*), or False otherwise.

SIGN

If the INFINITY or NAN keyword is set, then set this keyword to one of the following values:

Value	Description
> 0	If the INFINITY keyword is set, return True (1) if X is positive infinity, False (0) otherwise. If the NAN keyword is set, return True (1) if X is +NaN (negative sign bit is not set), False (0) otherwise.
0 (the default)	The sign of X (positive or negative) is ignored.
< 0	If the INFINITY is set, return True (1) if X is negative infinity, False (0) otherwise. If the NAN keyword is set, return True (1) if X is -NaN (negative sign bit is set), False (0) otherwise.

Table 3-41: SIGN Keyword Values

If neither the INFINITY nor NAN keyword is set, then this keyword is ignored.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Example 1

To find out if the logarithm of 5.0 is finite, enter:

```
PRINT, FINITE(ALOG(5.0))
```

IDL prints "1" because the argument is finite.

Example 2

To determine which elements of an array are infinity or NaN (Not a Number) values:

```
A = FLTARR(10)

; Set some values to +/-NaN and positive or negative Infinity:
A[3] = !VALUES.F_NAN
A[4] = -!VALUES.F_NAN
A[6] = !VALUES.F_INFINITY
A[7] = -!VALUES.F_INFINITY
```

Find the location of values in A that are positive or negative

```
Infinity:
PRINT, WHERE(FINITE(A, /INFINITY))
```

IDL prints:

```
          6          7
```

Find the location of values in A that are NaN:

```
PRINT, WHERE(FINITE(A, /NAN))
```

IDL prints:

```
          3          4
```

Find the location of values in A that are negative Infinity:

```
PRINT, WHERE(FINITE(A, /INFINITY, SIGN=-1))
```

IDL prints:

```
          7
```

Find the location of values in A that are +NaN:

```
PRINT, WHERE(FINITE(A, /NAN, SIGN=1))
```

IDL prints:

```
          3
```

Note

On some platforms, there is no distinction between +NaN and -NaN.

Version History

Introduced: Original

See Also

[CHECK_MATH](#), [MACHAR](#), [!VALUES](#), “[Special Floating-Point Values](#)” on page 468.

FIX

The FIX function converts a given expression to an integer type. Optionally, the conversion type can be specified at runtime, allowing flexible runtime type-conversion to arbitrary types.

Syntax

```
Result = FIX( Expression [, Offset [, D1 [, ..., D8]]] [, /PRINT]
            [, TYPE=type code{0 to 15} ] )
```

Return Value

Returns a result equal to *Expression* converted to integer type.

Arguments

Expression

The expression to be converted.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as integer data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

The *Offset* and *Dim_i* arguments are not allowed when converting to or from the string type.

D_i

When extracting fields of data, the *D_i* arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The *D_i* arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

The *Offset* and *D_i* arguments are not allowed when converting to or from the string type.

Keywords

PRINT

Set this keyword to specify that any special-case processing when converting between string and byte data, or the reverse, should be suppressed. The PRINT keyword is ignored unless the TYPE keyword is used to convert to these types.

TYPE

FIX normally converts *Expression* to the integer type. If TYPE is specified, it is the type code to set the type of the conversion. This feature allows dynamic type conversion, where the desired type is not known until runtime, to be carried out without the use of large CASE or IF...THEN logic. When TYPE is specified, FIX behaves as if the appropriate type conversion routine for the desired type had been called. See the “See Also” list below for the complete list of such routines.

When using the TYPE keyword to convert BYTE data to STRING or the reverse, you should be aware of the special-case processing that the BYTE and STRING functions do in this case. To prevent this, and get a simple type conversion in these cases, you must specify the PRINT keyword.

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Convert the floating-point array [2.2, 3.0, 4.5] to integer type and store the new array in the variable I by entering:

```
I = FIX([2.2, 3.0, 4.5])
```

Version History

Introduced: Original

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FLOAT](#), [LONG](#), [LONG64](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

FLICK

The `FLICK` procedure causes the display to flicker between two output images at a given rate.

This routine is written in the IDL language. Its source code can be found in the file `flick.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
FLICK, A, B [, Rate]
```

Arguments

A

Byte image number 1, scaled from 0 to 255.

B

Byte image number 2, scaled from 0 to 255.

Rate

The flicker rate. The default is 1.0 sec/frame

Keywords

None.

Version History

Introduced: Original

See Also

[CW_ANIMATE](#), [XINTERANIMATE](#)

FLOAT

The FLOAT function converts a given *Expression* into a single-precision floating-point value.

Syntax

$$\text{Result} = \text{FLOAT}(\text{Expression} [, \text{Offset} [, D_1 [, \dots, D_8]]])$$

Return Value

Returns the conversion of the given expression into single-precision floating point values. If *Expression* is a complex number, FLOAT returns the real part.

Arguments

Expression

The expression to be converted to single-precision floating-point.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as single-precision floating point data.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid floating-point value and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

If *A* contains the integer value 6, it can be converted to floating-point and stored in the variable *B* by entering:

```
B = FLOAT(A)
```

Version History

Introduced: Original

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [LONG](#), [LONG64](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

FLOOR

The FLOOR function returns the closest integer less than or equal to its argument.

Syntax

Result = FLOOR(*X* [, /L64])

Return Value

If the input argument *X* is an integer type, *Result* has the same value and type as *X*. Otherwise, *Result* is a 32-bit longword integer with the same structure as *X*.

Arguments

X

The value for which the FLOOR function is to be evaluated. This value can be any numeric type (integer, floating, or complex).

Keywords

L64

If set, the result type is 64-bit integer regardless of the input type. This is useful for situations in which a floating point number contains a value too large to be represented in a 32-bit integer.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To print the floor function of 5.9, enter:

```
PRINT, FLOOR(5.9)
; IDL prints:
5
```

To print the floor function of 3000000000.1, the result of which is too large to represent in a 32-bit integer:

```
PRINT, FLOOR(3000000000.1D, /L64)
; IDL prints:
3000000000
```

Version History

Introduced: Pre 4.0

See Also

[CEIL](#), [COMPLEXROUND](#), [ROUND](#)

FLOW3

The FLOW3 procedure draws lines representing a 3D flow/velocity field. Note that the 3D scaling system must be in place before calling FLOW3. This procedure works best with Z buffer output device.

This routine is written in the IDL language. Its source code can be found in the file `flow3.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
FLOW3, Vx, Vy, Vz [, ARROWSIZE=value] [, /BLOB] [, LEN=value]  
[, NSTEPS=value] [, NVECS=value] [, SX, SY, SZ=vector]
```

Arguments

Vx, Vy, Vz

3D arrays containing X, Y, and Z components of the field.

Keywords

ARROWSIZE

Size of arrowheads (default = 0.05).

BLOB

Set this keyword to draw a blob at the beginning of each flow line and suppress the arrows.

LEN

Length of each step used to follow flow lines (default = 2.0). Expressed in units of largest field vector (i.e., the length of the longest step is set to len times the grid spacing).

NSTEPS

Number of steps used to follow the flow lines (default = largest dimension of $Vx / 5$).

NVECS

Number of random flow lines to draw (default = 200). Only used if S_x , S_y , S_z are not present.

SX, SY, SZ

Optional vectors containing the starting coordinates of the flow lines. If omitted random starting points are chosen.

Examples

```
; Create a set of random three-dimensional arrays to represent
; the field:
vx = RANDOMU(seed, 5, 5, 5)
vy = RANDOMU(seed, 5, 5, 5)
vz = RANDOMU(seed, 5, 5, 5)

; Set up the 3D scaling system:
SCALE3, xr=[0,4], yr=[0,4], zr = [0,4]

; Plot the vector field:
FLOW3, vx, vy, vz
```

Version History

Introduced: Pre 4.0

See Also

[VEL](#), [VELOVECT](#)

FLTARR

The FLTARR function creates a floating-point vector or array of the specified dimensions.

Syntax

$$\text{Result} = \text{FLTARR}(D_1 [, \dots, D_8] [, \text{NOZERO}])$$

Return Value

Returns a single-precision, floating-point vector or array.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, FLTARR sets every element of the result to zero. Set this keyword to inhibit zeroing of the array elements and cause FLTARR to execute faster.

Examples

Create F, a 3-element by 3-element floating-point array with each element set to 0.0 by entering:

```
F = FLTARR(3, 3)
```

Version History

Introduced: Original

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [INTARR](#), [LON64ARR](#),
[LONARR](#), [MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

FLUSH

The FLUSH procedure causes all buffered output on the specified file units to be written. IDL uses buffered output for reasons of efficiency. This buffering leads to rare occasions where a program needs to be certain that output data are not waiting in a buffer, but have actually been output.

Syntax

FLUSH, *Unit*₁, ..., *Unit*_{*n*}

Arguments

Unit_{*j*}

The file units (logical unit numbers) to flush.

Version History

Introduced: Original

See Also

[CLOSE](#), [EMPTY](#), [EXIT](#)

FOR

The FOR statement executes one or more statements repeatedly, incrementing or decrementing a variable with each repetition, until a condition is met.

Note

For more information on using FOR and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

```
FOR variable = init, limit [, Increment] DO statement
```

or

```
FOR variable = init, limit [, Increment] DO BEGIN
```

```
    statements
```

```
ENDFOR
```

Examples

The following example iterates over the elements of an array, printing the value of each element:

```
array = ['one', 'two', 'three']
n = N_ELEMENTS(array)
FOR i=0,n-1 DO BEGIN
    PRINT, array[i]
ENDFOR
```

Version History

Introduced: Original

FORMAT_AXIS_VALUES

The `FORMAT_AXIS_VALUES` function converts a vector of numeric values into a vector of string values. This routine uses the same rules for formatting as do the axis routines that label tick marks given a set of tick values.

Syntax

Result = `FORMAT_AXIS_VALUES(Values)`

Return Value

Returns a vector of formatted string values from an input vector of numeric values.

Arguments

Values

Set this argument to a vector of numeric values to be formatted.

Keywords

None.

Examples

Suppose we have a vector of axis values:

```
axis_values = [7.9, 12.1, 15.3, 19.0]
```

Convert these values into an array of strings:

```
new_values = FORMAT_AXIS_VALUES(axis_values)
HELP, new_values
PRINT, new_values
PRINT, axis_values
```

IDL prints:

```
NEW_VALUES      STRING      = Array[4]
7.9 12.1 15.3 19.0
7.90000      12.1000      15.3000      19.0000
```

Version History

Introduced: 5.1

FORWARD_FUNCTION

The FORWARD_FUNCTION statement causes argument(s) to be interpreted as functions rather than variables (versions of IDL prior to 5.0 used parentheses to declare arrays).

Note

For information on using the FORWARD_FUNCTION statement, see [Chapter 4, “Procedures and Functions”](#) in the *Building IDL Applications* manual.

Syntax

FORWARD_FUNCTION *Name*₁, *Name*₂, ..., *Name*_{*n*}

Version History

Introduced: Pre 4.0

FREE_LUN

The `FREE_LUN` procedure deallocates previously-allocated file units. This routine is usually used with file units allocated with `GET_LUN`, but it will also close any other specified file unit. If the specified file units are open, they are closed prior to the deallocation.

Syntax

```
FREE_LUN [, Unit1, ..., Unitn] [, EXIT_STATUS=variable] [, /FORCE]
```

Arguments

*Unit*_{*j*}

The IDL file units (logical unit numbers) to deallocate.

Keywords

EXIT_STATUS

Set this keyword to a named variable that will contain the exit status reported by a UNIX child process started via the `UNIT` keyword to `SPAWN`. This value is the exit value reported by the process by calling `EXIT`, and is analogous to the value returned by `$?` under most UNIX shells.

FORCE

Set this keyword to override the IDL file output buffer and force the file to be closed no matter what errors occur in the process.

IDL buffers file output for performance reasons. If it is not possible to properly flush this data when a file close is requested, an error is normally issued and the file remains open. An example of this might be that your disk does not have room to write the remaining data. This default behavior prevents data from being lost. To override it and force the file to be closed no matter what errors occur in the process, specify `FORCE`.

Examples

See the example for the `GET_LUN` procedure.

Version History

Introduced: Original

See Also

[CLOSE](#), [GET_LUN](#)

FSTAT

The FSTAT function returns status information about a specified file unit.

Syntax

Result = FSTAT(*Unit*)

Return Value

The FSTAT function returns a structure expression of type FSTAT (or FSTAT64 in the case of files that are longer than $2^{31}-1$ bytes in length) containing status information about a specified file unit. The contents of this structure are documented in “[The FSTAT Function](#)” in Chapter 11 of the *Building IDL Applications* manual.

Fields of the FSTAT Structure

The following descriptions are of *fields* in the structure returned by the FSTAT function. They are *not* keywords to FSTAT.

- **UNIT** — The IDL logical unit number (LUN).
- **NAME** — The name of the file.
- **OPEN** — Nonzero if the file unit is open. If OPEN is zero, the remaining fields in FSTAT will not contain useful information.
- **ISATTY** — Nonzero if the file is actually a terminal instead of a normal file. For example, if you are using an `xterm` window on a UNIX system and you invoke FSTAT on logical unit 0 (standard input), ISATTY will be set to 1.
- **ISAGUI** — Nonzero if the file is actually a Graphical User Interface (for example, a logical unit associated with the IDL Development Environment). Thus, if you are using the IDLDE and you invoke FSTAT on logical unit 0 (standard input), ISAGUI will be set to 1.
- **INTERACTIVE** — Nonzero if *either* ISATTY or ISAGUI is nonzero.
- **XDR** — Nonzero if the file was opened with the XDR keyword, and is therefore considered to contain data in the XDR format.
- **COMPRESS** — Nonzero if the file was opened with the COMPRESS keyword, and is therefore considered to contain compressed data in the GZIP format.
- **READ** — Nonzero if the file is open for read access.

- **WRITE** — Nonzero if the file is open for write access.
- **ATIME, CTIME, MTIME** — The date of last access, date of creation, and date of last modification given in seconds since 1 January 1970 UTC. Use the **SYSTIME** function to convert these dates into a textual representation.

Note

Some file systems do not maintain all of these dates (e.g. MS DOS FAT file systems), and may return 0. On some non-UNIX operating systems, access time is not maintained, and **ATIME** and **MTIME** will always return the same date.

- **TRANSFER_COUNT** — The number of scalar IDL data items transferred in the last input/output operation on the unit. This is set by the following IDL routines: **READU**, **WRITEU**, **PRINT**, **PRINTF**, **READ**, and **READF**. **TRANSFER_COUNT** is useful when attempting to recover from input/output errors.
- **CUR_PTR** — The current position of the file pointer, given in bytes from the start of the file. If the device is a terminal (**ISATTY** is nonzero), the value of **CUR_PTR** will not contain useful information. When reporting on file units opened with the **COMPRESS** keyword to **OPEN**, the position reported by **CUR_PTR** is the “logical” position—the position it would be at in the uncompressed version of the same file.
- **SIZE** — The current length of the file in bytes. If the device is a terminal (**ISATTY** is nonzero), the value of **SIZE** will not contain useful information. When reporting on file units opened with the **COMPRESS** keyword to **OPEN**, the size reported by **SIZE** is the compressed size of the actual file, and not the logical length of the uncompressed data contained within. This is inconsistent with the position reported by **CUR_PTR**. The reason for reporting the size in this way is that the logical length of the data cannot be known without reading the entire file from beginning to end and counting the uncompressed bytes, and this would be extremely inefficient.
- **REC_LEN** — This field is obsolete and will always contain zero.

Arguments

Unit

The file unit about which information is required. This parameter can be an integer or an associated variable, in which case information about the variable’s associated file is returned.

Keywords

None.

Examples

If file unit number 1 is open, the FSTAT information on that unit can be seen by entering:

```
PRINT, FSTAT(1)
```

Specific information can be obtained by referring to single fields within the structure returned by FSTAT. The following code prints the name and length of the file open on unit 1:

```
; Put FSTAT information in variable A:  
A = FSTAT(1)  
  
; Print the name and size fields:  
PRINT, 'File: ', A.NAME, ' is ', A.SIZE, ' bytes long.'
```

Version History

Introduced: Original

See Also

[ASSOC](#), [FILE_INFO](#), [FILE_TEST](#), [OPEN](#)

FULSTR

The `FULSTR` restores a row-indexed sparse array to full storage mode. If the sparse array was created with the `SPRSIN` function using the `THRESH` keyword, any values in the original array that were below the specified threshold are replaced with zeros.

Syntax

Result = `FULSTR(A)`

Return Value

Returns a given array to full storage mode.

Arguments

A

A row-indexed sparse array created by the `SPRSIN` function.

Keywords

None.

Examples

Suppose we have converted an array to sparse storage format with the following commands:

```
A = [[ 5.0, -0.2, 0.1], $
      [ 3.0, -2.0, 0.3], $
      [ 4.0, -1.0, 0.0]]
```

```
; Convert to sparse storage mode. All elements of the array A that
; have absolute values less than THRESH are set to zero:
sparse = SPRSIN(A, THRESH=0.5)
```

The variable `SPARSE` now contains a representation of the array `A` in structure form. To restore the array from the sparse-format structure:

```
; Restore the array:  
result = FULSTR(sparse)
```

```
; Print the result:  
PRINT, result
```

IDL prints:

```
5.00000      0.00000      0.00000  
3.00000     -2.00000      0.00000  
4.00000     -1.00000      0.00000
```

Note that the elements with an absolute value less than the specified threshold have been set to zero.

Version History

Introduced: 4.0

See Also

[LINBCG](#), [SPRSAB](#), [SPRSAX](#), [SPRSIN](#), [SPRSTP](#), [READ_SPR](#), [WRITE_SPR](#)

FUNCT

The FUNCT procedure evaluates the sum of a Gaussian and a 2nd-order polynomial and optionally returns the value of its partial derivatives. Normally, this function is used by CURVEFIT to fit the sum of a line and a varying background to actual data.

This routine is written in the IDL language. Its source code can be found in the file `funct.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
FUNCT, X, A, F [, Pder]
```

Arguments

X

A vector of values for the independent variable.

A

A vector of coefficients for the equations:

$$F = A_0 e^{-Z^2/2} + A_3 + A_4 X + A_5 X^2$$

$$Z = (X - A_1) / A_2$$

F

A named variable that will contain the value of the function at each X_i .

Pder

A named variable that will contain an array of the size $(N_ELEMENTS(X), 6)$ that contains the partial derivatives. $Pder(i, j)$ represents the derivative at the i^{th} point with respect to j^{th} parameter.

Version History

Introduced: 4.0

See Also

[CURVEFIT](#)

FUNCTION

The FUNCTION statement defines a function.

Note

For information on using the FUNCTION statement, see [Chapter 4, “Procedures and Functions”](#) in the *Building IDL Applications* manual.

Syntax

```
FUNCTION Function_Name, parameter1, ..., parametern
```

Version History

Introduced: Original

FV_TEST

The FV_TEST function computes the F-statistic and the probability that two sample populations X and Y have significantly different variances. X and Y may be of different lengths. The F-statistic formula for sample populations x and y with means \bar{x} and \bar{y} is defined as:

$$F = \left(\frac{M-1}{N-1} \right) \frac{\left[\sum_{j=0}^{N-1} (x_j - \bar{x})^2 - \frac{1}{N} \left[\sum_{j=0}^{N-1} (x_j - \bar{x}) \right]^2 \right]}{\left[\sum_{j=0}^{M-1} (y_j - \bar{y})^2 - \frac{1}{M} \left[\sum_{j=0}^{M-1} (y_j - \bar{y}) \right]^2 \right]}$$

where $x = (x_0, x_1, x_2, \dots, x_{N-1})$ and $y = (y_0, y_1, y_2, \dots, y_{M-1})$

This routine is written in the IDL language. Its source code can be found in the file `fv_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = FV_TEST(*X*, *Y*)

Return Value

The result is a two-element vector containing the F-statistic and its significance. The significance is a value in the interval [0.0, 1.0]; a small value (0.05 or 0.01) indicates that X and Y have significantly different variances. This type of test is often referred to as the F-variance test.

Arguments

X

An n -element integer, single- or double-precision floating-point vector.

Y

An m -element integer, single- or double-precision floating-point vector.

Keywords

None.

Examples

```
; Define two n-element sample populations:
X = [257, 208, 296, 324, 240, 246, 267, 311, 324, 323, 263, $
     305, 270, 260, 251, 275, 288, 242, 304, 267]
Y = [201,  56, 185, 221, 165, 161, 182, 239, 278, 243, 197, $
     271, 214, 216, 175, 192, 208, 150, 281, 196]

; Compute the F-statistic (of X and Y) and its significance:
PRINT, FV_TEST(X, Y)
```

IDL prints:

```
2.48578      0.0540116
```

The result indicates that X and Y have significantly different variances.

Version History

Introduced: 4.0

See Also

[KW_TEST](#), [MOMENT](#), [RS_TEST](#), [S_TEST](#), [TM_TEST](#)

FX_ROOT

The `FX_ROOT` function computes a real or complex root of a univariate nonlinear function using an optimal Müller's method.

This routine is written in the IDL language. Its source code can be found in the file `fx_root.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = FX_ROOT(X, Func [, /DOUBLE] [, ITMAX=value] [, /STOP]
               [, TOL=value] )
```

Return Value

The return value is the real or complex root of a univariate nonlinear function. Which root results depends on the initial guess provided for this routine.

Arguments

X

A 3-element real or complex initial guess vector. Real initial guesses may result in real or complex roots. Complex initial guesses will result in complex roots.

Func

A scalar string specifying the name of a user-supplied IDL function that defines the univariate nonlinear function. This function must accept the vector argument `X`.

For example, suppose we wish to find a root of the following function:

$$y = e^{(\sin x^2 + \cos x^2 - 1)} - 1$$

We write a function `FUNC` to express the function in the IDL language:

```
FUNCTION func, X
  RETURN, EXP(SIN(X)^2 + COS(X)^2 - 1) - 1
END
```

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ITMAX

The maximum allowed number of iterations. The default is 100.

STOP

Use this keyword to specify the stopping criterion used to judge the accuracy of a computed root $r(k)$. Setting `STOP = 0` (the default) checks whether the absolute value of the difference between two successively-computed roots, $|r(k) - r(k+1)|$ is less than the stopping tolerance `TOL`. Setting `STOP = 1` checks whether the absolute value of the function `FUNC` at the current root, $|FUNC(r(k))|$, is less than `TOL`.

TOL

Use this keyword to specify the stopping error tolerance. The default is 1.0×10^{-4} .

Examples

This example finds the roots of the function `FUNC` defined above:

```
; First define a real 3-element initial guess vector:
x = [0.0, -!pi/2, !pi]

; Compute a root of the function using double-precision
; arithmetic:
root = FX_ROOT(X, 'FUNC', /DOUBLE)

; Check the accuracy of the computed root:
PRINT, EXP(SIN(ROOT)^2 + COS(ROOT)^2 - 1) - 1
```

IDL prints:

```
0.0000000
```

We can also define a complex 3-element initial guess vector:

```
x = [COMPLEX(-!PI/3, 0), COMPLEX(0, !PI), COMPLEX(0, -!PI/6)]

; Compute the root of the function:
root = FX_ROOT(x, 'FUNC')
```

```
    ; Check the accuracy of the computed complex root:  
    PRINT, EXP(SIN(ROOT)^2 + COS(ROOT)^2 - 1) - 1
```

IDL prints:

```
(      0.00000,      0.00000)
```

Version History

Introduced: Pre 4.0

See Also

[BROYDEN](#), [NEWTON](#), [FZ_ROOTS](#)

FZ_ROOTS

The FZ_ROOTS function is used to find the roots of an m -degree complex polynomial, using Laguerre's method.

FZ_ROOTS is based on the routine `zroots` described in section 9.5 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = FZ_ROOTS(C [, /DOUBLE] [, EPS=value] [, /NO_POLISH] )
```

Return Value

Returns an m -element complex vector containing the roots of an m -degree complex polynomial.

Arguments

C

A vector of length $m+1$ containing the coefficients of the polynomial, in ascending order (see example). The type can be real or complex.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EPS

The desired fractional accuracy. The default value is 2.0×10^{-6} .

NO_POLISH

Set this keyword to suppress the usual polishing of the roots by Laguerre's method.

Examples

Example 1: Real coefficients yielding real roots.

Find the roots of the polynomial:

$$P(x) = 6x^3 - 7x^2 - 9x - 2$$

The exact roots are $-1/2, -1/3, 2.0$.

```
coeffs = [-2.0, -9.0, -7.0, 6.0]
roots = FZ_ROOTS(coeffs)
PRINT, roots
```

IDL prints:

```
( -0.500000, 0.000000)( -0.333333, 0.000000)( 2.000000, 0.000000)
```

Example 2: Real coefficients yielding complex roots.

Find the roots of the polynomial:

$$P(x) = x^4 + 3x^2 + 2$$

The exact roots are:

$$0.0 - i\sqrt{2.0}, 0.0 + i\sqrt{2.0}, 0.0 - i, 0.0 + i$$

```
coeffs = [2.0, 0.0, 3.0, 0.0, 1.0]
roots = FZ_ROOTS(coeffs)
PRINT, roots
```

IDL Prints:

```
(0.000000, -1.41421)(0.000000, 1.41421)
(0.000000, -1.00000)(0.000000, 1.00000)
```

Example 3: Real and complex coefficients yielding real and complex roots.

Find the roots of the polynomial:

$$P(x) = x^3 + (-4 - i4)x^2 + s(-3 + i4)x + (18 + i24)$$

The exact roots are $-2.0, 3.0, (3.0 + i4.0)$

```
coeffs = [COMPLEX(18,24), COMPLEX(-3,4), COMPLEX(-4,-4), 1.0]
roots = FZ_ROOTS(coeffs)
PRINT, roots
```

IDL Prints:

```
( -2.000000, 0.000000) ( 3.000000, 0.000000) ( 3.000000, 4.000000)
```

Version History

Introduced: 4.0

See Also

[FX_ROOT](#), [BROYDEN](#), [NEWTON](#), [POLY](#)

GAMMA

The GAMMA function returns the gamma function of Z .

The gamma function is defined as:

$$\Gamma(x) \equiv \int_0^{\infty} t^{x-1} e^{-t} dt$$

Use the LNGAMMA function to obtain the natural logarithm of the gamma function when there is a possibility of overflow.

Syntax

Result = GAMMA(Z)

Return Value

If Z is double-precision, the result is double-precision (either double or double complex), otherwise the result is single-precision (either float or complex).

Arguments

Z

The expression for which the gamma function will be evaluated. Z may be complex.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Plot the gamma function over the range 0.01 to 1.0 with a step size of 0.01 by entering:

```
Z = FINDGEN(99)/100. + 0.01  
PLOT, Z, GAMMA(Z)
```

Version History

Introduced: Original

Z argument accepts complex input: 5.6

See Also

[BETA](#), [IBETA](#), [IGAMMA](#), [LNGAMMA](#)

GAMMA_CT

The GAMMA_CT procedure applies gamma correction to a color table.

This routine is written in the IDL language. Its source code can be found in the file `gamma_ct.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
GAMMA_CT, Gamma [, /CURRENT] [, /INTENSITY]
```

Arguments

Gamma

The value of gamma correction. A value of 1.0 indicates a linear ramp (i.e., no gamma correction). Higher values of *Gamma* give more contrast. Values less than 1.0 yield lower contrast.

Keywords

CURRENT

Set this keyword to apply correction from the “current” color table (i.e., the values R_CURR, G_CURR, and B_CURR in the COLORS common block). Otherwise, correction is applied from the “original” color table (i.e., the values R_ORIG, G_ORIG, and B_ORIG in the COLORS common block). The gamma corrected color table is always saved in the “current” table (R_CURR, G_CURR, B_CURR) and the new table is loaded.

INTENSITY

Set this keyword to correct the individual intensities of each color in the colortable. Otherwise, the colors are shifted according to the gamma function.

Version History

Introduced: Pre 4.0

See Also

[PSEUDO](#), [STRETCH](#), [XLOADCT](#)

GAUSS_CVF

The GAUSS_CVF function computes the cutoff value V in a standard Gaussian (normal) distribution with a mean of 0.0 and a variance of 1.0 such that the probability that a random variable X is greater than V is equal to a user-supplied probability P .

This routine is written in the IDL language. Its source code can be found in the file `gauss_cvf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = GAUSS_CVF(*P*)

Return Value

Returns the cutoff value V in a standard Gaussian (normal) distribution with a mean of 0.0 and a variance of 1.0.

Arguments

P

A non-negative single- or double-precision floating-point scalar, in the interval [0.0, 1.0], that specifies the probability of occurrence or success.

Keywords

None.

Examples

Use the following command to compute the cutoff value in a Gaussian distribution such that the probability that a random variable X is greater than the cutoff value is 0.025:

```
PRINT, GAUSS_CVF(0.025)
```

IDL prints:

```
1.95997
```

Version History

Introduced: 4.0

See Also

[CHISQR_CVF](#), [F_CVF](#), [GAUSS_PDF](#), [T_CVF](#)

GAUSS_PDF

The GAUSS_PDF function computes the probability P that, in a standard Gaussian (normal) distribution with a mean of 0.0 and a variance of 1.0, a random variable X is less than or equal to a user-specified cutoff value V .

This routine is written in the IDL language. Its source code can be found in the file `gauss_pdf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = GAUSS_PDF(*V*)

Return Value

This function returns a scalar or array with the same dimensions as V . If V is double-precision, the result is double-precision, otherwise the result is single-precision.

Arguments

V

A scalar or array that specifies the cutoff value(s).

Keywords

None.

Examples

Example 1

Compute the probability that a random variable X , from the standard Gaussian (normal) distribution, is less than or equal to 2.44:

```
PRINT, GAUSS_PDF(2.44)
```

IDL Prints:

```
0.992656
```

Example 2

Compute the probability that a random variable X, from the standard Gaussian (normal) distribution, is less than or equal to 10.0 and greater than or equal to 2.0:

```
PRINT, GAUSS_PDF(10.0) - GAUSS_PDF(2.0)
```

IDL Prints:

```
0.0227501
```

Example 3

Compute the probability that a random variable X, from the Gaussian (normal) distribution with a mean of 0.8 and a variance of 4.0, is less than or equal to 2.44:

```
PRINT, GAUSS_PDF( (2.44 - 0.80)/SQRT(4.0) )
```

IDL Prints:

```
0.793892
```

Version History

Introduced: 4.0

See Also

[BINOMIAL](#), [CHISQR_PDF](#), [F_PDF](#), [GAUSS_CVF](#), [T_PDF](#)

GAUSS2DFIT

The GAUSS2DFIT function fits a two-dimensional, elliptical Gaussian equation to rectilinearly gridded data.

$$Z = F(x, y)$$

where:

$$F(x, y) = A_0 + A_1 e^{-U/2}$$

And the elliptical function is:

$$U = (x'/a)^2 + (y'/b)^2$$

The parameters of the ellipse U are:

- Axis lengths are $2a$ and $2b$, in the unrotated X and Y axes, respectively.
- Center is at (h, k) .
- Rotation of T radians from the X axis, in the *clockwise* direction.

The rotated coordinate system is defined as:

$$x' = (x - h)\cos T - (y - k)\sin T$$

$$y' = (x - h)\sin T + (y - k)\cos T$$

The rotation is optional, and can be forced to 0, making the major and minor axes of the ellipse parallel to the X and Y axes.

Coefficients of the computed fit are returned in argument A .

Procedure Used and Other Notes

The peak/valley is found by first smoothing Z and then finding the maximum or minimum, respectively. GAUSSFIT is then applied to the row and column running through the peak/valley to estimate the parameters of the Gaussian in X and Y . Finally, CURVEFIT is used to fit the 2D Gaussian to the data.

Be sure that the 2D array to be fit contains the entire peak/valley out to at least 5 to 8 half-widths, or the curve-fitter may not converge.

This is a computationally-intensive routine. The time required is roughly proportional to the number of elements in Z .

This routine is written in the IDL language. Its source code can be found in the file `gauss2dfit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = GAUSS2DFIT( Z, A [, X, Y] [, /NEGATIVE] [, /TILT ] )
```

Arguments

Z

The dependent variable. Z should be a two-dimensional array with dimensions (N_x , N_y). Gridding in the array must be rectilinear.

A

A named variable in which the coefficients of the fit are returned. A is returned as a seven element vector the coefficients of the fitted function. The meanings of the seven elements in relation to the discussion above is:

- $A[0] = A_0 =$ constant term
- $A[1] = A_1 =$ scale factor
- $A[2] = a =$ width of Gaussian in the X direction
- $A[3] = b =$ width of Gaussian in the Y direction
- $A[4] = h =$ center X location
- $A[5] = k =$ center Y location.
- $A[6] = T =$ *Theta*, the rotation of the ellipse from the X axis in radians, *counter-clockwise*.

X

An optional vector with N_x elements that contains the X values of Z (i.e., X_i is the X value for $Z_{i,j}$). If this argument is omitted, a regular grid in X is assumed, and the X location of $Z_{i,j} = i$.

Y

An optional vector with N_y elements that contains the Y values of Z (i.e., Y_j is the Y value for $Z_{i,j}$). If this argument is omitted, a regular grid in Y is assumed, and the Y location of $Z_{i,j} = j$.

Keywords

NEGATIVE

Set this keyword to indicate that the Gaussian to be fitted is a valley (such as an absorption line). By default, a peak is fit.

TILT

Set this keyword to allow the orientation of the major and minor axes of the ellipse to be unrestricted. The default is that the axes of the ellipse must be parallel to the X and Y axes. Therefore, in the default case, A[6] is always returned as 0.

Examples

This example creates a 2D gaussian, adds random noise and then applies GAUSS2DFIT.

```

; Define array dimensions:
nx = 128 & ny = 100
; Define input function parameters:
A = [ 5., 10., nx/6., ny/10., nx/2., .6*ny]
; Create X and Y arrays:
X = FINDGEN(nx) # REPLICATE(1.0, ny)
Y = REPLICATE(1.0, nx) # FINDGEN(ny)
; Create an ellipse:
U = ((X-A[4])/A[2])^2 + ((Y-A[5])/A[3])^2
; Create gaussian Z:
Z = A[0] + A[1] * EXP(-U/2)
; Add random noise, SD = 1:
Z = Z + RANDOMN(seed, nx, ny)
; Fit the function, no rotation:
yfit = GAUSS2DFIT(Z, B)
; Report results:
PRINT, 'Should be: ', STRING(A, FORMAT='(6f10.4)')
PRINT, 'Is: ', STRING(B(0:5), FORMAT='(6f10.4)')
```

Version History

Introduced: 4.0.1

See Also

[COMFIT](#), [GAUSSFIT](#), [POLY_FIT](#), [REGRESS](#), [SFIT](#), [SVDFIT](#)

GAUSSFIT

The GAUSSFIT function computes a non-linear least-squares fit to a function $f(x)$ with from three to six unknown parameters. $f(x)$ is a linear combination of a Gaussian and a quadratic; the number of terms is controlled by the keyword parameter NTERMS.

This routine is written in the IDL language. Its source code can be found in the file `gaussfit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = GAUSSFIT( X, Y [, A] [, CHISQ=variable] [, ESTIMATES=array]
  [, MEASURE_ERRORS=vector] [, NTERMS=integer{3 to 6}]
  [, SIGMA=variable] [, YERROR=variable])
```

Return Value

Returns three to six values that are the non-linear least squares fit to a function $f(x)$.

Arguments

X

An n -element vector of independent variables.

Y

A vector of dependent variables, the same length as X.

A

A named variable that will contain the coefficients A of the fit.

Keywords

CHISQ

Set this keyword to a named variable that will contain the value of the reduced chi-square goodness-of-fit statistic.

ESTIMATES

Set this keyword equal to an array of starting estimates for the parameters of the equation. If the NTERMS keyword is specified, the ESTIMATES array should have NTERMS elements. If NTERMS is not specified, the ESTIMATES array should have six elements. If the ESTIMATES array is not specified, estimates are calculated by first subtracting a constant (if NTERMS \geq 4) or a linear term (if NTERMS \geq 5), and then forming a simple estimate of the Gaussian coefficients.

Note

If ESTIMATES[2] (the Gaussian width) is zero, then GAUSSFIT will automatically compute a best guess for the starting value for this term.

MEASURE_ERRORS

Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y .

Note

For Gaussian errors (e.g., instrumental uncertainties), MEASURE_ERRORS should be set to the standard deviations of each point in Y . For Poisson or statistical weighting, MEASURE_ERRORS should be set to $\text{SQRT}(Y)$.

NTERMS

Set this keyword to an integer value between three and six to specify the function to be used for the fit. The values correspond to the functions shown below. In all cases:

$$z = \frac{x - A_1}{A_2}$$

NTERMS=6

$$f(x) = A_0 e^{\frac{-z^2}{2}} + A_3 + A_4 x + A_5 x^2$$

NTERMS=5

$$f(x) = A_0 e^{\frac{-z^2}{2}} + A_3 + A_4 x$$

NTERMS=4

$$f(x) = A_0 e^{\frac{-z^2}{2}} + A_3$$

NTERMS=3

$$f(x) = A_0 e^{\frac{-z^2}{2}}$$

NTERMS=6 is the default setting. Here, A0 is the height of the Gaussian, A1 is the center of the Gaussian, A2 is the width (the standard deviation) of the Gaussian, A3 is the constant term, A4 is the linear term, and A5 is the quadratic term.

Tip

The full width at half maximum (FWHM) of the Gaussian may be computed as $2 * \text{SQRT}(2 * \text{ALOG}(2)) * A2$.

SIGMA

Set this keyword to a named variable that will contain the 1-sigma error estimates of the returned parameters.

Note

If MEASURE_ERRORS is omitted, then you are assuming that a polynomial is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by $\text{SQRT}(\text{CHISQ}/(N * M))$, where N is the number of points in X , and M is the number of coefficients. See Section 15.2 of *Numerical Recipes in C (Second Edition)* for details.

YERROR

Set this keyword to a named variable that will contain the standard error between YFIT and Y.

Examples

The following example shows how to use GAUSSFIT to fit to four different functions, with NTERMS=3,4,5,6. To simulate actual data, random noise has been added to each function.

```

pro ex_gaussfit

    ; Define the independent variable.
    n = 101
    x = (FINDGEN(n)-(n/2))/4

    ; Define the coefficients.
    a = [4.0, 1.0, 2.0, 1.0, 0.25, 0.01]
    print, 'Expected: ', a
    z = (x - a[1])/a[2]      ; Gaussian variable
    !P.MULTI = [0,2,2]      ; set up 2x2 plot window
    seed = 123321           ; Pick a starting seed value

    for nterms=3,6 do begin
        ; Define the dependent variables. Start with random noise.
        y = 0.4*RANDOMN(seed, n)

        ; Use a switch statement so we fall through to each term.
        switch nterms of
            6: y = y + a[5]*x^2
            5: y = y + a[4]*x
            4: y = y + a[3]
            3: y = y + a[0]*exp(-z^2/2)
        endswitch

        ; Fit the data to the function, storing coefficients in
        ; coeff:
        yfit = GAUSSFIT(x, y, coeff, NTERMS=nterms)
        print, 'Result: ', coeff[0:nterms-1]
        ; Plot the original data and the fitted curve:
        PLOT, x, y, TITLE='nterms='+STRTRIM(nterms,2)
        OPLOT, x, yfit, THICK=2
    endfor
end

```

When this program is compiled and executed, IDL prints the following results:

```

IDL> ex_gaussfit
Expected: 4.00000 1.00000 2.00000 1.00000 0.250000 0.0100000
Result: 3.95437 1.03176 2.07216
Result: 4.38669 0.948479 1.84426 0.909676
Result: 3.93348 0.839296 2.02112 1.05237 0.249002
Result: 3.80389 0.993375 2.07302 1.16684 0.249051 0.0484357

```

Version History

Introduced: Original

CHISQ, SIGMA, and YERROR keywords added: 5.6

MEASURE_ERRORS keyword added: 6.0

See Also

[COMFIT](#), [CURVEFIT](#), [GAUSS2DFIT](#), [POLY_FIT](#), [REGRESS](#), [SFIT](#), [SVDFIT](#)

GAUSSINT

The GAUSSINT function evaluates the integral of the Gaussian probability function.

The Gaussian integral is defined as:

$$\text{Gaussint}(x) \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt$$

Syntax

Result = GAUSSINT(*X*)

Return Value

Returns the result of the Gaussian probability function integral evaluation. If *X* is double-precision, the result is double-precision, otherwise the argument is converted to floating-point and the result is floating-point. The result has the same structure as the input argument, *X*.

Arguments

X

The expression for which the Gaussian integral is to be evaluated.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Plot the Gaussian probability function over the range -5 to 5 with a step size of 0.1 by entering:

```
X = FINDGEN(101)/10. - 5.  
PLOT, X, GAUSSINT(X)
```

Version History

Introduced: Original

See Also

[GAUSS_CVF](#), [GAUSS_PDF](#)

GET_DRIVE_LIST

The GET_DRIVE_LIST function returns valid drive or volume names for the file system. Under Microsoft Windows, keywords can be used to specify that only drives of certain types should be returned.

Note

The UNIX operating system presents all files within a single unified file hierarchy, and does not support the concept of drive letters or volume names. As such, GET_DRIVE_LIST always returns a scalar null string under UNIX.

Syntax

Result = GET_DRIVE_LIST([COUNT=*variable*])

Windows-Only Keywords: [, /CDROM] [, /FIXED] [, /REMOTE]
[, /REMOVABLE]

Return Value

Returns a string array of the names of valid drives / volumes for the file system. If GET_DRIVE_LIST has no drives to return, it returns a scalar null string.

Arguments

None.

Keywords

Note

If a Windows-only keyword is specified, only drives of the specified types are reported.

CDROM

This keyword is only available on Windows platforms.

If set, compact disk drives are reported. Note that although CDROM devices are removable, they are treated as a special case, and the REMOVABLE keyword does not apply to them.

COUNT

A named variable into which the number of drives/volumes found is placed. If no drives/volumes are found, a value of zero is returned. Under UNIX, the value returned by this keyword will always be zero.

FIXED

This keyword is only available on Windows platforms.

If set, hard drives physically attached to the current system are reported.

REMOTE

This keyword is only available on Windows platforms.

If set, remote (i.e. network) drives are reported.

REMOVABLE

This keyword is only available on Windows platforms.

If set, removable media devices (e.g. floppy, zip drive) other than CDROMs are reported.

Examples

Under Windows, the following will report all local hard drives:

```
drives = GET_DRIVE_LIST(/FIXED)
```

This statement obtains the names of all floppy drives, cdroms, and other removable media drives:

```
drives = GET_DRIVE_LIST(/CDROM, /REMOVABLE)
```

Version History

Introduced: 5.3

GET_KBRD

The GET_KBRD function returns the next character available from the standard input (IDL file unit 0).

Warning

GET_KBRD is intended for use in IDL's UNIX command-line mode. While GET_KBRD will return values for some characters when run from the IDL Development Environment (either UNIX or Microsoft Windows), other characters are treated as special cases by the underlying windowing system, and may not be returned by GET_KBRD. In addition, the event-handling mechanism used by the IDL Development Environment may not respond to keyboard events in the way that you expect. Use caution when using GET_KBRD within a loop.

Syntax

Result = GET_KBRD(*Wait*)

Return Value

Returns a one-character string containing the next available character that is input from the keyboard.

Arguments

Wait

If *Wait* is zero, GET_KBRD returns the null string if there are no characters in the terminal type-ahead buffer. If it is nonzero, the function waits for a character to be typed before returning.

Keywords

None.

Examples

To wait for keyboard input and store one character in the variable R, enter:

```
R = GET_KBRD(1)
```

Press any key to return to the IDL prompt. To see the character that was typed, enter:

```
PRINT, R
```

The following code fragment reads one character at a time and echoes that character's numeric code. It quits when a "q" is entered:

```
REPEAT BEGIN
  A = GET_KBRD(1)
  PRINT, BYTE(A)
ENDREP UNTIL A EQ 'q'
```

Note

The GET_KBRD function can be used to return Windows special characters (in addition to standard keyboard characters), created by holding down the Alt key and entering the character's ANSI equivalent. For example, to return the paragraph marker (¶), ANSI number 0182, enter:

```
C = GET_KBRD(1)
```

While GET_KBRD is waiting, press and hold the Alt key and type 0182 on the numeric keypad. When the IDL prompt returns, enter:

```
PRINT, C
```

IDL prints the paragraph marker, "¶".

GET_KBRD *cannot* be used to return control characters or other editing keys (e.g., Delete, Backspace, etc.). These characters are used for keyboard shortcuts and command line editing only. GET_KBRD can be used to return the Enter key.

Version History

Introduced: Original

See Also

[READ/READF](#)

GET_LUN

The GET_LUN procedure allocates a file unit from a pool of free units. Instead of writing routines to assume the use of certain file units, IDL functions and procedures should use GET_LUN to reserve unit numbers in order to avoid conflicts with other routines. Use FREE_LUN to free the file units when finished.

Syntax

```
GET_LUN, Unit
```

Arguments

Unit

The named variable into which GET_LUN should place the file unit number. *Unit* is converted into a longword integer in the process. The file unit number obtained is in the range 100 to 128.

Keywords

None.

Examples

Instead of explicitly specifying a file unit number that may already be used, use GET_LUN to obtain a free one and store the result in the variable U by entering:

```
GET_LUN, U
```

Now U can be used in opening a file:

```
OPENR, U, 'file.dat'
```

Once the data from “file.dat” has been read, the file can be closed and the file unit can be freed with the command:

```
FREE_LUN, U
```

Note also that OPENR has a GET_LUN keyword that allows you to simultaneously obtain a free file unit and open a file. The following command performs the same tasks as the first two commands above:

```
OPENR, U, 'file.dat', /GET_LUN
```

Version History

Introduced: Original

See Also

[FREE_LUN](#), [OPEN](#)

GET_SCREEN_SIZE

The `GET_SCREEN_SIZE` function returns size, measured in device units, of the screen.

Syntax

Result = `GET_SCREEN_SIZE`([*Display_name*] [, **RESOLUTION**=*variable*])

Unix-Only Keywords: [, **DISPLAY_NAME**=*string*]

Return Value

Returns a two-element vector of the form [*width*, *height*] that represents the dimensions, measured in device units, of the screen

Arguments

Display_name

This argument is only available on UNIX platforms.

A string indicating the name of the X Windows display that should be used to determine the screen size.

Keywords

DISPLAY_NAME

This keyword is only available on UNIX platforms.

Set this keyword equal to a string indicating the name of the X Windows display that should be used to determine the screen size. Setting this keyword is equivalent to setting the optional *Display_name* argument.

RESOLUTION

Set this keyword equal to a named variable that will contain a two-element vector, [*xres*, *yres*], specifying the screen resolution in cm/pixel.

Examples

You can find the dimensions and screen resolution of your screen by entering the following:

```
dimensions = GET_SCREEN_SIZE(RESOLUTION=resolution)
PRINT, dimensions, resolution
```

For the screen on which this was tested, IDL prints:

```
1280.00      1024.00
0.0282031    0.0281250
```

Version History

Introduced: 5.0

GETENV

The GETENV function returns the value of one or more specified environment variables from the environment of the IDL process.

About the Process Environment

Every process has an *environment* consisting of environment variables, each of which has an associated string value. Some environment variables always exist, such as PATH, which tells the shell where to look for programs. Others can be added by the user, either interactively via a shell, via a UNIX startup file such as `.login`, or a via a Windows control panel.

When a process is created, it is given a copy of the environment from its parent process. IDL is no exception to this; when started, it inherits a copy of the environment of its parent process, which may be an interactive shell, the windowing system's desktop environment, or some other process. In turn, any child process created by IDL (such as those from the SPAWN procedure) inherits a copy of IDL's current environment.

Note

It is important to realize that environment variables are not an IDL feature; they are part of every process. Although they can serve as a form of global memory, it is best to avoid using them in that way. Instead, IDL heap variables (pointers or object references), IDL system variables, or common blocks should be used in that role. Environment variables should be used for communicating with child processes. One example is setting the value of the SHELL environment variable prior to calling SPAWN to change the shell executed by SPAWN.

Syntax

Result = GETENV(*Name* [, /ENVIRONMENT])

Return Value

Returns the value of the environment variable *Name* from the environment of the IDL process, or a null string if *Name* does not exist in the environment. If *Name* is an array, the result has the same structure, with each element containing the value for the corresponding element of *Name*.

Arguments

Name

A scalar string or string array variable containing the names of environment variables for which values are desired.

Special Handling of the IDL_TMPDIR Environment Variable

If you specify 'IDL_TMPDIR' as the value of *Name*, and an environment variable with that name exists, GETENV returns its defined value as usual. However, if IDL_TMPDIR is *not* defined, GETENV returns the path of the location where IDL's internals believe temporary files should be written on your system:

- On UNIX systems, IDL uses the value of the standard TMPDIR environment variable. If TMPDIR is not defined, IDL chooses a reasonable temporary directory based on operating system and vendor conventions.
- On Windows systems, IDL uses the value provided by Windows, which is the first of the following that is defined: the value of the TMP environment variable, the value of the TEMP environment variable, or the default value for the current Windows version.

Using IDL_TMPDIR in this manner makes it simple for code written in IDL to use the same temporary directory as IDL itself uses, and provides an easy way for the user to override the default.

Keywords

ENVIRONMENT

Set this keyword to return a string array containing all entries in the current process, one variable per entry, in the SETENV format (*Variable=Value*). If ENVIRONMENT is set, the *Name* argument should not be supplied.

Examples

To print the name of the current UNIX shell, enter the command:

```
PRINT, 'The current shell is: ', GETENV('SHELL')
```

To store the path to the directory where IDL believes temporary files should be placed in the variable `mytemp`, use the following statement:

```
mytemp = GETENV('IDL_TMPDIR')
```

See Also

[SETENV](#)

Version History

Introduced: Original

ENVIRONMENT keyword modified to work on all platforms: 6.1

GOTO

The GOTO statement transfers program control to point specified by a label. The GOTO statement is generally considered to be a poor programming practice that leads to unwieldy programs. Its use should be avoided. However, for those cases in which the use of a GOTO is appropriate, IDL does provide the GOTO statement.

Note that using a GOTO to jump into the middle of a loop results in an error.

Warning

You must be careful in programming with GOTO statements. It is not difficult to get into a loop that will never terminate, especially if there is not an escape (or test) within the statements spanned by the GOTO.

For information on using GOTO and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

GOTO, *label*

Examples

In the following example, the statement at label JUMP1 is executed after the GOTO statement, skipping any intermediate statements:

```
GOTO, JUMP1
PRINT, 'Skip this' ; This statement is skipped
PRINT, 'Skip this' ; This statement is also skipped
JUMP1: PRINT, 'Do this'
```

The label can also occur before the GOTO statement that refers to the label, but you must be careful to avoid an endless loop. GOTO statements are frequently the subjects of IF statements, as in the following statement:

```
IF A NE G THEN GOTO, MISTAKE
```

Version History

Introduced: Original

GRID_INPUT

The GRID_INPUT procedure preprocesses and sorts two-dimensional scattered data points, and removes duplicate values. This procedure is also used for converting spherical coordinates to Cartesian coordinates.

Note

GRID_INPUT accepts complex input but only utilizes the real part of any complex number.

Syntax

```
GRID_INPUT, X, Y, F, XI, YI, FI [, DUPPLICATES=string ] [, EPSILON=value ]
[, EXCLUDE=vector ]
```

or

```
GRID_INPUT, Lon, Lat, F, Xyz, FI, /SPHERE [, /DEGREES]
[, DUPPLICATES=string ] [, EPSILON=value ] [, EXCLUDE=vector ]
```

or

```
GRID_INPUT, R, Theta, F, XI, YI, FI, /POLAR [, /DEGREES]
[, DUPPLICATES=string ] [, EPSILON=value ] [, EXCLUDE=vector ]
```

Arguments

X, Y

These are input arguments for scattered data points, where X, and Y are location. All of these arguments are N point vectors.

F

The function value at each location in the form of an N point vector.

Lon, Lat

These are input arguments representing scattered data points on a sphere, specifying location (longitude and latitude). All are N point vectors. Lon, Lat are in degrees or radians (default).

R, Theta

These are scattered data point input arguments representing the *R* and *Theta* polar coordinate location in degrees or radians (default). All arguments are *N* point vectors.

X1, Y1, F1

These output arguments are processed and sorted single precision floating point data which are passed as the input points to the GRIDDATA function.

Xyz

Upon return, a named variable that contains a 3-by-*n* array of Cartesian coordinates representing points on a sphere.

Keywords

DEGREES

By default, all angular inputs and keywords are assumed to be in radian units. Set the DEGREES keyword to change the angular input units to degrees.

DUPLICATES

Set this keyword to a string indicating how duplicate data points are handled per the following table. The case (upper or lower) is ignored. The default setting for DUPLICATES is "First".

String	Meaning
"First"	Retain only the first encounter of the duplicate locations.
"Last"	Retain only the last encounter of the duplicate locations.
"All"	Retains all locations, which is invalid for any gridding technique that requires a TRIANGULATION. Some methods, such as Inverse Distance or Polynomial Regression with no search criteria can handle duplicates.
"Avg"	Retain the average F value of the duplicate locations.

Table 3-42: DUPLICATE Keyword Values

String	Meaning
"Midrange"	Retain the average of the minimum and maximum duplicate locations $((\text{Max}(F) + \text{Min}(F)) / 2)$.
"Min"	Retain the minimum of the duplicate locations $(\text{Min}(F))$.
"Max"	Retain the maximum of the duplicate locations $(\text{Max}(F))$.

Table 3-42: DUPLICATE Keyword Values (Continued)

EPSILON

The tolerance for finding duplicates. Points within EPSILON distance of each other are considered duplicates. For spherical coordinates, EPSILON is in units of angular distance, as set by the DEGREES keyword.

EXCLUDE

An N -point vector specifying the indices of the points to exclude.

POLAR

Set to indicate inputs are in polar coordinates.

SPHERE

Set to indicate inputs are in spherical coordinates. In this case, the output argument Xyz is set to a 3-by- n array containing the spherical coordinates converted to 3-dimensional Cartesian points on a sphere.

Examples

The following example uses the data from the `irreg_grid1.txt` ASCII file included in the `examples/data` subdirectory of the IDL distribution. This file contains scattered elevation data of a model of an inlet. This scattered elevation data contains two duplicate locations. The `GRID_INPUT` procedure is used to omit the duplicate locations.

```

; Import the Data:

; Determine the path to the file.
```

```

file = FILE_SEARCH(!DIR, 'irreg_grid1.txt')

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
data = dataArray[*, 2]

; Display the Data:

; Scale the data to range from 1 to 253 so a color table can be
; applied. The values of 0, 254, and 255 are reserved as outliers.
scaled = BYTSCL(data, TOP = !D.TABLE_SIZE - 4) + 1B

; Load the color table. If you are on a TrueColor, set the
; DECOMPOSED keyword to the DEVICE command before running a
; color table related routine.
DEVICE, DECOMPOSED = 0
LOADCT, 38

; Open a display window and plot the data points.
WINDOW, 0
PLOT, x, y, /XSTYLE, /YSTYLE, LINESSTYLE = 1, $
    TITLE = 'Original Data, Scaled (1 to 253)', $
    XTITLE = 'x', YTITLE = 'y'

; Now display the data values with respect to the color table.
FOR i = 0L, (N_ELEMENTS(x) - 1) DO PLOTS, x[i], y[i], PSYM = -1, $
    SYMSIZE = 2., COLOR = scaled[i]

; Preprocess and sort the data. GRID_INPUT will
; remove any duplicate locations.
GRID_INPUT, x, y, data, xSorted, ySorted, dataSorted

; Display the results from GRID_INPUT:

; Scale the resulting data.
scaled = BYTSCL(dataSorted, TOP = !D.TABLE_SIZE - 4) + 1B

; Open a display window and plot the resulting data points.
WINDOW, 1
PLOT, xSorted, ySorted, /XSTYLE, /YSTYLE, LINESSTYLE = 1, $

```

```
TITLE = 'The Data Preprocessed and Sorted, Scaled (1 to 253)', $
XTITLE = 'x', YTITLE = 'y'

; Now display the resulting data values with respect to the color
; table.
FOR i = 0L, (N_ELEMENTS(xSorted) - 1) DO PLOTS, $
    xSorted[i], ySorted[i], PSYM = -1, COLOR = scaled[i], $
    SYMSIZE = 2.
```

Version History

Introduced: 5.5

See Also

[GRIDDATA](#)

GRID_TPS

The GRID_TPS function uses thin plate splines to interpolate a set of values over a regular two dimensional grid, from irregularly sampled data values. Thin plate splines are ideal for modeling functions with complex local distortions, such as warping functions, which are too complex to be fit with polynomials.

Given n points, (x_i, y_i) in the plane, a thin plate spline can be defined as:

$$f(x, y) = a_0 + a_1x + a_2y + \frac{1}{2} \sum_{i=0}^{n-1} b_i r_i^2 \log r_i^2$$

with the constraints:

$$\sum_{i=1}^{n-1} b_i = \sum_{i=1}^{n-1} b_i x_i = \sum_{i=0}^{n-1} b_i y_i = 0$$

where $r_i^2 = (x-x_i)^2 + (y-y_i)^2$. A thin plate spline (TPS) is a smooth function, which implies that it has continuous first partial derivatives. It also grows almost linearly when far away from the points (x_i, y_i) . The TPS surface passes through the original points: $f(x_i, y_i) = z_i$.

Note

GRID_TPS requires at least 7 noncolinear points.

Syntax

Result = GRID_TPS (*Xp*, *Yp*, *Values* [, **COEFFICIENTS**=*variable*] [, **NGRID**=[*nx*, *ny*] [, **START**=[*x0*, *y0*] [, **DELTA**=[*dx*, *dy*]])

Return Value

An array of dimension (nx, ny) of interpolated values. If the values argument is a two-dimensional array, the output array has dimensions (nz, nx, ny) , where nz is the leading dimension of the values array allowing for the interpolation of arbitrarily sized vectors in a single call. Keywords can be used to specify the grid dimensions, size, and location.

Note

If the Cholesky factorization used within GRID_TPS fails, then *Result* will be a scalar -1.

Arguments

Xp

A vector of x points.

Yp

A vector of y points, with the same number of elements as the *Xp* argument.

Values

A vector or two-dimensional array of values to interpolate. If values are a two-dimensional array, the leading dimension is the number of values for which interpolation is performed.

Keywords

COEFFICIENTS

A named variable in which to store the resulting coefficients of the thin plate spline function for the last set of Values. The first N elements, where N is the number of input points, contain the coefficients b_i , in the previous equation. Coefficients with subscripts n , $n+1$, and $n+2$, contain the values of a_0 , a_1 , and a_2 , in the above equation.

DELTA

A two-element array specifying the grid spacing in X and Y. If this keyword is not specified, or if one of the elements is set equal to zero, the grid spacing is determined from the values of the NGRID and START keywords, according to the following rules:

$$\text{DELTA}_0 = (\text{MAX}(Xp) - \text{START}_0) / (\text{NGRID}_0 - 1)$$

$$\text{DELTA}_1 = (\text{MAX}(Yp) - \text{START}_1) / (\text{NGRID}_1 - 1)$$

DELTA can also be set to a scalar value to be used for the grid size in both X and Y.

NGRID

A two-element array of the size of the grid to interpolate (n_x, n_y). If a scalar is passed, the value is used for both n_x and n_y . The default value is [25, 25].

START

A two-element array of the location of grid point (x_0, y_0). If a scalar is passed, the value is used for both x_0 and y_0 . The default is the minimum values in the xp and yp arrays.

References

I. Barrodale, et al, “Note: Warping digital images using thin plate splines”, Pattern Recognition, Vol 26, No. 2, pp 375-376, 1993.

M. J. D. Powell, “Tabulation of thin plate splines on a very fine two-dimensional grid”, Report No. DAMTP 1992/NA2, University of Cambridge, Cambridge, U.K. (1992).

Examples

The following example creates a set of 25 random values defining a surface on a square, 100 units on a side, starting at the origin. Then, we use GRID_TPS to create a regularly gridded surface, with dimensions of 101 by 101 over the square, which is then displayed. The same data set is then interpolated using TRIGRID, and the two results are displayed for comparison.

```

;X values
x = RANDOMU(seed, 25) * 100

;Y values
y = RANDOMU(seed, 25) * 100

;Z values
z = RANDOMU(seed, 25) * 10

z1 = GRID_TPS(x, y, z, NGRID=[101, 101], START=[0,0], DELTA=[1,1])

;Show the result
SHADE_SURF, z1, TITLE='TPS'

;Grid using TRIGRID
TRIANGULATE, x, y, tr, bounds

```

```
z2 = TRIGRID(x, y, z, tr, [1,1], [0,0,100, 100], $  
    EXTRAPOLATE=bounds)  
  
;Show triangulated surface  
SHADE_SURF, z2, TITLE='TRIGRID - Quintic'
```

Version History

Introduced: 5.2

See Also

[MIN_CURVE_SURF](#)

GRID3

The GRID3 function fits a smooth function to a set of 3D scattered nodes (x_i, y_i, z_i) with associated data values (f_i) . The function can be sampled over a set of user-specified points, or over an arbitrary 3D grid which can then be viewed using the SLICER3 procedure.

GRID3 uses the method described in Renka, R. J., “Multivariate Interpolation of Large Sets of Scattered Data,” *ACM Transactions on Mathematical Software*, Vol. 14, No. 2, June 1988, Pages 139-148, which has been referred to as the Modified Shepard’s Method. The function described by this method has the advantages of being equal to the values of f_i , at each (x_i, y_i, z_i) , and being smooth (having continuous first partial derivatives).

Syntax

Result = GRID3(*X*, *Y*, *Z*, *F*, *Gx*, *Gy*, *Gz* [, DELTA=*scalar/vector*] [, DTOL=*value*] [, GRID=*value*] [, NGRID=*value*] [, START=*[x, y, z]*])

Return Value

If no optional or keyword parameters are supplied, GRID3 produces a regularly-sampled volume with dimensions of (25, 25, 25), made up of single-precision, floating-point values, enclosing the original data points.

Arguments

X, Y, Z and F

Arrays containing the locations of the data points, and the value of the variable to be interpolated at that point. *X*, *Y*, *Z*, and *F* must have the same number of elements (with a minimum of 10 elements per array) and are converted to floating-point if necessary.

Note

For the greatest possible accuracy, the arrays *X*, *Y*, and *Z* should be scaled to fit in the range [0,1].

G_x, G_y, and G_z

Optional arrays containing the locations within the volume to be sampled (if the GRID keyword is not set), or the locations along each axis of the sampling grid (if the

GRID keyword is set). If these parameters are supplied, the keywords DELTA, NGRID, and START are ignored.

If the keyword GRID is *not* set, the result has the same number of elements as G_x , G_y , and G_z . The i th element of the result contains the value of the interpolate at (G_{xi}, G_{yi}, G_{zi}) . The result has the same dimensions as G_x .

If the GRID keyword is set, the result of GRID3 is a three-dimensional, single-precision, floating-point array with dimensions of (N_x, N_y, N_z) , where N_x , N_y , and N_z are the number of elements in G_x , G_y , and G_z , respectively.

Keywords

DELTA

Set this keyword to a three-element vector or a scalar that specifies the grid spacing in the X, Y, and Z dimensions. The default spacing produces NGRID samples within the range of each axis.

DTOL

The tolerance for detecting an ill-conditioned system of equations. The default value is 0.01, which is appropriate for small ranges of X, Y, and Z. For large ranges of X, Y, or Z, it may be necessary to decrease the value of DTOL. If you receive the error message “GRID3: Ill-conditioned matrix or all nodes co-planar,” try decreasing the value of DTOL.

GRID

This keyword specifies the interpretation of G_x , G_y , and G_z . The default value for GRID is zero if G_x , G_y , and G_z are supplied, otherwise a regularly-gridded volume is produced.

NGRID

The number of samples along each axis. NGRID can be set to a scalar, in which case each axis has the same number of samples, or to a three-element array containing the number of samples for each axis. The default value for NGRID is 25.

START

A three-element array that specifies the starting value for each grid. The default value for START is the minimum value in the respective X, Y, and Z array.

Examples

Produce a set random points within the (0,1) unit cube and simulate a function:

```

; Number of irregular samples:
N = 300

; Generate random values between 0 and 1:
X = RANDOMU(SEED, N)
Y = RANDOMU(SEED, N)
Z = RANDOMU(SEED, N)

; The function to simulate:
F = (X-.5)^2 + (Y-.5)^2 + Z

; Return a cube with 25 equal-interval samples along each axis:
Result = GRID3(X, Y, Z, F)

; Return a cube with 11 elements along each dimension, which
; samples each axis at (0, 0.1, ..., 1.0):
Result = GRID3(X, Y, Z, F, START=[0., 0., 0], $
              DELTA=0.1, NGRID=10)

```

The same result is produced by the statements:

```

; Create sample values:
S = FINDGEN(11) / 10.
Result = GRID3(X, Y, Z, F, S, S, S, /GRID)

```

Version History

Introduced: Pre 4.0

See Also

[SLICER3](#), [QGRID3](#), [QHULL](#)

GRIDDATA

The GRIDDATA function interpolates scattered data values on a plane or a sphere to a regular grid, an irregular grid, a specified set of interpolates, or scattered data points. This is accomplished using one of several available methods. The function result is a two-dimensional floating point array. Computations are performed in single precision floating point. Interpolation methods supported by this function are as follows:

- Inverse Distance (default)
- Natural Neighbor
- Kriging
- Nearest Neighbor
- Linear
- Polynomial Regression
- Minimum Curvature
- Quintic
- Modified Shepard's
- Radial Basis Function

Note

GRIDDATA accepts complex input but only utilizes the real part of any complex number.

Syntax

Interleaved

Result = GRIDDATA(*X*, *F*)

Planar

Result = GRIDDATA(*X*, *Y*, *F*)

Sphere From Cartesian Coordinates

Result = GRIDDATA(*X*, *Y*, *Z*, *F*, /SPHERE)

Sphere From Spherical Coordinates

Result = GRIDDATA(*Lon*, *Lat*, *F*, /SPHERE)

Inverse Distance Keywords:

[, [METHOD](#)='InverseDistance' | /INVERSE_DISTANCE]
 [, [ANISOTROPY](#)=vector] [, /DEGREES] [, [DELTA](#)=vector]
 [, [DIMENSION](#)=vector] [, [TRIANGLES](#)=array] [, [EMPTY_SECTORS](#)=value]

```
[, MAX_PER_SECTOR=value ] [, MIN_POINTS=value ]
[, SEARCH_ELLIPSE=vector ] [, FAULT_POLYGONS=vector ]
[, FAULT_XY=array ] [, /GRID, XOUT=vector, YOUT=vector ]
[, MISSING=value ] [, POWER=value ]
[, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 } ] [, SMOOTHING=value ] [, /SPHERE]
[, START=vector ]
```

Kriging Keywords: METHOD='Kriging' | /KRIGING [, ANISOTROPY=vector]
 [, DELTA=vector] [, DIMENSION=vector]
 [, TRIANGLES=array [, EMPTY_SECTORS=value]
 [, MAX_PER_SECTOR=value] [, MIN_POINTS=value]
 [, SEARCH_ELLIPSE=vector] [, FAULT_POLYGONS=vector]
 [, FAULT_XY=array] [, /GRID, XOUT=vector, YOUT=vector]
 [, MISSING=value] [, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 }] [, /SPHERE]
 [, START=vector] [, VARIOGRAM=vector]

Linear Interpolation Keywords:

```
METHOD='Linear' | /LINEAR , TRIANGLES=array [, DELTA=vector ]
[, DIMENSION=vector ] [, /GRID, XOUT=vector, YOUT=vector ]
[, MISSING=value ] [, START=vector ]
```

Minimum Curvature Keywords:

```
METHOD='MinimumCurvature' | /MIN_CURVATURE [, DELTA=vector ]
[, DIMENSION=vector ] [, START=vector ]
```

Modified Shepard's Keywords: METHOD='ModifiedShepards' | /SHEPARDS,

```
TRIANGLES=array [, ANISOTROPY=vector ] [, DELTA=vector ]
[, DIMENSION=vector ] [, EMPTY_SECTORS=value ]
[, FAULT_POLYGONS=vector ] [, FAULT_XY=array ] [, /GRID,
XOUT=vector, YOUT=vector ] [, MAX_PER_SECTOR=value ]
[, MIN_POINTS=value ] [, MISSING=value ] [, NEIGHBORHOOD=array ]
[, SEARCH_ELLIPSE=vector ] [, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 } ]
[, START=vector ]
```

Natural Neighbor Keywords:

```
METHOD='NaturalNeighbor' | /NATURAL_NEIGHBOR, TRIANGLES=array
[, /DEGREES ] [, DELTA=vector ] [, DIMENSION=vector ]
[, /GRID, XOUT=vector, YOUT=vector ] [, MISSING=value ]
[, /SPHERE] [, START=vector ]
```

Nearest Neighbor Keywords:

```
METHOD='NearestNeighbor' | /NEAREST_NEIGHBOR, TRIANGLES=array
[, /DEGREES ] [, DELTA=vector ] [, DIMENSION=vector ]
[, FAULT_POLYGONS=vector ] [, FAULT_XY=array ] [, /GRID,
```

`XOUT=vector, YOUT=vector] [, MISSING=value]`
`[, /SPHERE] [, START=vector]`

Polynomial Regression Keywords:

`METHOD='PolynomialRegression' | /POLYNOMIAL_REGRESSION,`
`[, DELTA=vector] [, DIMENSION=vector]`
`[, TRIANGLES=array [, EMPTY_SECTORS=value]`
`[, MAX_PER_SECTOR=value] [, MIN_POINTS=value]`
`[, SEARCH_ELLIPSE=vector] [, FAULT_POLYGONS=vector]`
`[, FAULT_XY=array] [, /GRID, XOUT=vector, YOUT=vector]`
`[, MISSING=value] [, POWER=value]`
`[, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 }] [, START=vector]`

Quintic Keywords: `METHOD='Quintic' | /QUINTIC, TRIANGLES=array`

`[, DELTA=vector] [, DIMENSION=vector] [, MISSING=value]`
`[, START=vector]`

Radial Basis Function Keywords:

`METHOD='RadialBasisFunction' | /RADIAL_BASIS_FUNCTION,`
`[, ANISOTROPY=vector] [, /DEGREES] [, DELTA=vector]`
`[, DIMENSION=vector] [, TRIANGLES=array [, EMPTY_SECTORS=value]`
`[, MAX_PER_SECTOR=value] [, MIN_POINTS=value]`
`[, SEARCH_ELLIPSE=vector] [, FAULT_POLYGONS=vector]`
`[, FAULT_XY=array] [, FUNCTION_TYPE={ 0 | 1 | 2 | 3 | 4 }]`
`[, /GRID, XOUT=vector, YOUT=vector] [, MISSING=value]`
`[, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 }] [, SMOOTHING=value] [, /SPHERE]`
`[, START=vector]`

Return Value

Result is a one- or two-dimensional floating point array. Computations are preformed in single precision floating point.

Arguments

X [, Y [, Z]]

The point locations. If only one input coordinate parameter is supplied, the points are interleaved; for the Cartesian coordinate system the points are 2-by-*n* dimensions; and 3-by-*n* for a sphere in Cartesian coordinates.

F

The function value at each location in the form of an *n*-element vector.

Note

GRIDDATA will use the minimum number of points specified in any of the X, Y, Z, or F array arguments as the number of input points and function values.

Lon, Lat

These arguments contain the locations (on a sphere) of the data points (similar to X, and Y) but are in degrees or radians (default) depending on the use of the keyword DEGREES.

Keywords**ANISOTROPY**

This keyword is a vector describing an ellipse (see the description for the SEARCH_ELLIPSE keyword). All points on the circumference of the ellipse have an equal influence on a point at the center of the ellipse.

For example, assume that atmospheric data are being interpolated, with one dimension being altitude, and the other dimension representing distance from a point. If the vertical mixing is half that of the horizontal mixing, a point 100 units from an interpolate and at the same level has the same influence as a point 50 units above or below the interpolate at the same horizontal location. This effect requires setting the ANISOTROPY keyword to [2, 1, 0] which forms an ellipse with an X-axis length twice as long as its Y-axis length.

DEGREES

By default, all angular inputs and keywords are assumed to be in radian units. Set the DEGREES keyword to change the angular input units to degrees.

DELTA

A two-element array specifying the grid spacing in X and Y.

If this keyword is not specified, or if one of the elements is set equal to zero, the grid spacing is determined from the values of the DIMENSION and START keywords, according to the following rules:

- $DELTA_0 = (\max(x) - START_0) / (DIMENSION_0 - 1)$
- $DELTA_1 = (\max(y) - START_1) / (DIMENSION_1 - 1)$

DELTA can also be set to a scalar value to be used for the grid size in both X and Y.

This keyword is ignored if the GRID, XOUT and YOUT keywords are specified.

DIMENSION

A two element array specifying the grid dimensions in X and Y . Default value is 25 for each dimension. This keyword can also be set to a scalar value to be used for the grid spacing in both X and Y .

This keyword is ignored if the GRID, XOUT and YOUT keywords are specified.

EMPTY_SECTORS

This keyword defines the search rules for the maximum number of sectors that may be empty when interpolating at each point. If this number or more sectors contain no data points, considering the search ellipse and/or the fault polygons, the resulting interpolant is the missing data value.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

FAULT_POLYGONS

Set this keyword to an array containing one or more polygon descriptions. A polygon description is an integer or longword array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0 \dots i_{n-1}$ are indices into the FAULT_XY vertices. The FAULT_POLYGON array may contain multiple polygon descriptions that have been concatenated. To have this keyword ignore an entry in the FAULT_POLYGONS array, set the vertex count, n , and all associated indices to 0. To end the drawing list, even if additional array space is available, set n to -1 . If this keyword is not specified, a single connected polygon is generated from FAULT_XY.

Note

FAULT_POLYGONS are not supported with spherical gridding.

FAULT_XY

The a 2-by- n array specifying the coordinates of points on the fault lines/polygons.

Note

FAULT_XY is not supported with spherical gridding.

FUNCTION_TYPE

Note

This keyword is only used with the Radial Basis Function method of interpolation.

Set this keyword to one of the values shown in the following table to indicate which basis function to use. Default is 0, the Inverse Multiquadric function.

Value	Function Type Used	Equation
0	Inverse Multiquadric	$B(h) = 1/(\sqrt{h^2 + R^2})$
1	Multilog	$B(h) = \log(h^2 + R^2)$
2	Multiquadric	$B(h) = \sqrt{h^2 + R^2}$
3	Natural Cubic Spline	$B(h) = (h^2 + R^2)^{3/2}$
4	Thin Plate Spline	$B(h) = (h^2 + R^2)\log(h^2 + R^2)$

Note - In the equations, h = the anisotropically scaled distance from the interpolant to the node, and R^2 = the value of the SMOOTHING keyword.

GRID

The GRID keyword controls how the XOUT and YOUT vectors specify where interpolates are desired.

If GRID is set, XOUT and YOUT must also be specified. Interpolation is performed on a regular or irregular grid specified by the vectors XOUT with m elements and YOUT with n elements. The *Result* is an m -by- n grid with point $[i, j]$ resulting from the interpolation at $(XOUT[i], YOUT[j])$. When XOUT and YOUT are used, the DELTA, DIMENSION and START keywords are ignored.

INVERSE_DISTANCE

Selects the Inverse Distance method of interpolation.

KRIGING

Selects the Kriging method of interpolation. The variogram type for the Kriging method is set by default, however the VARIOGRAM keyword can be used to set variogram parameters.

LINEAR

Selects the Linear method of interpolation. The TRIANGLES keyword is required when the LINEAR keyword is used.

MAX_PER_SECTOR

This keyword defines the search rules for the maximum number of data points to include in each sector when interpolating. Search rules effectively limit the number of data points used in computing each interpolate. For example, to use the nearest n nodes to compute each interpolant, specify `MAX_PER_SECTOR = n` and use the TRIANGLES keyword.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

METHOD

A string containing one of the method names as shown in the following table. The default for METHOD is “InverseDistance”.

Note

The interpolation method can be chosen using the METHOD keyword set to the specific string, or by setting the corresponding method name keyword.

Note

There are no spaces between words in the method strings and the strings are case insensitive.

Method String	Meaning
"InverseDistance"	Data points closer to the grid points have more effect than those which are further away.
"Kriging"	Data points and their spatial variance are used to determine trends which are applied to the grid points.
"Linear"	Grid points are linearly interpolated from triangles formed by Delaunay triangulation.
"MinimumCurvature"	A plane of grid points is conformed to the data points while trying to minimize the amount of bending in the plane.
"ModifiedShepards"	Inverse Distance weighted with the least squares method.
"NaturalNeighbor"	Each interpolant is a linear combination of the three vertices of its enclosing Delaunay triangle and their adjacent vertices.
"NearestNeighbor"	The grid points have the same value as the nearest data point.
"PolynomialRegression"	Each interpolant is a least-squares fit of a polynomial in X and Y of the specified power to the specified data points.
"Quintic"	Grid points are interpolated with quintic polynomials from triangles formed by Delaunay triangulation.
"RadialBasisFunction"	The effects of data points are weighted by a function of their radial distance from a grid point.

Table 3-43: METHOD Keyword Values

MIN_CURVATURE

Selects the Minimum Curvature method of interpolation.

Note

If the Cholesky factorization used within the Minimum Curvature method fails, then a scalar -1 will be returned instead of the two-dimensional array.

MIN_POINTS

If fewer than this number of data points are encountered in all sectors, the value of the resulting grid point is set to the value of the MISSING keyword.

The MIN_POINTS keyword also indicates the number of closest points used for each local fit, if SEARCH_ELLIPSE isn't specified.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

MISSING

Set this keyword to the value to use for missing data values. Default is 0.

NATURAL_NEIGHBOR

Selects the Natural Neighbor method of interpolation.

Note

The TRIANGLES keyword is required when the NATURAL_NEIGHBOR keyword is used.

NEAREST_NEIGHBOR

Selects the Nearest Neighbor method of interpolation.

Note

The TRIANGLES keyword is required when the NEAREST_NEIGHBOR keyword is used.

NEIGHBORHOOD

Note

The NEIGHBORHOOD keyword is only used for the Modified Shepard's method of interpolation.

A two-element array, $[Nq, Nw]$ defining the quadratic fit, Nq , and weighting, Nw , neighborhood sizes for the Modified Shepard's method. The default for Nq is the smaller of 13 and the number of points minus 1, with a minimum of 5. The default for Nw is the smaller of 19 and the number of points. The Modified Shepard's method

first computes the coefficients of a quadratic fit for each input point, using its Nq closest neighbors.

When interpolating an output point, the quadratic fits from the Nw closest input points are weighted inversely by a function of distance and then combined. The size of the neighborhood used for Shepard's method interpolation may also be specified by the search rules keywords.

POLYNOMIAL_REGRESSION

Selects the Polynomial Regression method for interpolation. The power of the polynomial regression is set to 2 by default, however the POWER keyword can be used to change the power to 1 or 3.

The function fit to each interpolant corresponding to the POWER keyword set equal to 1, 2 (the default), and 3 respectively is as follows:

$$F(x,y) = a_0 + a_1x + a_2y$$

$$F(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4y^2 + a_5xy$$

$$F(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4y^2 + a_5xy + a_6x^3 + a_7y^3 + a_8x^2y + a_9xy^2$$

By inspection, a minimum of three data points are required to fit the linear polynomial, six data points for the second polynomial equation (where POWER = 2), and ten data points for the third polynomial (POWER = 3). If not enough data points exist for a given interpolant, the missing data values are set to the value of the MISSING keyword.

POWER

The weighting power of the distance, or the maximum order in the polynomial fitting function. For polynomial regression, this value is either 1, 2 (the default), or 3.

Note

The POWER keyword is only used for the Inverse Distance and Polynomial Regression methods of interpolation.

QUINTIC

Selects the triangulation with Quintic interpolation method.

Note

The TRIANGLES keyword is required when the QUINTIC keyword is used.

RADIAL_BASIS_FUNCTION

Selects the Radial Basis Function method of interpolation.

SEARCH_ELLIPSE

This keyword defines the search rules as a scalar or vector of from 1 to 3 elements that specify an ellipse or circle in the form $[R1]$, $[R1, R2]$, or $[R1, R2, Theta]$. $R1$ is one radius, $R2$ the other radius, and $Theta$ describes the angle between the X -axis to the $R1$ -axis, counterclockwise, in degrees or radians as specified by the DEGREES keyword. Only data points within this ellipse, centered on the location of the interpolate, are considered. If not specified, or 0, this distance test is not applied. Search rules effectively limit the number of data points used in computing each interpolate.

For example, to only consider data points within a distance of 5 units of each interpolant, specify the keyword as `SEARCH_ELLIPSE = 5`.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

SECTORS

This keyword defines the search rules for the number of sectors used in applying the MAX_SECTOR, EMPTY_SECTORS, and MIN_POINTS tests, an integer from 1 (the default setting) to 8.

SHEPARDS

Selects the Modified Shepard's method of interpolation. The parameters for the Modified Shepard's method are set by default, however the NEIGHBORHOOD keyword can be used to modify the parameters.

Note

The TRIANGLES keyword is required when the SHEPARDS keyword is used.

SMOOTHING

A scalar value defining the smoothing radius. For the Radial Basis Function method, if SMOOTHING is not specified, the default value is equal to the average point spacing, assuming a uniform distribution. For the Inverse Distance method, the default value is 0, implying no smoothing.

Note

The SMOOTHING keyword is used only for the Inverse Distance and Radial Basis Function methods of interpolation.

SPHERE

If set, data points lie on the surface of a sphere.

When the SPHERE keyword is used with Cartesian input coordinates (x,y,z), the coordinates must lie on a unit sphere (i.e., center at 0,0,0 and radius = 1). If any points do not lie on the sphere (within some tolerance), they are forced to be on the sphere, and GRIDDATA issues a warning and continues execution.

Furthermore, if a point lies at the origin, GRIDDATA throws an error that Cartesian coordinates cannot be zero, and halts execution.

START

A scalar or a two-element array specifying the start of the grid in X and Y. Default value is [min(x), min(y)].

This keyword is ignored if the GRID, XOUT and YOUT keywords are specified.

TRIANGLES

A 3-by-*nt* longword array describing the connectivity of the input points, as returned by TRIANGULATE, where *nt* is the number of triangles. If duplicate point locations are input and the TRIANGLES keyword is present, only one of the points is considered.

Note

The TRIANGLES keyword is required for the Natural Neighbor, Nearest Neighbor, Modified Shepard's, Linear, and Quintic Interpolation methods.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

VARIOGRAM

Specifies the variogram type and parameters for the Kriging method. This parameter is a vector of one to four elements in the form of: [*Type*, *Range*, *Nugget*, *Scale*]. The *Type* is encoded as: 1 for linear, 2 for exponential, 3 for gaussian, 4 for spherical.

Defaults values are: *Type* is exponential, *Range* is 8 times the average point spacing assuming a uniform distribution, *Nugget* is zero, and *Scale* is 1.

Note

The VARIOGRAM keyword is only used with the Kriging method of interpolation.

The following functions are used to model the variogram functions:

Linear Covariance:

$$\text{Covariance} = \begin{array}{ll} N & \text{if } d = 0 \\ Sd & \text{if } d < R \\ 0 & \text{if } d > R \end{array}$$

Exponential Covariance:

$$\text{Covariance} = \begin{array}{ll} N + S & \text{if } d = 0 \\ Se^{(-3d/R)} & \text{if } d > 0 \end{array}$$

Gaussian Covariance:

$$\text{Covariance} = \begin{array}{ll} N + S & \text{if } d = 0 \\ Se^{(-3d^2/R^2)} & \text{if } d > 0 \end{array}$$

Spherical Covariance:

$$\text{Covariance} = \begin{array}{ll} N + S & \text{if } d = 0 \\ S - 1.5S(d/R) + 0.5(d/R)^3 & \text{if } d < R \\ 0 & \text{if } d > R \end{array}$$

where d is the distance from one point to another, R is the range value, N is the nugget value, and S is the scale value.

XOUT

If the GRID keyword is set, use XOUT to specify irregularly spaced rectangular output grids. If XOUT is specified, YOUT must also be specified. When XOUT and YOUT are used, the DELTA, DIMENSION and START keywords are ignored.

If GRID is not set (the default), the location vectors XOUT and YOUT directly contain the X and Y values of the interpolates, and must have the same number of elements. The *Result* has the same structure and number of elements as XOUT and YOUT, with point $[i]$ resulting from the interpolation at (XOUT $[i]$, YOUT $[i]$).

YOUT

If the GRID keyword is set, use YOUT to specify irregularly spaced rectangular output grids. If YOUT is specified, XOUT must also be specified. When XOUT and YOUT are used, the DELTA, DIMENSION and START keywords are ignored.

If GRID is not set (the default), the location vectors XOUT and YOUT directly contain the X and Y values of the interpolates, and must have the same number of elements. The *Result* has the same structure and number of elements as XOUT and YOUT, with point $[i]$ resulting from the interpolation at (XOUT $[i]$, YOUT $[i]$).

Examples

Example 1

This example interpolates a data set measured on an irregular grid. Various types of the Inverse Distance interpolation method (the default method) are used in this example.

```

; Create a dataset of N points.
n = 100                                ;# of scattered points
seed = -121147L                        ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 768, TITLE = 'Inverse Distance'
!P.MULTI = [0, 1, 3, 0, 0]

```

```

; Inverse distance: Simplest default case which produces a 25 x
; 25 grid.
grid = GRIDDATA(x, y, f)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Simple Example'

; Default case, Inverse distance.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Larger Grid'

; Inverse distance + smoothing.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    SMOOTH = 0.05)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Smoothing'

; Set system variable back to default value.
!P.MULTI = 0

```

Example 2

This example uses the same data as the previous one, however in this example we use the Radial Basis Function and the Modified Shepard's interpolation methods.

```

; Create a dataset of N points.
n = 100 ;# of scattered points
seed = -121147L ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form of a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 512, $
    TITLE = 'Different Methods of Gridding'
!P.MULTI = [0, 1, 2, 0, 0]

; Use radial basis function with multilog basis function.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    /RADIAL_BASIS_FUNCTION, FUNCTION_TYPE = 1)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Radial Basis Function'

; The following example requires triangulation.
TRIANGULATE, x, y, tr

```

```

; Use Modified Shepard's method.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /SHEPARDS)
SURFACE, grid, CHARSIZE = 3, TITLE = "Modified Shepard's Method"

; Set system variable back to default value.
!P.MULTI = 0

```

Example 3

This example uses the same data as the previous ones, however in this example we use various types of the Polynomial Regression interpolation method.

```

; Create a dataset of N points.
n = 100                                ;# of scattered points
seed = -121147L                         ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 768, $
    TITLE = 'Polynomial Regression'
!P.MULTI = [0, 1, 3, 0, 0]

; The following examples require the triangulation.
TRIANGULATE, x, y, tr

; Fit with a 2nd degree polynomial in x and y. This fit considers
; all points when fitting the surface, obliterating the individual
; peaks.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /POLYNOMIAL_REGRESSION)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Global Degree 2 Polynomial'

; Fit with a 2nd degree polynomial in x and y, but this time use
; only the 10 closest nodes to each interpolant. This provides a
; relatively smooth surface, but still shows the individual peaks.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /POLYNOMIAL_REGRESSION, MAX_PER_SECTOR = 10)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Local Polynomial, 10 Point'

; As above, but use only the nodes within a distance of 0.4 when

```

```

; fitting each interpolant.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /POLYNOMIAL_REGRESSION, SEARCH_ELLIPSE = 0.4)
SURFACE, grid, CHARSIZE = 3, $
    TITLE = 'Local Polynomial, Radius = 0.4'

!P.MULTI = 0          ; Set system variable back to default value.

```

Example 4

This example uses the same data as the previous ones; however, in this example we show how to speed up the interpolation by limiting the interpolation to the local area around each interpolate.

```

; Create a dataset of N points.\.
n = 100                      ;# of scattered points
seed = -121147L              ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Note: the inverse distance, kriging, polynomial regression, and
; radial basis function methods are, by default, global methods in
; which each input node affects each output node. With these
; methods, large datasets can require a prohibitively long time to
; compute unless the scope of the interpolation is limited to a
; local area around each interpolate by specifying search rules.
; In fact, the radial basis function requires time proportional to
; the cube of the number of input points.

; For example, with 2,000 input points, a typical workstation
; required 500 seconds to interpolate a 10,000 point grid using
; radial basis functions. By limiting the size of the fit to the
; 20 closest points to each interpolate, via the MIN_POINTS
; keyword, the time required dropped to less than a second.

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 512, $
    TITLE = 'Radial Basis Function'
!P.MULTI = [0, 1, 2, 0, 0]

; Slow way:
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $

```

```

/RADIAL_BASIS_FUNCTION)
SURFACE, grid, CHARSIZE = 3, TITLE = 'All Points'

; The following example requires triangulation.
TRIANGULATE, x, y, tr

; Faster way:
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
/RADIAL_BASIS_FUNCTION, MIN_POINTS = 15, TRIANGLES = tr)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Nearest 15 Points'

; Set system variable back to default value.
!P.MULTI = 0

```

Example 5

This example interpolates a spherical data set measured on an irregular grid. We use the Kriging and Natural Neighbors interpolation methods in this example.

```

; Create a 100 scattered points on a sphere and form a function
; of their latitude and longitude. Then grid them to a 2 degree
; grid over the sphere, display a Mollweide projection map, and
; overlay the contours of the result on the map.

; Create a dataset of N points.
n = 100
; A 2 degree grid with grid dimensions.
delta = 2
dims = [360, 180]/delta
; Longitude and latitudes (for uniformly placed data)
lon = RANDOMU(seed, n) * 360 - 180
lat = ACOS(2*RANDOMU(seed, n) - 1.) * !RADEG - 90
; The lon/lat grid locations
lon_grid = FINDGEN(dims[0]) * delta - 180
lat_grid = FINDGEN(dims[1]) * delta - 90

; Create a dependent variable in the form of a smoothly varying
; function.
f = SIN(2*lon*!DTOR) + COS(lat*!DTOR) ;

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 768, TITLE = 'Spherical Gridding'
!P.MULTI = [0, 1, 3, 0, 0]

; Kriging: Simplest default case.
z = GRIDDATA(lon, lat, f, /KRIGING, /DEGREES, START = 0, /SPHERE, $
DELTA = delta, DIMENSION = dims)
MAP_SET, /MOLLWEIDE, /ISOTROPIC, /HORIZON, /GRID, CHARSIZE = 3, $
TITLE = 'Sphere: Kriging'

```

```

CONTOUR, z, lon_grid, lat_grid, /OVERPLOT, NLEVELS = 10, /FOLLOW

; This example is the same as above, but with the addition of a
; call to QHULL to triangulate the points on the sphere, and to
; then interpolate using the 10 closest points. The gridding
; portion of this example requires about one-fourth the time as
; above.
QHULL, lon, lat, tr, /DELAUNAY, SPHERE = s
z = GRIDDATA(lon, lat, f, /DEGREES, START = 0, DELTA = delta, $
  DIMENSION = dims, TRIANGLES = tr, MIN_POINTS = 10, /KRIGING, $
  /SPHERE)
MAP_SET, /MOLLWEIDE, /ISOTROPIC, /HORIZON, /GRID, /ADVANCE, $
  CHARSIZE = 3, TITLE = 'Sphere: Kriging, 10 Closest Points'
CONTOUR, z, lon_grid, lat_grid, /OVERPLOT, NLEVELS = 10, /FOLLOW

; This example uses the natural neighbor method, which is about
; four times faster than the above example but does not give as
; smooth a surface.
z = GRIDDATA(lon, lat, f, /DEGREES, START = 0, DELTA = delta, $
  DIMENSION = dims, /SPHERE, /NATURAL_NEIGHBOR, TRIANGLES = tr)
MAP_SET, /MOLLWEIDE, /ISOTROPIC, /HORIZON, /GRID, /ADVANCE, $
  CHARSIZE = 3, TITLE = 'Sphere: Natural Neighbor'
CONTOUR, z, lon_grid, lat_grid, /OVERPLOT, NLEVELS = 10, /FOLLOW

; Set system variable back to default value.
!P.MULTI = 0

```

Example 6

The following example uses the data from the `irreg_grid1.txt` ASCII file. This file contains scattered elevation data of a model of an inlet. This scattered elevation data contains two duplicate locations.

The `GRID_INPUT` procedure is used to omit the duplicate locations for the `GRIDDATA` function. The `GRIDDATA` function is then used to grid the data using the Radial Basis Function method. This method is specified by setting the `METHOD` keyword the `RadialBasisFunction` string, although it could easily be done using the `RADIAL_BASIS_FUNCTION` keyword.

```

; Import the Data:

; Determine the path to the file.
file = FILEPATH('irreg_grid1.txt', $
  SUBDIRECTORY = ['examples', 'data'])

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

```

```

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
data = dataArray[*, 2]

; Display the Data:

; Scale the data to range from 1 to 253 so a color table can be
; applied. The values of 0, 254, and 255 are reserved as outliers.
scaled = BYTSCL(data, TOP = !D.TABLE_SIZE - 4) + 1B

; Load the color table. If you are on a TrueColor, set the
; DECOMPOSED keyword to the DEVICE command before running a
; color table related routine.
DEVICE, DECOMPOSED = 0
LOADCT, 38

; Open a display window and plot the data points.
WINDOW, 0
PLOT, x, y, /XSTYLE, /YSTYLE, LINSTYLE = 1, $
    TITLE = 'Original Data, Scaled (1 to 253)', $
    XTITLE = 'x', YTITLE = 'y'

; Now display the data values with respect to the color table.
FOR i = 0L, (N_ELEMENTS(x) - 1) DO PLOTS, x[i], y[i], PSYM = -1, $
    SYMSIZE = 2., COLOR = scaled[i]

; Grid the Data and Display the Results:

; Preprocess and sort the data. GRID_INPUT will
; remove any duplicate locations.
GRID_INPUT, x, y, data, xSorted, ySorted, dataSorted

; Initialize the grid parameters.
gridSize = [51, 51]

; Use the equation of a straight line and the grid parameters to
; determine the x of the resulting grid.
slope = (MAX(xSorted) - MIN(xSorted))/(gridSize[0] - 1)
intercept = MIN(xSorted)
xGrid = (slope*FINDGEN(gridSize[0])) + intercept

; Use the equation of a straight line and the grid parameters to
; determine the y of the resulting grid.

```

```

slope = (MAX(ySorted) - MIN(ySorted))/(gridSize[1] - 1)
intercept = MIN(ySorted)
yGrid = (slope*FINDGEN(gridSize[1])) + intercept

; Grid the data with the Radial Basis Function method.
grid = GRIDDATA(xSorted, ySorted, dataSorted, $
    DIMENSION = gridSize, METHOD = 'RadialBasisFunction')

; Open a display window and contour the Radial Basis Function
; results.
WINDOW, 1
scaled = BYTSCL(grid, TOP = !D.TABLE_SIZE - 4) + 1B
CONTOUR, scaled, xGrid, YGrid, /XSTYLE, /YSTYLE, /FILL, $
    LEVELS = BYTSCL(INDGEN(18), TOP = !D.TABLE_SIZE - 4) + 1B, $
    C_COLORS = BYTSCL(INDGEN(18), TOP = !D.TABLE_SIZE - 4) + 1B, $
    TITLE = 'The Resulting Grid with Radial Basis Function', $
    XTITLE = 'x', YTITLE = 'y'

```

Example 7

The following example uses the data from the `irreg_grid1.txt` ASCII file. This file contains scattered elevation data of a model of an inlet. This scattered elevation data contains two duplicate locations. The same data is used in the previous example.

The `GRID_INPUT` procedure is used to omit the duplicate locations for the `GRIDDATA` function. The `GRIDDATA` function is then used to grid the data using the Radial Basis Function method. This method is specified by setting the `METHOD` keyword the `RadialBasisFunction` string, although it could easily be done using the `RADIAL_BASIS_FUNCTION` keyword.

Faulting is also applied in this example. First, a fault area is placed around the right side of the dataset. This fault area contains data points. The data points within this area are gridded separately from the points outside of the fault area.

Then, a fault area is defined within a region that does not contain any data points. Since this fault area does not contain any points, the grid within this region simply results to the value defined by the `MISSING` keyword. The points outside of the fault area are gridded independent of the fault region.

```

; Import the Data:

; Determine the path to the file.
file = FILEPATH('irreg_grid1.txt', $
    SUBDIRECTORY = ['examples', 'data'])

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

```

```

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
data = dataArray[*, 2]

; Grid the Data and Display the Results:

; Preprocess and sort the data. GRID_INPUT will
; remove any duplicate locations.
GRID_INPUT, x, y, data, xSorted, ySorted, dataSorted

; Initialize the grid parameters.
gridSize = [51, 51]

; Use the equation of a straight line and the grid parameters to
; determine the x of the resulting grid.
slope = (MAX(xSorted) - MIN(xSorted))/(gridSize[0] - 1)
intercept = MIN(xSorted)
xGrid = (slope*FINDGEN(gridSize[0])) + intercept

; Use the equation of a straight line and the grid parameters to
; determine the y of the resulting grid.
slope = (MAX(ySorted) - MIN(ySorted))/(gridSize[1] - 1)
intercept = MIN(ySorted)
yGrid = (slope*FINDGEN(gridSize[1])) + intercept

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 38
WINDOW, 0, XSIZE = 600, YSIZE = 600, $
    TITLE = 'The Resulting Grid from the Radial Basis Function '+ $
        'Method with Faulting'
!P.MULTI = [0, 1, 2, 0, 0]

; Define a fault area, which contains data points.
faultVertices = [[2200, 4000], [2200, 3000], [2600, 2700], $
    [2600, -50], [5050, -50], [5050, 4000], [2200, 4000]]
faultConnectivity = [7, 0, 1, 2, 3, 4, 5, 6, -1]

; Grid the data with faulting using the Radial Basis Function
; method.
grid = GRIDDATA(xSorted, ySorted, dataSorted, $
    DIMENSION = gridSize, METHOD = 'RadialBasisFunction', $
    FAULT_XY = faultVertices, FAULT_POLYGONS = faultConnectivity, $

```

```

MISSING = MIN(dataSorted))

; Display grid results.
CONTOUR, BYTSCL(grid), xGrid, YGrid, /XSTYLE, /YSTYLE, /FILL, $
  LEVELS = BYTSCL(INDGEN(18)), C_COLORS = BYTSCL(INDGEN(18)), $
  TITLE = 'Fault Area Contains Data ' + $
  '(Fault Area in Dashed Lines)', XTITLE = 'x', YTITLE = 'y'

; Display outline of fault area.
PLOTS, faultVertices, /DATA, LINESTYLE = 2, THICK = 2

; Define a fault area, which does not contain data points.
faultVertices = [[2600, -50], [2800, -50], [2800, 2700], $
  [2400, 3000], [2400, 4000], [2200, 4000], [2200, 3000], $
  [2600, 2700], [2600, -50]]
faultConnectivity = [9, 0, 1, 2, 3, 4, 5, 6, 7, 8, -1]

; Grid the data with faulting using the Radial Basis Function
; method.
grid = GRIDDATA(xSorted, ySorted, dataSorted, $
  DIMENSION = gridSize, METHOD = 'RadialBasisFunction', $
  FAULT_XY = faultVertices, FAULT_POLYGONS = faultConnectivity, $
  MISSING = MIN(dataSorted))

; Display grid results.
CONTOUR, BYTSCL(grid), xGrid, YGrid, /XSTYLE, /YSTYLE, /FILL, $
  LEVELS = BYTSCL(INDGEN(18)), C_COLORS = BYTSCL(INDGEN(18)), $
  TITLE = 'Fault Area Does Not Contain Data ' + $
  '(Fault Area in Dashed Lines)', XTITLE = 'x', YTITLE = 'y'

; Display outline of fault area.
PLOTS, faultVertices, /DATA, LINESTYLE = 2, THICK = 2

; Set system variable back to default value.
!P.MULTI = 0

```

References

Kriging

Isaaks, E. H., and Srivastava, R. M., *An Introduction to Applied Geostatistics*, Oxford University Press, New York, 1989.

Minimum Curvature

Barrodale, I., et al, "Warping Digital Images Using Thin Plate Splines", Pattern Recognition, Vol 26, No 2, pp. 375-376., 1993.

Powell, M.J.D., "Tabulation of thin plate splines on a very fine two-dimensional grid", Report No. DAMTP 1992/NA2, University of Cambridge, Cambridge, U.K. 1992.

Modified Shepard's

Franke, R., and Nielson, G. , "Smooth Interpolation of Large Sets of Scattered Data", International Journal for Numerical Methods in Engineering, v. 15, 1980, pp. 1691-1704.

Renka, R. J., Algorithm 790 - CSHEP2D: Cubic Shepard Method for Bivariate Interpolation of Scattered Data, Robert J. Renka, ACM Trans. Math Softw. 25, 1 (March 1999), pp. 70-73.

Shepard, D., "A Two Dimensional Interpolation Function for Irregularly Spaced Data", Proc. 23rd Nat. Conf. ACM, 1968, pp. 517-523.

Natural Neighbor

Watson, D. F., Contouring: A Guide to the Analysis and Display of Spatial Data, Pergamon Press, ISBN 0 08 040286 0, 1992.

Watson, D. F., Nngridr - An Implementation of Natural Neighbor Interpolation, David Watson, P.O. Box 734, Claremont, WA 6010, Australia, 1994.

Quintic

Akima, H., Algorithm 761 - Scattered-data Surface Fitting that has the Accuracy of a Cubic Polynomial, Hiroshi Akima, ACM Trans. Math. Softw. 22, 3 (Sep. 1996), pp. 362 - 371.

Renka, R.J., "A Triangle-based C1 Interpolation Method", Rocky Mountain Journal of Mathematics, Vol 14, No. 1, 1984.

Radial Basis Function

Franke, R., A Critical Comparison of Some Methods for Interpolation of Scattered Data, Naval Postgraduate School, Technical Report, NPS 53-79-003, 1979.

Hardy, R.L., "Theory and Applications of the Multiquadric-biharmonic Method", Computers Math. With Applic, v 19, no. 8/9, 1990, pp.163-208.

Version History

Introduced: 5.5

See Also

[GRID_INPUT](#)

GS_ITER

The GS_ITER function solves an n by n linear system of equations using Gauss-Seidel iteration with over- and under-relaxation to enhance convergence.

Note

The equations must be entered in *diagonally dominant* form to guarantee convergence. A system is diagonally dominant if the diagonal element in a given row is greater than the sum of the absolute values of the non-diagonal elements in that row.

This routine is written in the IDL language. Its source code can be found in the file `gs_iter.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = GS_ITER( A, B [, /CHECK] [, /DOUBLE] [, LAMBDA=value{0.0 to 2.0}]  
                [, MAX_ITER=value] [, TOL=value] [, X_0=vector] )
```

Return Value

Returns the solution to the linear system of equations of the specified dimensions.

Arguments

A

An n by n integer, single-, or double-precision floating-point array. On output, A is divided by its diagonal elements. Integer input values are converted to single-precision floating-point values.

B

A vector containing the right-hand side of the linear system $\mathbf{Ax}=\mathbf{b}$. On output, B is divided by the diagonal elements of A .

Keywords

CHECK

Set this keyword to check the array *A* for diagonal dominance. If *A* is not in diagonally dominant form, `GS_ITER` reports the fact but continues processing on the chance that the algorithm may converge.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

LAMBDA

A scalar value in the range: [0.0, 2.0]. This value determines the amount of *relaxation*. Relaxation is a weighting technique used to enhance convergence.

- If `LAMBDA = 1.0`, no weighting is used. This is the default.
- If $0.0 \leq \text{LAMBDA} < 1.0$, convergence improves in oscillatory and non-convergent systems.
- If $1.0 < \text{LAMBDA} \leq 2.0$, convergence improves in systems already known to converge.

MAX_ITER

The maximum allowed number of iterations. The default value is 30.

TOL

The relative error tolerance between current and past iterates calculated as: $|(\text{current-past})/\text{current}|$. The default is 1.0×10^{-4} .

X_0

An *n*-element vector that provides the algorithm's starting point. The default is [1.0, 1.0, ... , 1.0].

Example

```
; Define an array A:
A = [[ 1.0,  7.0, -4.0], $
      [ 4.0, -4.0,  9.0], $
      [12.0, -1.0,  3.0]]
```

```

; Define the right-hand side vector B:
B = [12.0, 2.0, -9.0]

; Compute the solution to the system:
RESULT = GS_ITER(A, B, /CHECK)

```

IDL prints:

```

Input matrix is not in Diagonally Dominant form.
Algorithm may not converge.
% GS_ITER: Algorithm failed to converge within given parameters.

```

Since the **A** represents a system of linear equations, we can reorder it into diagonally dominant form by rearranging the rows:

```

A = [[12.0, -1.0, 3.0], $
      [ 1.0, 7.0, -4.0], $
      [ 4.0, -4.0, 9.0]]

; Make corresponding changes in the ordering of B:
B = [-9.0, 12.0, 2.0]

; Compute the solution to the system:
RESULT = GS_ITER(A, B, /CHECK)

```

IDL prints:

```

-0.999982      2.99988      1.99994

```

Version History

Introduced: Pre 4.0

See Also

[CRAMER](#), [LU_COMPLEX](#), [CHOLSOL](#), [LUSOL](#), [SVSOL](#), [TRISOL](#)

H_EQ_CT

The `H_EQ_CT` procedure histogram-equalizes the color tables for an image or a region of the display. A pixel-distribution histogram is obtained, the cumulative integral is taken and scaled, and the result is applied to the current color table.

This routine is written in the IDL language. Its source code can be found in the file `h_eq_ct.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
H_EQ_CT [, Image]
```

Arguments

Image

A two-dimensional byte array representing the image whose histogram is to be used in determining the new color tables. If this value is omitted, the user is prompted to mark the diagonal corners of a region of the display. If *Image* is specified, it is assumed that the image is loaded into the current IDL window. *Image* must be scaled the same way as the image loaded to the display.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[H_EQ_INT](#)

H_EQ_INT

The H_EQ_INT procedure interactively histogram-equalizes the color tables of an image or a region of the display. By moving the cursor across the screen, the amount of histogram-equalization can be varied.

Either the image parameter or a region of the display marked by the user is used to obtain a pixel-distribution histogram. The cumulative integral is taken and scaled and the result is applied to the current color tables.

This routine is written in the IDL language. Its source code can be found in the file `h_eq_int.pro` in the `lib` subdirectory of the IDL distribution.

Using the H_EQ_INT Interface

A window is created and the histogram equalization function is plotted. A linear ramp is overplotted. Move the cursor from left to right to vary the amount of histogram equalization applied to the color tables from 0 to 100%. Press the right mouse button to exit.

Syntax

```
H_EQ_INT [, Image]
```

Arguments

Image

A two-dimensional byte array representing the image whose histogram is to be used in determining the new color tables. If this value is omitted, the user is prompted to mark the diagonal corners of a region of the display. If *Image* is specified, it is assumed that the image is loaded into the current IDL window. *Image* must be scaled the same way as the image loaded to the display.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[H_EQ_CT](#)

H5_* Routines

For information, see [Chapter 3, “Hierarchical Data Format - HDF5”](#) in the *IDL Scientific Data Formats* manual.

H5_BROWSER

The H5_BROWSER function presents a graphical user interface for viewing and reading HDF5 files. The browser provides a tree view of the HDF5 file or files, a data preview window, and an information window for the selected objects. The browser may be created as either a selection dialog with Open/Cancel buttons, or as a standalone browser that can import data to the IDL main program level.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5_BROWSER([*Files*] [, /DIALOG_READ])

Return Value

If the DIALOG_READ keyword is specified then the *Result* is a structure containing the selected group or dataset (as described in the H5_PARSE function), or a zero if the Cancel button was pressed. If the DIALOG_READ keyword is not specified then the *Result* is the widget ID of the HDF5 browser.

Arguments

Files

An optional scalar string or string array giving the name of the files to initially open. Additional files may be opened interactively. If *Files* is not provided then the user is automatically presented with a File Open dialog upon startup.

Keywords

DIALOG_READ

If this keyword is set then the HDF5 browser is created as a modal Open/Cancel dialog instead of a standalone GUI. In this case, the IDL command line is blocked, and no further input is taken until the Open or Cancel button is pressed. If the GROUP_LEADER keyword is specified, then that widget ID is used as the group leader, otherwise a default group leader base is created.

All keywords to WIDGET_BASE, such as GROUP_LEADER and TITLE, are passed on to the top-level base.

Graphical User Interface Options

Open HDF5 file

Click on this button to bring up a file selection dialog. Multiple files may be selected for parsing. All selected files are added to the tree view.

Show preview

If this toggle button is selected, then the data within datasets will be shown in the preview window. One-dimensional datasets will be shown as line plots. Two-dimensional datasets will be shown as images, along with any provided image palettes. For three or higher-dimensional datasets, a two dimensional slice will be shown.

Fit in window

If this toggle button is selected, then the preview image will be scaled larger or smaller to fit within the preview window. The aspect ratio of the image will be unchanged.

Flip vertical

If this toggle button is selected, then the preview image will be flipped from top to bottom.

Flip horizontal

If this toggle button is selected, then the preview image will be flipped from left to right.

Note

If the DIALOG_READ keyword is present then the following options are available:

Open

Click on this button to close the HDF5 browser, and return an IDL structure containing the selected group or dataset, as described in the H5_PARSE function.

Cancel

Click on this button to close the HDF5 browser, and return a scalar zero for the result.

Note

If the `DIALOG_READ` keyword is not present then the following options are available:

Variable name for import

Set this text string to the name of the IDL variable to construct when importing HDF5 data to IDL structures. If the entered name is not a valid IDL identifier, then a valid identifier will be constructed by converting all non-alphanumeric characters to underscores.

Include data

If this toggle button is selected, then all data within the selected datasets will be read in from the HDF5 file and included in the IDL structure.

Import to IDL

Click on this button to import the currently selected HDF5 object into the IDL main program level. Imported variables will consist of a nested hierarchy of IDL structures, as described in the `H5_PARSE` function.

Done

Click on this button to close the HDF5 browser.

Example

The following example starts up the HDF5 browser on a sample file:

```
File = FILEPATH('hdf5_test.h5', SUBDIR=['examples','data'])
Result = H5_BROWSER(File)
```

Version History

Introduced 5.6

See Also

[H5_PARSE](#)

HANNING

The HANNING function is used to create a “window” for Fourier Transform filtering. It can be used to create both Hanning and Hamming windows.

This routine is written in the IDL language. Its source code can be found in the file `hanning.pro` in the `lib` subdirectory of the IDL distribution.

The Hanning window is defined as:

$$w(k) = \alpha - (1-\alpha) \cdot \cos(2 \pi k / N), \quad k = 0, 1, \dots, N-1$$

where $\alpha=0.5$ for the Hanning, and $\alpha=0.54$ for the Hamming window.

Note

Because of the factor of $1/N$ (rather than $1/(N-1)$) in the above equation, the Hanning filter is not exactly symmetric, and does not go to zero at the last point. The factor of $1/N$ is chosen to give the best behavior for spectral estimation of discrete data.

Syntax

Result = HANNING(*N*₁ [, *N*₂] [, ALPHA=*value*{0.5 to 1.0}] [, /DOUBLE])

Return Value

If only *N*₁ is specified, this function returns an array of dimensions [*N*₁]. If both *N*₁ and *N*₂ are specified, this function returns an array of dimensions [*N*₁, *N*₂]. If any of the inputs are double-precision or if the DOUBLE keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

*N*₁

The number of columns in the resulting array.

*N*₂

The number of rows in the resulting array.

Keywords

ALPHA

Set this keyword equal to the width parameter of a generalized Hamming window. ALPHA must be in the range of 0.5 to 1.0. If ALPHA = 0.5 (the default) the function is called a “Hanning” window. If ALPHA = 0.54, the result is called a “Hamming” window.

DOUBLE

Set this keyword to force the computations to be done in double-precision arithmetic.

Examples

```

; Construct a time series with three sine waves.
n = 1024
dt = 0.02
w = 2*!DPI*dt*DINDGEN(n)
x = -0.3d + SIN(2.8d * w) + SIN(6.25d * w) + SIN(11.0d * w)
; Find the power spectrum with and without the Hanning filter.
han = HANNING(n, /DOUBLE)
powerHan = ABS(FFT(han*x))^2
powerUnfilt = ABS(FFT(x))^2
freq = FINDGEN(n)/(n*dt)
; Plot the results.
PLOT, freq, powerHan, /XLOG, /YLOG, $
      XRANGE=[1,1./(2*dt)], XSTYLE=1, $
      TITLE='Power spectrum with Hanning (solid) and without
(dashed)'
OPLOT, freq, powerUnfilt, LINESTYLE=2

```

Version History

Introduced: Original

See Also

[FFT](#)

HDF_* Routines

For information, see [Chapter 4, “Hierarchical Data Format”](#) in the *IDL Scientific Data Formats* manual.

HDF_BROWSER

The `HDF_BROWSER` function presents a graphical user interface (GUI) that allows the user to view the contents of a Hierarchical Data Format (HDF), HDF-EOS, or NetCDF file, and prepare a template for the extraction of HDF data and metadata into IDL. The output template is an IDL structure that may be used when reading HDF files with the `HDF_READ` routine. If you have several HDF files of identical form, the returned template from `HDF_BROWSER` may be reused to extract data from these files with `HDF_READ`. If you do not need a multi-use template, you may call `HDF_READ` directly.

Graphical User Interface Menu Options

The following options are available from the graphical user interface menus.

Pulldown Menu

The following table shows the options available with the pulldown menu.

Menu Selection	Description
HDF/NetCDF Summary	
DF24 (24-bit Images)	24-bit images and their attributes
DFR8 (8-bit Images)	8-bit images and their attributes
DFP (Palettes)	Image palettes
SD (Variables/Attributes)	Scientific Datasets and attributes
AN (Annotations)	Annotations
GR (Generic Raster)	Images
GR Global (File) Attributes	Image attributes
VGroups	Generic data groups
VData	Generic data and attributes
HDF-EOS Summary	
Point	EOS point data and attributes

Table 3-44: HDF_BROWSER Pulldown Menu Options

Menu Selection	Description
Swath	EOS swath data and attributes
Grid	EOS grid data and attributes

Table 3-44: HDF_BROWSER Pulldown Menu Options (Continued)

Preview Button

If you have selected an image, 2D data set, or 3xnxm data set from the pulldown menu, click on this button to view the image. If you have selected a data item that can be plotted in two dimensions, click on this button to view a 2D plot of the data (the default) or click on the “Surface” radio button to display a surface plot, click on the “Contour” radio button to display a contour plot, or click on the “Show3” radio button for an image, surface, and contour display. You can also select the “Fit to Window” checkbox to fit the image to the window.

Read Checkbox

Select this checkbox to extract the current data or metadata item from the HDF file.

Extract As

Specify a name for the extracted data or metadata item

Note

The Read Checkbox must be selected for the item to be extracted. Default names are generated for all data items, but may be changed at any time by the user.

Syntax

```
Template = HDF_BROWSER([Filename] [, CANCEL=variable]
[, GROUP=widget_id] [, PREFIX=string])
```

Return Value

Returns a template structure containing heap variable references, or 0 if no file was selected. The user is required to clean up the heap variable references when done with them.

Arguments

Filename

A string containing the name of an HDF file to browse. If *Filename* is not specified, a dialog allows you to choose a file.

Keywords

CANCEL

Set this keyword to a named variable that will contain the byte value 1 (one) if the user clicked the “Cancel” button or the byte value 0 (zero) otherwise.

GROUP

Set this keyword to the widget ID of a widget that calls HDF_BROWSER. When this ID is specified, a death of the caller results in the death of the HDF_BROWSER. The following example demonstrates how to use the GROUP keyword to properly call HDF_BROWSER from within a widget application. To run this example, save the following code as `browser_example.pro`:

```

PRO BROWSER_EXAMPLE_EVENT, ev

    WIDGET_CONTROL, ev.id, GET_VALUE=val
    CASE val of
        'Browser': BEGIN
            a=HDF_BROWSER(GROUP=ev.top)
            HELP, a, /st
        END
        'Exit': WIDGET_CONTROL, ev.top, /DESTROY
    ENDCASE

END

PRO BROWSER_EXAMPLE

a=WIDGET_BASE(/ROW)
b=WIDGET_BUTTON(a, VALUE='Browser')
c=WIDGET_BUTTON(a, VALUE='Exit')
WIDGET_CONTROL, a, /REALIZE
XMANAGER, 'browser_example', a

END

```

PREFIX

When `HDF_BROWSER` reviews the contents of an HDF file, it creates default output names for the various data elements. By default these default names begin with a prefix derived from the filename. Set this keyword to a string value to be used in place of the default prefix.

Examples

```
template = HDF_BROWSER('test.hdf')
output_structure = HDF_READ(TEMPLATE=template)

or,

output_structure = HDF_READ('test.hdf', TEMPLATE=template)
```

Version History

Introduced: 5.1

See Also

[HDF_READ](#)

HDF_READ

The HDF_READ function allows extraction of Hierarchical Data Format (HDF), HDF-EOS, and NetCDF data and metadata into an output structure based upon information provided through a graphical user interface or through a file template generated by HDF_BROWSER. The output structure is a single level structure corresponding to the data elements and names specified by HDF_BROWSER or its output template. Templates generated by HDF_BROWSER may be re-used for HDF files of identical format.

Graphical User Interface Menu Options

The following options are available from the graphical user interface menus.

Pulldown Menu

The following table shows the options available with the pulldown menu.

Menu Selection	Description
HDF/NetCDF Summary	
DF24 (24-bit Images)	24-bit images and their attributes
DFR8 (8-bit Images)	8-bit images and their attributes
DFP (Palettes)	Image palettes
SD (Variables/Attributes)	Scientific Datasets and attributes
AN (Annotations)	Annotations
GR (Generic Raster)	Images
GR Global (File) Attributes	Image attributes
VGroups	Generic data groups
VData	Generic data and attributes
HDF-EOS Summary	
Point	EOS point data and attributes
Swath	EOS swath data and attributes

Table 3-45: HDF_BROWSER Pulldown Menu Options

Menu Selection	Description
Grid	EOS grid data and attributes

Table 3-45: HDF_BROWSER Pulldown Menu Options (Continued)

Preview Button

If you have selected an image, 2D data set, or 3xnxm data set from the pulldown menu, click on this button to view the image. If you have selected a data item that can be plotted in two dimensions, click on this button to view a 2D plot of the data (the default) or click on the “Surface” radio button to display a surface plot, click on the “Contour” radio button to display a contour plot, or click on the “Show3” radio button for an image, surface, and contour display. You can also select the “Fit to Window” checkbox to fit the image to the window.

Read Checkbox

Select this checkbox to extract the current data or metadata item from the HDF file.

Extract As

Specify a name for the extracted data or metadata item

Note

The Read Checkbox must be selected for the item to be extracted. Default names are generated for all data items, but may be changed at any time by the user.

Syntax

```
Result = HDF_READ( [Filename] [, DFR8=variable] [, DF24=variable]
  [, PREFIX=string] [, TEMPLATE =value] )
```

Return Value

Returns an output structure containing the specified Hierarchical Data Format (HDF), HDF-EOS, and NetCDF data and metadata.

Arguments

Filename

A string containing the name of a HDF file to extract data from. If Filename is not specified, a dialog allows you to specify a file. Note that if a template is specified, the template must match the HDF file selected.

Keywords

DFR8

Set this keyword to a named variable that will contain a $2 \times n$ string array of extracted DFR8 images and their palettes. The first column will contain the extracted DFR8 image names, while the second column will contain the extracted name of the associated palette. If no palette is associated with a DFR8 image the palette name will be set to the null string. If no DFR8 images were extracted from the HDF file, this returned string will be the null string array ["", ""].

DF24

Set this keyword to a named variable that will contain a string array of the names of all the extracted DF24 24-bit images. This is useful in determining whether a $(3, n, m)$ extracted data element is a 24-bit image or another type of data. If no DF24 24-bit images were extracted from the HDF file, the returned string will be the null string ("").

PREFIX

When HDF_READ is called without a template, it calls HDF_BROWSER to review the contents of an HDF file and create the default output names for the various data elements. By default, these names begin with a prefix derived from the filename. Set this keyword to a string value to be used in place of the default prefix.

TEMPLATE

Set this keyword to specify the HDF file template (generated by the function HDF_BROWSER), that defines which data elements to extract from the selected HDF file. Templates may be used on any files that have a format identical to the file the template was created from.

Examples

```
template = HDF_BROWSER('my.hdf')
output_structure = HDF_READ(TEMPLATE=template)

or,

output_structure = HDF_READ('my.hdf')

or,

;Select 'my.hdf' with the file locator
output_structure = HDF_READ()

or,

output_structure = HDF_READ('just_like_my.hdf', TEMPLATE=template)
```

Version History

Introduced: 5.1

See Also

[HDF_BROWSER](#)

HEAP_FREE

The `HEAP_FREE` procedure recursively frees all heap variables (pointers or objects) referenced by its input argument. This routine examines the input variable, including all array elements and structure fields. When a valid pointer or object reference is encountered, that heap variable is marked for removal, and then is recursively examined for additional heap variables to be freed. In this way, all heap variables that are referenced directly or indirectly by the input argument are located. Once all such heap variables are identified, `HEAP_FREE` releases them in a final pass. Pointers are released as if the `PTR_FREE` procedure was called. Objects are released as with a call to `OBJ_DESTROY`.

As with the related `HEAP_GC` procedure, there are some disadvantages to using `HEAP_FREE`:

- When freeing object heap variables, `HEAP_FREE` calls `OBJ_DESTROY` without supplying any plain or keyword arguments. Depending on the objects being released, this may not be sufficient. In such cases, call `OBJ_DESTROY` explicitly with the proper arguments rather than using `HEAP_FREE`.
- `HEAP_FREE` releases the referenced heap variables in an unspecified order which depends on the current state of the internal data structure used by IDL to hold them. This can be confusing for object destructor methods that expect all of their contained data to be present. If your application requires a specific order for the release of its heap variables, you must explicitly free them in the correct order. `HEAP_FREE` cannot be used in such cases.
- The algorithm used by `HEAP_FREE` to release variables requires examination of every existing heap variable (that is, it is an $O(n)$ algorithm). This may be slow if an IDL session has thousands of current heap variables.

For these reasons, RSI recommends that applications keep careful track of their heap variable usage, and explicitly free them at the proper time (for example, using the object destructor method) rather than resorting to simple-looking but potentially expensive expedients such as `HEAP_FREE` or `HEAP_GC`.

`HEAP_FREE` is recommended when:

- The data structures involved are highly complex, nested, or variable, and writing cleanup code is difficult and error prone.
- The data structures are opaque, and the code cleaning up does not have knowledge of the structure.

Syntax

```
HEAP_FREE, Var [, /OBJ] [, /PTR] [, /VERBOSE]
```

Arguments

Var

The variable whose data is used as the starting point for heap variables to be freed.

Keywords

OBJ

Set this keyword to free object heap variables only.

PTR

Set this keyword to free pointer heap variables only.

Note

Setting both the PTR and OBJ keywords is the same as setting neither.

VERBOSE

If this keyword is set, HEAP_FREE writes a one line description of each heap variable, in the format used by the HELP procedure, as the variable is released. This is a debugging aid that can be used by program developers to check for heap variable leaks that need to be located and eliminated.

Examples

```
; Create a structure variable.
mySubStructure = {pointer:PTR_NEW(INDGEN(100)), $
  obj:OBJ_NEW('Idl_Container')}
myStructure = {substruct:mySubStructure, $
  ptrs:[PTR_NEW(INDGEN(10)), PTR_NEW(INDGEN(11))]}

;Look at the heap.
HELP, /HEAP_VARIABLES

; Now free the heap variables contained in myStructure.
HEAP_FREE, myStructure, /VERBOSE
HELP, /HEAP_VARIABLES
```

Version History

Introduced: 5.3

See Also

[HEAP_GC](#)

HEAP_GC

The `HEAP_GC` procedure performs *garbage collection* on heap variables. It searches all current IDL variables (including common blocks, widget user values, etc.) for pointers and object references and determines which heap variables have become inaccessible. Pointer heap variables are freed (via `PTR_FREE`) and all memory used by the heap variable is released. Object heap variables are destroyed (via `OBJ_DESTROY`), also freeing all used memory.

The default action is to perform garbage collection on all heap variables regardless of type. Use the `POINTER` and `OBJECT` keywords to remove only specific types.

Note

Garbage collection is an expensive operation. When possible, applications should be written to avoid losing pointer and object references and avoid the need for garbage collection.

Warning

`HEAP_GC` uses a recursive algorithm to search for unreferenced heap variables. If `HEAP_GC` is used to manage certain data structures, such as large linked lists, a potentially large number of operations may be pushed onto the system stack. If so many operations are pushed that the stack runs out of room, IDL will crash.

Syntax

```
HEAP_GC [, /OBJ | , /PTR] [, /VERBOSE]
```

Arguments

None.

Keywords

OBJ

Set this keyword to perform garbage collection on object heap variables only.

PTR

Set this keyword to perform garbage collection on pointer heap variables only.

Note

Setting *both* the PTR and OBJ keywords is the same as setting neither.

VERBOSE

If this keyword is set, HEAP_GC writes a one line description of each heap variable, in the format used by the HELP procedure, as the variable is destroyed. This is a debugging aid that can be used by program developers to check for heap variable leaks that need to be located and eliminated.

Version History

Introduced: 5.0

See Also

[HEAP_FREE](#)

HELP

The HELP procedure gives the user information on many aspects of the current IDL session. The specific area for which help is desired is selected by specifying the appropriate keyword. If no arguments or keywords are specified, the default is to show the current nesting of procedures and functions, all current variables at the current program level, and open files. Only one keyword can be specified at a time.

Syntax

```
HELP, Expression1, ..., Expressionn [, /ALL_KEYS] [, /BREAKPOINTS] [, /BRIEF]
    [, /CALLS=variable] [, /DEVICE] [, /DLM] [, /FILES] [, /FULL]
    [, /FUNCTIONS] [, /HEAP_VARIABLES] [, /KEYS] [, /LAST_MESSAGE]
    [, /LEVEL=value] [, /MEMORY] [, /MESSAGES]
    [, /NAMES=string_of_variable_names] [, /OBJECTS] [, /OUTPUT=variable]
    [, /PATH_CACHE] [, /PROCEDURES] [, /RECALL_COMMANDS]
    [, /ROUTINES] [, /SHARED_MEMORY] [, /SOURCE_FILES]
    [, /STRUCTURES] [, /SYSTEM_VARIABLES] [, /TRACEBACK]
```

Arguments

Expression(s)

The arguments are interpreted differently depending on the keyword selected. If no keyword is selected, HELP displays basic information for its parameters. For example, to see the type and structure of the variable A, enter:

```
HELP, A
```

Keywords

Note that the use of some of the following keywords causes any arguments to HELP to be ignored and HELP provides other types of information instead. If the description of the keyword does not explicitly mention the arguments, the arguments are ignored.

ALL_KEYS

Set this keyword to show current function-key definitions as set by DEFINE_KEY. If no arguments are supplied, information on all function keys is displayed. If arguments are provided, they must be scalar strings containing the names of function keys, and information on the specified keys is given. Under UNIX, this keyword is

different from KEYS because every key is displayed, no matter what its current programming. Setting ALL_KEYS is equivalent to setting both KEYS and FULL. Under Windows, every key is always displayed; setting KEYS produces the same result as setting ALL_KEYS.

BREAKPOINTS

Set this keyword to display the breakpoint table which shows the program module and location for each breakpoint.

BRIEF

If set in conjunction with one of the following keywords, BRIEF produces very terse summary style output instead of the output normally displayed by those keywords:

- DLM
- HEAP_VARIABLES
- MESSAGES
- OBJECTS
- ROUTINES
- SOURCE_FILES
- STRUCTURES
- SYSTEM_VARIABLES

CALLS

Set this keyword to a named variable in which to store the procedure call stack. Each string element contains the name of the program module, source file name, and line number. Array element zero contains the information about the caller of HELP, element one contains information about its caller, etc. This keyword is useful for programs that require traceback information.

DEVICE

Set this keyword to show information about the currently selected graphics device. This information is dependent on the abilities of the current device, but the name of the device is always given. Arguments to HELP are ignored when DEVICE is specified.

DLM

Set this keyword to display all known dynamically loadable modules and their state (loaded or not loaded).

FILES

Set this keyword to display information about file units. If no arguments are supplied in the call to `HELP`, information on all open file units (except the special units 0, -1, and -2) is displayed. If arguments are provided, they are taken to be integer file unit numbers, and information on the specified file units is given.

For example, the command:

```
HELP, /FILES, -2, -1, 0
```

gives information below about the default file units:

Unit	Attributes	Name
-2	Write, Truncate, Tty, Reserved	<stderr>
-1	Write, Truncate, Tty, Reserved	<stdout>
0	Read, Tty, Reserved	<stdin>

The attributes column tells about the characteristics of the file. For instance, the file connected to logical file unit 2 is called “stderr” and is the standard error file. It is opened for write access (Write), is a new file (Truncate), is a terminal (Tty), and cannot be closed by the `CLOSE` command (Reserved).

FULL

By default, `HELP` filters its output in an attempt to only display information likely to be of use to the IDL end user. Specify `FULL` to see all available information on a given topic without any such filtering. The filtering applied by default depends on the type of information being requested:

- **Function keys:** By default, IDL will not display undefined function keys.
- **Structure Definitions And Objects:** Structures and objects that have had their definition hidden using the `STRUCT_HIDE` procedure are not usually listed.
- **Functions and Procedures:** Functions and procedures compiled with the `COMPILE_OPT HIDDEN` directive are not usually included in `HELP` output.

FUNCTIONS

Normally, the `ROUTINES` or `SOURCE_FILES` keywords produce information on both functions and procedures. If `FUNCTIONS` is specified, only output on functions is produced. If `FUNCTIONS` is used without either `ROUTINES` or `SOURCE_FILES`, `ROUTINES` is assumed.

HEAP_VARIABLES

Set this keyword to display help information for all the current heap variables.

KEYS

Set this keyword to show current function key definitions as set by `DEFINE_KEY`, for those function keys that are currently programmed to perform a function. If no arguments are supplied, information on all function keys is displayed. If arguments are provided, they must be scalar strings containing the names of function keys, and information on the specified keys is given. Under UNIX, this keyword is different from `ALL_KEYS` because that keyword displays every key, no matter what its current programming. Under Windows, every key is always displayed; setting `KEYS` produces the same result as setting `ALL_KEYS`.

LAST_MESSAGE

Set this keyword to display the last error message issued by IDL.

LEVEL

By default, `HELP` displays information for variables in the scope of the currently executing routine. To change this behavior, set this keyword to a scalar integer indicating the scope in which the variables for displaying help should be found. The level value can be either absolute or relative to the current scope, as shown in the following table:

Value	Meaning
0	Level 0 always refers to the frame for the currently executing routine
< 0	Negative levels are a relative specification and refer to the frames of active routines relative to the currently executing routine. Level -1 is the direct caller of the current routine, -2 is the caller of the caller of the current routine, and so forth. If you specify a level that is deeper than the current call stack, <code>SCOPE_VARFETCH</code> clips the value not to go past <code>\$MAIN\$</code> .

Table 3-46: `HELP LEVEL` Keyword Values

Value	Meaning
> 0	Positive levels are an absolute specification and refer directly to the specified frame. Level 1 is \$MAIN\$, level 2 is the routine called from \$MAIN\$, and so forth. If you specify a level that is past the currently executing routine, SCOPE_VARFETCH clips the value to the level of the current routine.

Table 3-46: HELP LEVEL Keyword Values

For more information on scope level, see [“SCOPE_VARFETCH”](#) on page 1846.

MEMORY

Set this keyword to see a report on the amount of dynamic memory (in bytes) currently in use by the IDL session; the maximum amount of dynamic memory allocated since the last call to `HELP, /MEMORY`; and the number of times dynamic memory has been allocated and deallocated. Arguments to `HELP` are ignored when `MEMORY` is specified.

MESSAGES

Set this keyword to display all known message blocks and the error space range into which they are loaded.

NAMES

A string used to determine the names of the variables, whose values are to be printed. A string match (equivalent to the `STRMATCH` function with the `FOLD_CASE` keyword set) is used to decide if a given variable will be displayed. The match string can contain any wildcard expression supported by `STRMATCH`, including “*” and “?”.

For example, to print only the values of variables beginning with “A”, use the command `HELP, NAMES='a*'`. Similarly, `HELP, NAMES='?'` prints the values of all variables with a single-character name.

`NAMES` also works with the output from the following keywords:

- `DLM`
- `HEAP_VARIABLES`

- MESSAGES
- OBJECTS
- ROUTINES
- SOURCE_FILES
- STRUCTURES
- SYSTEM_VARIABLES

OBJECTS

Set this keyword to display information on defined object classes. If no arguments are provided, all currently-defined object classes are shown. If no arguments are provided, and the information you are looking for is not displayed, use the FULL keyword to prevent HELP from filtering the output. If arguments are provided, the definition of the object class for the heap variables referred to is displayed.

Information is provided on inherited superclasses and all *known methods*. A method is known to IDL only if it has been compiled in the current IDL session and called by its own class or a subclass. Methods that have not been compiled yet will not be shown. Thus, the list of methods displayed by HELP is not necessarily a complete list of all possible method for the object class.

If called within a class' method, the OBJECTS keyword also displays the instance data of the object on which it was called.

OUTPUT

Set this keyword equal to a named variable that will contain a string array containing the formatted output of the HELP command. Each line of formatted output becomes a single element in the string array.

Warning

The OUTPUT keyword is primarily for use in capturing HELP output in order to display it someplace else, such as in a text widget. This keyword is *not* intended to be used in obtaining programmatic information about the IDL session, and is formatted to be human readable. RSI reserves the right to change the format and content of this text *at any time, without warning*. If you find yourself using OUTPUT for a non-display purpose, consider using the STRUCTURE keyword to the [SIZE](#) function.

PATH_CACHE

Set this keyword to display a list of directories currently included in the IDL path cache, along with the number of .pro or .sav files found in those directories. See [PATH_CACHE](#) for details.

PROCEDURES

Normally, the `ROUTINES` or `SOURCE_FILES` keywords produce information on both functions and procedures. If `PROCEDURES` is specified, only output on procedures is produced. If `PROCEDURES` is used without either `ROUTINES` or `SOURCE_FILES`, `ROUTINES` is assumed.

RECALL_COMMANDS

Set this keyword to display the saved commands in the command input buffer. By default, IDL saves the last 20 lines of input in a buffer from which they can be recalled for command line editing. Arguments to `HELP` are ignored when `RECALL` is specified.

The number of lines saved can be changed by assigning the desired number of lines to the environment variable `!EDIT_INPUT` in the IDL startup file. See [“!EDIT_INPUT”](#) on page 4367 for details.

ROUTINES

Set this keyword to show a list of all compiled procedures and functions with their parameter names. Keyword parameters accepted by each module are shown to the right of the routine name. If no arguments are provided, and the information you are looking for is not displayed, use the `FULL` keyword to prevent `HELP` from filtering the output.

SHARED_MEMORY

Set this keyword to display information about all current shared memory and memory mapped file segments mapped into the current IDL process via the [SHMMAP](#) procedure.

SOURCE_FILES

Set this keyword to display information on procedures and functions written in the IDL language that have been compiled during the current IDL session. Full path names (relative to the current directory) of compiled `.pro` files are displayed. If no arguments are provided, and the information you are looking for is not displayed, use the `FULL` keyword to prevent `HELP` from filtering the output.

STRUCTURES

Set this keyword to display information on structure-type variables. If no arguments are provided, all currently-defined structures are shown. If no arguments are provided, and the information you are looking for is not displayed, use the `FULL`

keyword to prevent `HELP` from filtering the output. If arguments are provided, the structure definition for those expressions is displayed. It is often more convenient to use `HELP, /STRUCTURES` instead of `PRINT` to look at the contents of a structure variable because it shows the names of the fields as well as the data.

SYSTEM_VARIABLES

Set this keyword to display information on all system variables. Arguments are ignored.

TRACEBACK

Set this keyword to display the current nesting of procedures and functions.

Examples

To see general information on the current IDL session, enter:

```
HELP
```

To see information on the structure definition of the system variable `!D`, enter:

```
HELP, !D, /STRUCTURES
```

Version History

Introduced: Original

`SHARED_MEMORY` keyword added: 5.6

`PATH_CACHE` keyword added: 6.0

`LEVEL` keyword added: 6.1

HILBERT

The HILBERT function outputs a series that has all periodic terms phase-shifted by 90 degrees. This transform has the interesting property that the correlation between a series and its own Hilbert transform is mathematically zero.

This routine is written in the IDL language. Its source code can be found in the file `hilbert.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = HILBERT(*X* [, *D*])

Return Value

The return value is a complex-valued vector with the same size as the input vector. HILBERT generates the fast Fourier transform using the FFT function, and shifts the first half of the transform products by +90 degrees and the second half by -90 degrees. The constant elements in the transform are not changed. Angle shifting is accomplished by multiplying or dividing by the complex number, $i = (0.0000, 1.0000)$. The shifted vector is then submitted to FFT for transformation back to the “time” domain and the output is divided by the number elements in the vector to correct for multiplication effect peculiar to the FFT algorithm.

Note

Because HILBERT uses FFT, it exhibits the same side effects with respect to input arguments as that function.

Arguments

X

An n -element floating-point or complex-valued vector.

D

A flag for rotation direction. Set $D = +1$ for a positive rotation (the default). Set $D = -1$ for a negative rotation.

Keywords

None.

Version History

Introduced: Original

See Also

[FFT](#)

HIST_2D

The HIST_2D function returns the two dimensional density function (histogram) of two variables.

This routine is written in the IDL language. Its source code can be found in the file `hist_2d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = HIST_2D( V1, V2 [, BIN1=width] [, BIN2=height] [, MAX1=value]
                [, MAX2=value] [, MIN1=value] [, MIN2=value] )
```

Return Value

Returns a longword array of dimensions $(\text{MAX}(V1)+1, \text{MAX}(V2)+1)$. $\text{Result}(i,j)$ is equal to the number of simultaneous occurrences of $V1 = i$ and $V2 = j$ at the specified element.

Arguments

V1, V2

The arguments are arrays containing the variables. $V1$ and $V2$ may be of any numeric type except complex. If $V1$ and $V2$ do not contain the same number of elements, then the extra elements in the longer array are ignored.

Keywords

BIN1

Set this keyword equal to the size of each bin in the $V1$ direction (column width). If this keyword is not specified, the size is set to 1.

BIN2

Set this keyword equal to the size of each bin in the $V2$ direction (row height). If this keyword is not specified, the size is set to 1.

MAX1

Set this keyword equal to the maximum *V1* value to consider. If this keyword is not specified, then *V1* is searched for its largest value.

MAX2

Set this keyword equal to the maximum *V2* value to consider. If this keyword is not specified, then *V2* is searched for its largest value.

MIN1

Set this keyword to the minimum *V1* value to consider. If this keyword is not specified and if the smallest value of *V1* is greater than zero, then `MIN1=0` is used, otherwise the smallest value of *V1* is used.

MIN2

Set this keyword to the minimum *V2* value to consider. If this keyword is not specified and if the smallest value of *V2* is greater than zero, then `MIN2=0` is used; otherwise, the smallest value of *V2* is used.

Examples

To return the 2D histogram of two byte images:

```
R = HIST_2D(image1, image2)
```

To display the 2D histogram made from two floating point images, restricting the range from -1 to +1, and with 101 bins:

```
F1 = RANDOMN(seed, 256, 256)
F2 = RANDOMN(seed, 256, 256)
Result = HIST_2D(F1, F2, MIN1=-1, MAX1=1, $
    MIN2=-1, MAX2=1, BIN1=0.02, BIN2=0.02)
TVSCL, Result
```

Version History

Introduced: Pre 4.0

See Also

[H_EQ_CT](#), [H_EQ_INT](#), [HIST_EQUAL](#), [HISTOGRAM](#)

HIST_EQUAL

The HIST_EQUAL function returns a histogram-equalized byte array.

The HISTOGRAM function is used to obtain the density distribution of the input array. The histogram is integrated to obtain the cumulative density-probability function and finally the lookup function is used to transform to the output image.

Note

The first element of the histogram is always zeroed to remove the background.

This routine is written in the IDL language. Its source code can be found in the file `hist_equal.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = HIST_EQUAL( A [, BINSIZE=value] [, FCN=vector]
                    [, /HISTOGRAM_ONLY] [, MAXV=value] [, MINV=value]
                    [, OMAX=variable] [, OMIN=variable] [, PERCENT=value] [, TOP=value] )
```

Return Value

This function returns a histogram-equalized array of type byte, with the same dimensions as the input array. If the HISTOGRAM_ONLY keyword is set, then the output will be a vector of type LONG.

Arguments

A

The array to be histogram-equalized.

Keywords

BINSIZE

Set this keyword to the size of the bin to use. The default is BINSIZE=1 if A is a byte array, or, for other input types, the default is $(MAXV - MINV)/5000$.

FCN

Set this keyword to the desired cumulative probability distribution function in the form of a 256-element vector. If a probability distribution function is not supplied, IDL uses a linear ramp, which yields equal probability bin results. This function is later normalized, so magnitude is inconsequential; the function should, however, increase monotonically.

HISTOGRAM_ONLY

Set this keyword to return a vector of type LONG containing the cumulative distribution histogram, rather than the histogram equalized array.

MAXV

Set this keyword to the maximum value to consider. The default is 255 if *A* is a byte array, otherwise the maximum data value is used. Input elements greater than or equal to MAXV are output as 255.

MINV

Set this keyword to the minimum value to consider. The default is 0 if *A* is a byte array, otherwise the minimum data value is used. Input elements less than or equal to MINV are output as 0.

OMAX

Set this keyword to a named variable that, upon exit, will contain the maximum data value used in constructing the histogram.

OMIN

Set this keyword to a named variable that, upon exit, will contain the minimum data value used in constructing the histogram.

PERCENT

Set this keyword to a value between 0 and 100 to stretch the image histogram. The histogram will be stretched linearly between the limits that exclude the PERCENT fraction of the lowest values, and the PERCENT fraction of the highest values. This is an automatic, semi-robust method of contrast enhancement.

TOP

The maximum value of the scaled result. If TOP is not specified, 255 is used. Note that the minimum value of the scaled result is always 0.

Example

Create a sample image using the DIST function and display it:

```
image = DIST(100)
TV, image
```

Create a histogram-equalized version of the byte array, `image`, and display the new version. Use a minimum input value of 10, a maximum input value of 200, and limit the top value of the output array to 220:

```
new = HIST_EQUAL(image, MINV = 10, MAXV = 200, TOP = 220)
TV, new
```

Version History

Introduced: Original

See Also

[ADAPT_HIST_EQUAL](#), [H_EQ_CT](#), [H_EQ_INT](#), [HIST_2D](#), [HISTOGRAM](#)

HISTOGRAM

The HISTOGRAM function computes the density function of *Array*. In the simplest case, the density function, at subscript *i*, is the number of *Array* elements in the argument with a value of *i*.

Let F_i = the value of element *i*, $0 \leq i < n$. Let H_v = result of histogram function, an integer vector. The definition of the histogram function becomes:

$$H_v = \sum_{i=0}^{n-1} P(F_i, v), \quad v = 0, 1, 2, \dots, \left\lfloor \frac{\text{Max} - \text{Min}}{\text{Binsize}} \right\rfloor$$

$$P(F_i, v) = \begin{cases} 1, & v \leq (F_i - \text{Min})/\text{Binsize} < v + 1 \\ 0, & \text{Otherwise} \end{cases}$$

Warning

There may not always be enough virtual memory available to find the density functions of arrays that contain a large number of bins.

For bivariate probability distributions, use the HIST_2D function.

HISTOGRAM can optionally return an array containing a list of the original array subscripts that contributed to each histogram bin. This list, commonly called the reverse (or backwards) index list, efficiently determines which array elements are accumulated in a set of histogram bins. A typical application of the reverse index list is reverse histogram or scatter plot interrogation—a histogram bin or 2D scatter plot location is marked with the cursor and the original data items within that bin are highlighted.

Syntax

```
Result = HISTOGRAM( Array [, BINSIZE=value] [, INPUT=variable]
  [, LOCATIONS=variable] [, MAX=value] [, MIN=value] [, /NAN]
  [, NBINS=value] [, OMAX=variable] [, OMIN=variable]
  [, /L64 | REVERSE_INDICES=variable] )
```

Return Value

Returns a 32-bit or a 64-bit integer vector equal to the density function of the input *Array*.

Arguments

Array

The vector or array for which the density function is to be computed.

Keywords

BINSIZE

Set this keyword to the size of the bin to use. If this keyword is not specified, and NBINS is not set, then a bin size of 1 is used. If NBINS is set, the default is $BINSIZE = (MAX - MIN) / (NBINS - 1)$.

Note

The data type of the value specified for BINSIZE should match the data type of the *Array* argument. Since BINSIZE is converted to the data type of *Array*, specifying mismatched data types may produce undesired results.

INPUT

Set this keyword to a named variable that contains an array to be added to the output of HISTOGRAM. The density function of *Array* is added to the existing contents of INPUT and returned as the result. The array is converted to longword type if necessary and must have at least as many elements as are required to form the histogram. Multiple histograms can be efficiently accumulated by specifying partial sums via this keyword.

L64

By default, the return value of HISTOGRAM is 32-bit integer when possible, and 64-bit integer if the number of elements being processed requires it. Set L64 to force 64-bit integers to be returned in all cases. L64 controls the type of *Result* as well as the output from the REVERSE_INDICES keyword.

Note

Only 64-bit versions of IDL are capable of creating variables requiring a 64-bit result. Check the value of `!VERSION.MEMORY_BITS` to see if your IDL is 64-bit or not.

LOCATIONS

Set this keyword to a named variable in which to return the starting locations for each bin. The starting locations are given by $\text{MIN} + v * \text{BINSIZE}$, with $v = 0, 1, \dots, \text{NBINS} - 1$. `LOCATIONS` has the same number of elements as the `Result`, and has the same type as the input `Array`.

MAX

Set this keyword to the maximum value to consider. If this keyword is not specified, `Array` is searched for its largest value. If this keyword is not specified, and `Array` is of type byte, 255 is used.

Note

The data type of the value specified for `MAX` should match the data type of the input array. Since `MAX` is converted to the data type of the input array, specifying mismatched data types may produce undesired results.

Note

If `NBINS` is specified, the value for `MAX` will be adjusted to $\text{NBINS} * \text{BINSIZE} + \text{MIN}$. This ensures that the last bin has the same width as the other bins.

MIN

Set this keyword to the minimum value to consider. If this keyword is not specified, and `Array` is of type byte, 0 is used. If this keyword is not specified and `Array` is not of byte type, `Array` is searched for its smallest value.

Note

The data type of the value specified for `MIN` should match the data type of the input array. Since `MIN` is converted to the data type of the input array, specifying mismatched data types may produce undesired results.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

NBINS

Set this keyword to the number of bins to use. If `BINSIZE` is specified, the number of bins in *Result* is `NBINS`, starting at `MIN` and ending at `MIN+(NBINS-1)*BINSIZE`. If `MAX` is specified, the bins will be evenly spaced between `MIN` and `MAX`. It is an error to specify `NBINS` with both `BINSIZE` and `MAX`.

OMAX

Set this keyword to a named variable that will contain the maximum data value used in constructing the histogram.

OMIN

A named variable that, upon exit, contains the minimum data value used in constructing the histogram.

REVERSE_INDICES

Set this keyword to a named variable in which the list of reverse indices is returned. When possible, this list is returned as a 32-bit integer vector whose number of elements is the sum of the number of elements in the histogram, N , and the number of array elements included in the histogram, plus one. If the number of elements is too large to be contained in a 32-bit integer, or if the `L64` keyword is set, `REVERSE_INDICES` is returned as a 64-bit integer.

The subscripts of the original array elements falling in the i^{th} bin, $0 \leq i < N$, are given by the expression: `R(R[i] : R[i+1]-1)`, where `R` is the reverse index list. If `R[i]` is equal to `R[i+1]`, no elements are present in the i^{th} bin.

For example, make the histogram of array `A`:

```
H = HISTOGRAM(A, REVERSE_INDICES = R)

;Set all elements of A that are in the ith bin of H to 0.
IF R[i] NE R[i+1] THEN A[R[R[I] : R[i+1]-1]] = 0
```

The above is usually more efficient than the following:

```
bini = WHERE(A EQ i, count)
IF count NE 0 THEN A[bini] = 0
```

Examples

```
; Create a simple, 2D dataset:
D = DIST(200)
; Plot the histogram of D with a bin size of 1 and the default
; minimum and maximum:
PLOT, HISTOGRAM(D)
; Plot a histogram considering only those values from 10 to 50
; using a bin size of 4:
PLOT, HISTOGRAM(D, MIN = 10, MAX = 50, BINSIZE = 4)
```

The HISTOGRAM function can also be used to increment the elements of one vector whose subscripts are contained in another vector. To increment those elements of vector A indicated by vector B, use the command:

```
A = HISTOGRAM(B, INPUT=A, MIN=0, MAX=N_ELEMENTS(A)-1)
```

This method works for duplicate subscripts, whereas the following statement never adds more than 1 to any element, even if that element is duplicated in vector B:

```
A[B] = A[B]+1
```

For example, for the following commands:

```
A = LONARR(5)
B = [2,2,3]
PRINT, HISTOGRAM(B, INPUT=A, MIN=0, MAX=4)
```

IDL prints:

```
0 0 2 1 0
```

The commands:

```
A = LONARR(5)
A[B] = A[B]+1
PRINT, A
```

give the result:

```
0 0 1 1 0
```

The following example demonstrates how to use HISTOGRAM:

```

PRO t_histogram
data = [[-5, 4, 2, -8, 1], $
        [ 3, 0, 5, -5, 1], $
        [ 6, -7, 4, -4, -8], $
        [-1, -5, -14, 2, 1]]
hist = HISTOGRAM(data)
bins = FINDGEN(N_ELEMENTS(hist)) + MIN(data)
PRINT, MIN(hist)
PRINT, bins
PLOT, bins, hist, YRANGE = [MIN(hist)-1, MAX(hist)+1], PSYM = 10, $
    XTITLE = 'Bin Number', YTITLE = 'Density per Bin'
END

```

IDL prints:

```

0

-14.0000  -13.0000  -12.0000  -11.0000  -10.0000  -9.00000
-8.00000  -7.00000  -6.00000  -5.00000  -4.00000  -3.00000
-2.00000  -1.00000  0.00000  1.00000  2.00000  3.00000
 4.00000  5.00000  6.00000

```

Version History

Introduced: Original

LOCATIONS keyword added: 5.6

See Also

[H_EQ_CT](#), [H_EQ_INT](#), [HIST_2D](#), [HIST_EQUAL](#)

HLS

The HLS procedure creates a color table based on the HLS (Hue, Lightness, Saturation) color system.

Using the input parameters, a spiral through the double-ended HLS cone is traced. Points along the cone are converted from HLS to RGB. The current colortable (and the COLORS common block) contains the new colortable on exit.

This routine is written in the IDL language. Its source code can be found in the file `hls.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

HLS, *Litlo*, *Lithi*, *Satlo*, *Sathi*, *Hue*, *Loops* [, *Colr*]

Arguments

Litlo

Starting lightness, from 0 to 100%.

Lithi

Ending lightness, from 0 to 100%.

Satlo

Starting saturation, from 0 to 100%.

Sathi

Ending saturation, from 0 to 100%.

Hue

Starting Hue, from 0 to 360 degrees. Red = 0 degs, green = 120, blue = 240.

Loops

The number of loops through the color spiral. This parameter does not have to be an integer. A negative value causes the loops to traverse the spiral in the opposite direction.

Colr

An optional (256,3) integer array in which the new R, G, and B values are returned.
Red = *Colr*[:,0], green = *Colr*[:,1], blue = *Colr*[:,2].

Keywords

None.

Version History

Introduced: Original

See Also

[COLOR_CONVERT](#), [HSV](#), [PSEUDO](#)

HOUGH

The HOUGH function implements the Hough transform, used to detect straight lines within a two-dimensional image. This function can be used to return either the Hough transform, which transforms each nonzero point in an image to a sinusoid in the Hough domain, or the Hough backprojection, where each point in the Hough domain is transformed to a straight line in the image.

Hough Transform Theory

The Hough transform is defined for a function $A(x, y)$ as:

$$H(\theta, \rho) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} A(x, y) \delta(\rho - x \cos \theta - y \sin \theta) dx dy$$

where δ is the Dirac delta-function. With $A(x, y)$, each point (x, y) in the original image, A , is transformed into a sinusoid $\rho = x \cos \theta - y \sin \theta$, where ρ is the perpendicular distance from the origin of a line at an angle θ (The angle θ will be limited to $0 \leq \theta < \pi$ which could result in negative ρ values.):

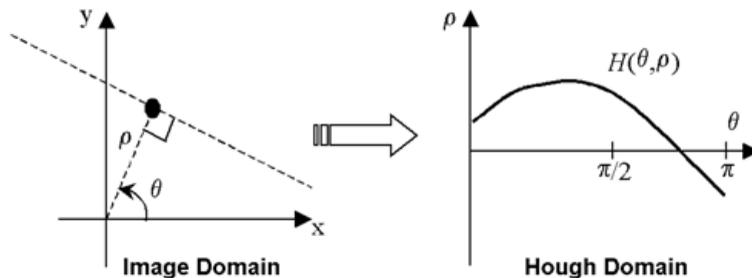


Figure 3-21: Hough Transform

Points that lie on the same line in the image will produce sinusoids that all cross at a single point in the Hough transform. For the inverse transform, or backprojection, each point in the Hough domain is transformed into a straight line in the image.

Usually, the Hough function is used with binary images, in which case $H(\theta, \rho)$ gives the total number of sinusoids that cross at point (θ, ρ) , and hence, the total number of points making up the line in the original image. By choosing a threshold T for $H(\theta, \rho)$, and using the inverse Hough function, you can filter the original image to keep only lines that contain at least T points.

How IDL Implements the Hough Transform

Consider an image A_{mn} of dimensions M by N , with array indices $m = 0, \dots, M-1$ and $n = 0, \dots, N-1$.

The discrete formula for the HOUGH function for A_{mn} is:

$$H(\theta, \rho) = \sum_m \sum_n A_{mn} \delta(\rho, [\rho'])$$

where the brackets [] indicate rounding to the nearest integer, and

$$\rho' = (m\Delta x + x_{\min})\cos\theta + (n\Delta y + y_{\min})\sin\theta$$

The pixels are assumed to have spacing Δx and Δy in the x and y directions. The delta-function is defined as:

$$\delta(\rho, [\rho']) = \begin{cases} 1 & \rho = [\rho'] \\ 0 & \text{otherwise} \end{cases}$$

How IDL Implements the Hough Backprojection

The backprojection, B_{mn} , contains all of the straight lines given by the (θ, ρ) points given in $H(\theta, \rho)$. The discrete formula is

$$B_{mn} = \begin{cases} \sum_{\theta} \sum_{\rho} H(\theta, \rho) \delta(n, [am + b]) & |\sin\theta| > \frac{\sqrt{2}}{2} \\ \sum_{\theta} \sum_{\rho} H(\theta, \rho) \delta(m, [a'n + b']) & |\sin\theta| \leq \frac{\sqrt{2}}{2} \end{cases}$$

where the slopes and offsets are given by:

$$a = -\frac{\Delta x \cos\theta}{\Delta y \sin\theta} \qquad b = \frac{\rho - x_{\min} \cos\theta - y_{\min} \sin\theta}{\Delta y \sin\theta}$$

$$a' = \frac{1}{a} \qquad b' = \frac{\rho - x_{\min} \cos\theta - y_{\min} \sin\theta}{\Delta x \cos\theta}$$

Syntax

Hough Transform:

```
Result = HOUGH( Array [, /DOUBLE] [, DRHO=scalar] [, DX=scalar]
  [, DY=scalar] [, /GRAY] [, NRHO=scalar] [, NTHETA=scalar]
  [, RHO=variable] [, RMIN=scalar] [, THETA=variable] [, XMIN=scalar]
  [, YMIN=scalar] )
```

Hough Backprojection:

```
Result = HOUGH( Array, /BACKPROJECT, RHO=variable, THETA=variable
  [, /DOUBLE] [, DX=scalar] [, DY=scalar] [, NX=scalar] [, NY=scalar]
  [, XMIN=scalar] [, YMIN=scalar] )
```

Return Value

The result of this function is a two-dimensional floating-point array, or a complex array if the input image is complex. If *Array* is double-precision, or if the `DOUBLE` keyword is set, the result is double-precision, otherwise, the result is single-precision.

Arguments

Array

The two-dimensional array of size M by N which will be transformed. If the keyword `GRAY` is not set, then, for the forward transform, *Array* is treated as a binary image with all nonzero pixels considered as 1.

Keywords

BACKPROJECT

If set, the backprojection is computed, otherwise, the forward transform is computed. When `BACKPROJECT` is set, *Result* will be an array of dimension NX by NY .

Note

The Hough transform is not one-to-one: each point (x, y) is not mapped to a single (θ, ρ) . Therefore, instead of the original image, the backprojection, or inverse transform, returns an image containing the set of all lines given by the (θ, ρ) points.

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

DRHO

Set this keyword equal to a scalar specifying the spacing $\Delta\rho$ between ρ coordinates, expressed in the same units as *Array*. The default is $1/\text{SQRT}(2)$ times the diagonal distance between pixels, $[(DX^2 + DY^2)/2]^{1/2}$. A larger value produces a coarser resolution by mapping multiple pixels onto a single ρ ; this is useful for images that do not contain perfectly straight lines. A smaller value may produce undersampling by trying to map fractional pixels onto ρ , and is not recommended. If **BACKPROJECT** is specified, this keyword is ignored.

DX

Set this keyword equal to a scalar specifying the spacing between the horizontal (X) coordinates. The default is 1.0.

DY

Set this keyword equal to a scalar specifying the spacing between the vertical (Y) coordinates. The default is 1.0.

GRAY

Set this keyword to perform a weighted Hough transform, with the weighting given by the pixel values. If **GRAY** is not set, the image is treated as a binary image with all nonzero pixels considered as 1. If **BACKPROJECT** is specified, this keyword is ignored.

NRHO

Set this keyword equal to a scalar specifying the number of ρ coordinates to use. The default is $2 \text{ CEIL}([\text{MAX}(X^2 + Y^2)]^{1/2} / \text{DRHO}) + 1$. If **BACKPROJECT** is specified, this keyword is ignored.

NTHETA

Set this keyword equal to a scalar specifying the number of θ coordinates to use over the interval $[0, \pi]$. The default is $\text{CEIL}(\pi [\text{MAX}(X^2 + Y^2)]^{1/2} / \text{DRHO})$. A larger value will produce smoother results, and is useful for filtering before backprojection. A smaller value will result in broken lines in the transform, and is not recommended. If **BACKPROJECT** is specified, this keyword is ignored.

NX

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of horizontal coordinates in the output array. The default is $\text{FLOOR}(2 \text{ MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform this keyword is ignored.

NY

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of vertical coordinates in the output array. The default is $\text{FLOOR}(2 \text{ MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform, this keyword is ignored.

RHO

For the forward transform, set this keyword to a named variable that, on exit, will contain the radial (ρ) coordinates. If BACKPROJECT is specified, this keyword must contain the ρ coordinates of the input *Array*.

RMIN

Set this keyword equal to a scalar specifying the minimum ρ coordinate to use for the forward transform. The default is $-0.5(\text{NRHO} - 1) \text{DRHO}$. If BACKPROJECT is specified, this keyword is ignored.

THETA

For the forward transform, set this keyword to a named variable containing a vector of angular (θ) coordinates to use for the transform. If NTHETA is specified instead, and THETA is set to a named variable, then on exit THETA will contain the θ coordinates. If BACKPROJECT is specified, this keyword must contain the θ coordinates of the input *Array*. HOUGH returns θ in $[0, \pi)$

XMIN

Set this keyword equal to a scalar specifying the X coordinate of the lower-left corner of the input *Array*. The default is $-(M-1)/2$, where *Array* is an M by N array. If BACKPROJECT is specified, set this keyword equal to a scalar specifying the X coordinate of the lower-left corner of the *Result*. In this case the default is $-\text{DX} (\text{NX}-1)/2$.

YMIN

Set this keyword equal to a scalar specifying the Y coordinate of the lower-left corner of the input *Array*. The default is $-(N-1)/2$, where *Array* is an M by N array. If *BACKPROJECT* is specified, set this keyword equal to a scalar specifying the Y coordinate of the lower-left corner of the *Result*. In this case the default is $-DY (NY-1)/2$.

Examples

This example computes the Hough transform of a random set of pixels:

```

PRO hough_example

;Create an image with a random set of pixels
seed = 12345 ; remove this line to get different random images
array = RANDOMU(seed,128,128) GT 0.95

;Draw three lines in the image
x = FINDGEN(32)*4
array[x,0.5*x+20] = 1b
array[x,0.5*x+30] = 1b
array[-0.5*x+100,x] = 1b

;Create display window, set graphics properties
WINDOW, XSIZE=330,YSIZE=630, TITLE='Hough Example'
!P.BACKGROUND = 255 ; white
!P.COLOR = 0 ; black
!P.FONT=2
ERASE

XYOUTS, .1, .94, 'Noise and Lines', /NORMAL
;Display the image. 255b changes black values to white:
TVSCL, 255b - array, .1, .72, /NORMAL

;Calculate and display the Hough transform
result = HOUGH(array, RHO=rho, THETA=theta)
XYOUTS, .1, .66, 'Hough Transform', /NORMAL
TVSCL, 255b - result, .1, .36, /NORMAL

;Keep only lines that contain more than 20 points:
result = (result - 20) > 0

;Find the Hough backprojection and display the output
backproject = HOUGH(result, /BACKPROJECT, RHO=rho, THETA=theta)
XYOUTS, .1, .30, 'Hough Backprojection', /NORMAL
TVSCL, 255b - backproject, .1, .08, /NORMAL

```

END

The following figure displays the output of this example. The top image shows three lines drawn within a random array of pixels that represent noise. The center image shows the Hough transform, displaying sinusoids for points that lie on the same line in the original image. The bottom image shows the Hough backprojection, after setting the threshold to retain only those lines that contain more than 20 points. The Hough inverse transform, or backprojection, transforms each point in the Hough domain into a straight line in the image.

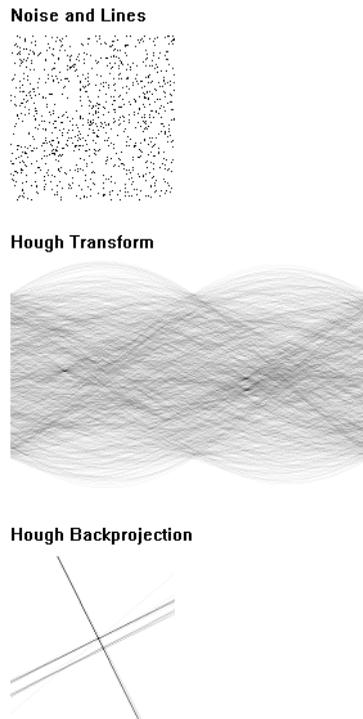


Figure 3-22: HOUGH example showing random pixels (top), Hough transform (center) and Hough backprojection (bottom)

References

1. Gonzalez, R.C., and R.E. Woods. *Digital Image Processing*. Reading, MA: Addison Wesley, 1992.

2. Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
3. Toft, Peter. *The Radon Transform: Theory and Implementation*. Denmark: Technical University; 1996. Ph.D. Thesis.
4. Weeks, Arthur. R. *Fundamentals of Electronic Image Processing*. New York: SPIE Optical Engineering Press, 1996.

Version History

Introduced: 5.4

See Also

[RADON](#)

HQR

The HQR function returns all eigenvalues of an upper Hessenberg array. Using the output produced by the ELMHES function, this function finds all eigenvalues of the original real, nonsymmetric array.

HQR is based on the routine `hqr` described in section 11.6 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the `LA_HQR` function.

Syntax

Result = HQR(*A* [, /COLUMN] [, /DOUBLE])

Return Value

The result is an n -element complex vector.

Arguments

A

An n by n upper Hessenberg array. Typically, *A* would be an array resulting from an application of ELMHES.

Keywords

COLUMN

Set this keyword if the input array *A* is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To compute the eigenvalues of a real, non-symmetric unbalanced array, first define the array *A*:

```
A = [[ 1.0, 2.0, 0.0, 0.0, 0.0], $
      [-2.0, 3.0, 0.0, 0.0, 0.0], $
      [ 3.0, 4.0, 50.0, 0.0, 0.0], $
      [-4.0, 5.0, -60.0, 7.0, 0.0], $
      [-5.0, 6.0, -70.0, 8.0, -9.0]]

; Compute the upper Hessenberg form of the array:
hes = ELMHES(A)
; Compute the eigenvalues:
evals = HQR(hes)

; Sort the eigenvalues into ascending order based on their
; real components:
evals = evals(SORT(FLOAT(eval)))

;Print the result.
PRINT, evals
```

IDL prints:

```
( -9.00000, 0.00000)( 2.00000, -1.73205)
( 2.00000, 1.73205)( 7.00000, 0.00000)
( 50.0000, 0.00000)
```

This is the exact solution vector to five-decimal accuracy.

Version History

Introduced: 4.0

See Also

[EIGENVEC](#), [ELMHES](#), [LA_HQR](#), [TRIQL](#), [TRIRED](#)

HSV

The HSV procedure creates a color table based on the HSV (Hue and Saturation Value) color system.

Using the input parameters, a spiral through the single-ended HSV cone is traced. Points along the cone are converted from HLS to RGB. The current colortable (and the COLORS common block) contains the new colortable on exit.

This routine is written in the IDL language. Its source code can be found in the file `hsv.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

HSV, *Vlo*, *Vhi*, *Satlo*, *Sathi*, *Hue*, *Loops* [, *Colr*]

Arguments

Vlo

Starting value, from 0 to 100%.

Vhi

Ending value, from 0 to 100%.

Satlo

Starting saturation, from 0 to 100%.

Sathi

Ending saturation, from 0 to 100%.

Hue

Starting Hue, from 0 to 360 degrees. Red = 0 degs, green = 120, blue = 240.

Loops

The number of loops through the color spiral. This parameter does not have to be an integer. A negative value causes the loops to traverse the spiral in the opposite direction.

Colr

An optional (256,3) integer array in which the new R, G, and B values are returned.
Red = *Colr*[:,0], green = *Colr*[:,1], blue = *Colr*[:,2].

Keywords

None.

Version History

Introduced: Original

See Also

[COLOR_CONVERT](#), [HLS](#), [PSEUDO](#)

IBETA

The IBETA function computes the incomplete beta function.

$$I_x(a, b) \equiv \frac{\int_0^x t^{a-1} (1-t)^{b-1} dt}{\int_0^1 t^{a-1} (1-t)^{b-1} dt}$$

This routine is written in the IDL language. Its source code can be found in the file `ibeta.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = IBETA(*A*, *B*, *Z* [, /DOUBLE] [, EPS=*value*] [, ITER=*variable*] [, ITMAX=*value*])

Return Value

If all arguments are scalar, the function returns a scalar. If all arguments are arrays, the function matches up the corresponding elements of *A*, *B*, and *Z*, returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other arguments are arrays, the function uses the scalar value with each element of the arrays, and returns an array with the same dimensions as the smallest input array.

If any of the arguments are double-precision or if the DOUBLE keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

A

A scalar or array that specifies the parametric exponent of the integrand. *A* may be complex.

B

A scalar or array that specifies the parametric exponent of the integrand. *B* may be complex.

Z

A scalar or array, in the interval $[0, 1]$, that specifies the upper limit of integration. Z may be complex. If Z is not complex then the values must be in the range $[0, 1]$.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double precision.

EPS

Set this keyword to the desired relative accuracy, or tolerance. The default tolerance is $3.0e-7$ for single precision, and $3.0d-12$ for double precision.

ITER

Set this keyword to a named variable that will contain the actual number of iterations performed.

ITMAX

Set this keyword to specify the maximum number of iterations. The default value is 100.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Example 1

Compute the incomplete beta function for the corresponding elements of A, B, and Z.

```
; Define an array of parametric exponents:  
A = [0.5, 0.5, 1.0, 5.0, 10.0, 20.0]  
B = [0.5, 0.5, 0.5, 5.0, 5.0, 10.0]
```

```

; Define the upper limits of integration:
Z = [0.01, 0.1, 0.1, 0.5, 1.0, 0.8]
; Compute the incomplete beta functions:
result = IBETA(A, B, Z)
PRINT, result

```

IDL prints:

```
[0.0637686, 0.204833, 0.0513167, 0.500000, 1.00000, 0.950736]
```

Example 2

This example shows the difference in accuracy between the incomplete beta function computed with a low tolerance and the incomplete beta function computed with a high tolerance. The resulting surfaces show the relative errors of each. The relative error of the low tolerance ranges from 0 to 0.002 percent. The relative error of the high tolerance ranges from 0 to 0.000000004 percent.

```

PRO UsingIBETAwithEPS

; Define an array of parametric exponents.
parameterA = (DINDGEN(101)/100. + 1.D) # REPLICATE(1.D, 101)
parameterB = REPLICATE(1.D, 101) # (DINDGEN(101)/10. + 1.D)

; Define the upper limits of integration.
upperLimits = REPLICATE(0.1D, 101, 101)

; Compute the incomplete beta functions.
betaFunctions = IBETA(parameterA, parameterB, upperLimits)

; Compute the incomplete beta functions with a less
; accurate tolerance set.
laBetaFunctions = IBETA(parameterA, parameterB, $
    upperLimits, EPS = 3.0e-4)

; Compute relative error.
relativeError = 100.* $
    ABS((betaFunctions - laBetaFunctions)/betaFunctions)

; Display resulting relative error.
WINDOW, 0, TITLE = 'Compare IBETA with Less Accurate EPS'
SURFACE, relativeError, parameterA, parameterB, $
    /XSTYLE, /YSTYLE, TITLE = 'Relative Error', $
    XTITLE = 'Parameter A', YTITLE = 'Parameter B', $
    ZTITLE = 'Percent Error (%)', CHARSIZE = 1.5

; Compute the incomplete beta functions with a more
; accurate tolerance set..

```

```

maBetaFunctions = IBETA(parameterA, parameterB, $
    upperLimits, EPS = 3.0e-10)

; Compute relative error.
relativeError = 100.* $
    ABS((maBetaFunctions - betaFunctions)/maBetaFunctions)

; Display resulting relative error.
WINDOW, 1, TITLE = 'Compare IBETA with More Accurate EPS'
SURFACE, relativeError, parameterA, parameterB, $
    /XSTYLE, /YSTYLE, TITLE = 'Relative Error', $
    XTITLE = 'Parameter A', YTITLE = 'Parameter B', $
    ZTITLE = 'Percent Error (%)', CHARSIZE = 1.5

in = ''
READ,"Press enter",in
WDELETE, 0, 1

END

```

Version History

Introduced: 4.0

A, *B*, and *Z* arguments accept complex input: 5.6

See Also

[BETA](#), [GAMMA](#), [IGAMMA](#), [LNGAMMA](#)

ICONTOUR

The ICONTOUR procedure creates an iTool and associated user interface (UI) configured to display and manipulate contour data.

Note

If no arguments are specified, the ICONTOUR procedure creates an empty Contour tool.

This routine is written in the IDL language. Its source code can be found in the file `icontour.pro` in the `lib/itools` subdirectory of the IDL distribution.

Using Palettes

Contour colors can be specified in several ways. By default, all contour levels are black. The `COLOR` keyword can be used to change the color of all contour levels. For example, you can change contour levels to red by setting `COLOR = [255, 0, 0]`. Individual color levels can be specified when the iContour tool is in palette color mode, which allows a color table to be used. You can activate the palette color mode from the IDL Command Line by setting either of the `RGB_TABLE` or `RGB_INDICES` keywords, or from the iContour tool's property sheet by changing the **Use palette color** setting to `True`.

Note

If you are not in the palette color mode, the colors of individual levels may be modified in the contour level properties dialog. If you are in the palette color mode, the ability to edit individual colors in the contour level properties dialog is disabled. However, changing the **Use palette color** setting to `False` does not switch you back to previously set colors. It simply converts the colors referenced by indices to direct color values that can be individually modified. A common practice is to switch to palette color mode, select a palette, then change **Use palette color** to `False`. The colors of the palette are now loaded as individual contour colors that can each be edited in the contour level properties dialog.

If the iContour tool is in palette color mode, a colorbar can be inserted through the **Insert** menu. The colorbar displays a sample of the current palette associated with the contour display. The data values of the axis of the colorbar are based on the data range of the `Z` argument and the contour level values.

The minimum value of the colorbar axis represents the minimum of the data range. The maximum value of the axis is the greater of than the maximum of the data range and the highest contour level value.

Note

When IDL computes default contour levels, the highest contour level may be above the maximum value of the data.

Syntax

ICONTOUR[, Z[, X, Y]]

iTool Common Keywords: [, BACKGROUND_COLOR=*value*]

[, DIMENSIONS=[*x, y*] [, IDENTIFIER=*variable*] [, LOCATION=[*x, y*]]

[, MACRO_NAMES=*string or string array*] [, NAME=*string*]

[, /NO_SAVEPROMPT] [, OVERPLOT=*iToolID*] [, STYLE_NAME=*string*] [,

TITLE=*string*] [, VIEW_GRID=[*columns, rows*]] [, /VIEW_NEXT]

[, VIEW_NUMBER=*integer*]

Contour Keywords: [, AM_PM=*vector*] [, ANISOTROPY=[*x, y, z*]]

[, C_COLOR=*array*] [, C_FILL_PATTERN=*array*]

[, C_LABEL_INTERVAL=*vector*] [, C_LABEL_NOGAPS=*vector*]

[, C_LABEL_OBJECTS=*array*] [, C_LABEL_SHOW=*vector*]

[, C_LINestyle=*array*] [, C_THICK=*float array*{each element 1.0 to 10.0}]

[, C_USE_LABEL_COLOR=*vector*]

[, C_USE_LABEL_ORIENTATION=*vector*] [, C_VALUE=*scalar or vector*]

[, CLIP_PLANES=*array*] [, COLOR=*RGB vector*] [, DAYS_OF_WEEK=*vector*]

[, DEPTH_OFFSET=*integer*] [, /DOWNHILL] [, /FILL] [, GRID_UNITS=*value*]

[, /HIDE] [, LABEL_FONT=*objref*] [, LABEL_FORMAT=*string*]

[, LABEL_FRMTDATA=*value*] [, LABEL_UNITS=*string*]

[, MAX_VALUE=*value*] [, MIN_VALUE=*value*] [, MONTHS=*vector*]

[, N_LEVELS=*value*] [, /PLANAR] [, RGB_INDICES=*vector*]

[, RGB_TABLE=*array*] [, SHADE_RANGE=[*min, max*] [, SHADING={0 | 1}]

[, TICKINTERVAL=*value*] [, TICKLEN=*value*]

[, USE_TEXT_ALIGNMENTS=*value*] [, ZVALUE=*value*]

Axis Keywords: [, [XYZ]GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]
 [, [XYZ]MAJOR=*integer*] [, [XYZ]MINOR=*integer*]
 [, [XYZ]RANGE=[*min, max*] [, [XYZ]SUBTICKLEN=*ratio*]
 [, [XYZ]TEXT_COLOR=*RGB vector*]
 [, [XYZ]TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
 [, [XYZ]TICKFONT_SIZE=*integer*]
 [, [XYZ]TICKFONT_STYLE={0 | 1 | 2 | 3}]
 [, [XYZ]TICKFORMAT=*string or string array*]
 [, [XYZ]TICKINTERVAL=*value*] [, [XYZ]TICKLAYOUT={0 | 1 | 2}]
 [, [XYZ]TICKLEN=*value*] [, [XYZ]TICKNAME=*string array*]
 [, [XYZ]TICKUNITS=*string*] [, [XYZ]TICKVALUES=*vector*]
 [, [XYZ]TITLE=*string*]

Arguments

X

A vector or two-dimensional array specifying the x -coordinates for the contour surface. If X is a vector, each element of X specifies the x -coordinate for a column of Z (e.g., $X[0]$ specifies the x -coordinate for $Z[0, *]$). If X is a two-dimensional array, each element of X specifies the x -coordinate of the corresponding point in Z (i.e., X_{ij} specifies the x -coordinate for Z_{ij}).

Y

A vector or two-dimensional array specifying the y -coordinates for the contour surface. If Y is a vector, each element of Y specifies the y -coordinate for a row of Z (e.g., $Y[0]$ specifies the y -coordinate for $Z[* , 0]$). If Y is a two-dimensional array, each element of Y specifies the y -coordinate of the corresponding point in Z (Y_{ij} specifies the y -coordinate for Z_{ij}).

Z

A vector or two-dimensional array containing the values to be contoured. If the X and Y arguments are provided, the contour is plotted as a function of the (x, y) locations specified by their contents. Otherwise, the contour is generated as a function of the two-dimensional array index of each element of Z .

Keywords

Note

Keywords to the ICONTOUR routine that correspond to the names of *registered properties* of the iContour tool must be specified in full, without abbreviation.

AM_PM

Set this keyword to a vector of 2 strings indicating the names of the AM and PM strings when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the LABEL_FORMAT keyword. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual for more information on format codes.

ANISOTROPY

Set this keyword equal to a three-element vector $[x, y, z]$ that represents the multipliers to be applied to the internally computed correction factors along each axis that account for anisotropic geometry. Correcting for anisotropy is particularly important for the appropriate representations of downhill tickmarks.

By default, IDL will automatically compute correction factors for anisotropy based on the $[XYZ]$ range of the contour geometry. If the geometry (as provided via the GEOMX, GEOMY, and GEOMZ keywords) falls within the range $[xmin, ymin, zmin]$ to $[xmax, ymax, zmax]$, then the default correction factors are computed as follows:

```
dx = xmax - xmin
dy = ymax - ymin
dz = zmax - zmin
; Get the maximum of the ranges:
maxRange = (dx > dy) > dz
IF (dx EQ 0) THEN xcorrection = 1.0 ELSE $
  xcorrection = maxRange / dx
IF (dy EQ 0) THEN ycorrection = 1.0 ELSE $
  ycorrection = maxRange / dy
IF (dz EQ 0) THEN zcorrection = 1.0 ELSE $
  zcorrection = maxRange / dz
```

This internally computed correction is then multiplied by the corresponding $[x, y, z]$ values of the ANISOTROPY keyword. The default value for this keyword is $[1,1,1]$. IDL converts, maintains, and returns this data as double-precision floating-point.

BACKGROUND_COLOR

Set this keyword to an RGB value specifying the color to be used as the background color for the view. The default is $[255, 255, 255]$ (white). The

`BACKGROUND_COLOR` keyword can be used when a tool is being created or when a new visualization is being created in an existing tool with the use of the `OVERPLOT`, `VIEW_NUMBER` or `VIEW_NEXT` keywords. The background color is applied to the current view. For example, if multiple views have been created with the `VIEW_GRID` keyword, and the `VIEW_NUMBER` keyword is used to create a visualization in the second view, use of the `BACKGROUND_COLOR` keyword would set the background color in the second view only.

C_COLOR

Set this keyword to a 3 by N array of RGB colors representing the colors to be applied at each contour level. If there are more contour levels than elements in this vector, the colors will be cyclically repeated. If `C_COLOR` is set to 0, all contour levels will be drawn in the color specified by the `COLOR` keyword (this is the default).

However, the `C_COLOR` keyword does not activate the palette color mode, which is recommended when working with contour levels and color. This mode can be activated with the `RGB_INDICES` and `RGB_TABLE` keywords. See [“Using Palettes”](#) on page 898 for more details.

C_FILL_PATTERN

Set this keyword to an array of `IDLgrPattern` objects representing the patterns to be applied at each contour level if the `FILL` keyword is non-zero. If there are more contour levels than fill patterns, the patterns will be cyclically repeated. If this keyword is set to 0, all contour levels are filled with a solid color (this is the default).

C_LABEL_INTERVAL

Set this keyword to a vector of values indicating the distance (measured parametrically relative to the length of each contour path) between labels for each contour level. If the number of contour levels exceeds the number of provided intervals, the `C_LABEL_INTERVAL` values will be repeated cyclically. The default is 0.4.

C_LABEL_NOGAPS

Set this keyword to a vector of values indicating whether gaps should be computed for the labels at the corresponding contour value. A zero value indicates that gaps will be computed for labels at that contour value; a non-zero value indicates that no gaps will be computed for labels at that contour value. If the number of contour levels exceeds the number of elements in this vector, the `C_LABEL_NOGAPS` values will be repeated cyclically. By default, gaps for the labels are computed for all levels (so that a contour line does not pass through the label).

C_LABEL_OBJECTS

Set this keyword to an array of object references to provide examples of labels to be drawn for each contour level. The objects specified via this keyword must inherit from one of the following classes:

- IDLgrSymbol
- IDLgrText

If a single object is provided, and it is an IDLgrText object, each of its strings will correspond to a contour level. If a vector of objects is used, any IDLgrText objects should have only a single string; each object will correspond to a contour level.

By default, with C_LABEL_OBJECTS set equal to a null object, IDL computes text labels that are the string representations of the corresponding contour level values.

Note

The objects specified via this keyword are used as descriptors only. The actual objects drawn as labels are generated by IDL.

The contour labels will have the same color as their contour level (see C_COLOR) unless the C_USE_LABEL_COLOR keyword is specified. The orientation of the label will be automatically computed unless the C_USE_LABEL_ORIENTATION keyword is specified. The horizontal and vertical alignment of any text labels will default to 0.5 (i.e., centered) unless the USE_TEXT_ALIGNMENTS keyword is specified.

Note

The object(s) set via this keyword will not be destroyed automatically when the contour is destroyed.

C_LABEL_SHOW

Set this keyword to a vector of integers. For each contour value, if the corresponding value in the C_LABEL_SHOW vector is non-zero, the contour line for that contour value will be labeled. If the number of contour levels exceeds the number of elements in this vector, the C_LABEL_SHOW values will be repeated cyclically. The default is 0 indicating that no contour levels will be labeled.

C_LINestyle

Set this keyword to an array of linestyles representing the linestyles to be applied at each contour level. The array may be either a vector of integers representing pre-

defined linestyles, or an array of 2-element vectors representing a stippling pattern specification. If there are more contour levels than linestyles, the linestyles will be cyclically repeated. If this keyword is set to 0, all levels are drawn as solid lines (this is the default).

To use a pre-defined line style, set the `C_LINestyle` property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq \textit{repeat} \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINestyle = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

C_THICK

Set this keyword to an array of line thicknesses representing the thickness to be applied at each contour level, where each element is a value between 1.0 and 10.0 points. If there are more contour levels than line thicknesses, the thicknesses will be cyclically repeated. If this keyword is set to 0, all contour levels are drawn with a line thickness of 1.0 points (this is the default).

C_USE_LABEL_COLOR

Set this keyword to a vector of values (0 or 1) to indicate whether the `COLOR` property value for each of the label objects (for the corresponding contour level) is to be used to draw that label. If the number of contour levels exceeds the number of elements in this vector, the `C_USE_LABEL_COLOR` values will be repeated

cyclically. By default, this value is zero, indicating that the COLOR properties of the label objects will be ignored, and the C_COLOR property for the contour object will be used instead.

C_USE_LABEL_ORIENTATION

Set this keyword to a vector of values (0 or 1) to indicate whether the orientation for each of the label objects (for the corresponding contour level) is to be used when drawing the label. For text, the orientation of the object corresponds to the BASELINE and UPDIR property values; for a symbol, this refers to the default (un-rotated) orientation of the symbol. If the number of contour levels exceeds the number of elements in this vector, the C_USE_LABEL_ORIENTATION values will be repeated cyclically. By default, this value is zero, indicating that orientation of the label object(s) will be set to automatically computed values (to correspond to the direction of the contour paths).

C_VALUE

Set this keyword to a scalar value or a vector of values for which contour values are to be drawn. If this keyword is set to 0, contour levels will be evenly sampled across the range of the Z argument, using the value of the N_LEVELS keyword to determine the number of samples. IDL converts, maintains, and returns this data as double-precision floating-point.

CLIP_PLANES

Set this keyword to an array of dimensions $[4, N]$ specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COLOR

Set this keyword to the color to be used to draw the contours. This color is specified as an RGB vector. The default is [0, 0, 0]. This value will be ignored if the C_COLOR keyword is set to a vector.

DAYS_OF_WEEK

Set this keyword to a vector of 7 strings to indicate the names to be used for the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdWA format codes) with the LABEL_FORMAT keyword. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual for more information on format codes.

DEPTH_OFFSET

Set this keyword to an integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.

The units are “Z-Buffer units,” where a value of 1 is used to specify a distance that corresponds to a single step in the device’s Z-Buffer.

Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms. This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for “layering” complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value, such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because a set of DEPTH_OFFSET values may produce better results on one machine than on another. Using IDL’s software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.

Note

DEPTH_OFFSET has no effect unless the FILL keyword is also set.

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in units specified by the UNITS keyword. If no value is provided, a default value of one half the screen size is used. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

DOWNHILL

Set this keyword to indicate that downhill tick marks should be rendered as part of each contour level to indicate the downhill direction relative to the contour line.

FILL

Set this keyword to indicate that the contours should be filled. The default is to draw the contour levels as lines without filling. Filling contours may produce less than satisfactory results if your data contains NaNs, or if the contours are not closed.

GRID_UNITS

Set this keyword to an integer specifying the units for the contour grid. This keyword applies only when there is a map projection inserted. It has the following values:

- 0 — None. The contour grid is in arbitrary units that are not tied to a map projection. The contour will not be warped to the current map projection.
- 1 — Meters. The contour grid is in meters, and is tied to a particular map projection. The contour will be automatically warped to the current map projection. You are responsible for choosing the dataspace map projection that matches the contour's map projection.
- 2 — Degrees. The contour grid is in degrees longitude/latitude, and will be automatically warped to the current map projection.

HIDE

Set this keyword to a boolean value to indicate whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

IDENTIFIER

Set this keyword to a named variable that will contain the `iToolID` for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LABEL_FONT

Set this keyword to an instance of an `IDLgrFont` object to describe the default font to be used for contour labels. This font will be used for all text labels automatically generated by IDL (i.e., if `C_LABEL_SHOW` is set but the corresponding `C_LABEL_OBJECTS` text object is not provided), or for any text label objects provided via `C_LABEL_OBJECTS` that do not already have the font property set. The default value for this keyword is a `NULL` object reference, indicating that 12 pt. Helvetica will be used.

LABEL_FORMAT

Set this keyword to a string that represents a format string or the name of a function to be used to format the contour labels. If the string begins with an open parenthesis, it is treated as a standard format string. (Refer to the Format Codes in the IDL Reference Guide.) If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate contour level labels.

The callback function is called with three parameters: *Axis*, *Index*, and *Value* and an optional `DATA` keyword, where:

- *Axis* is simply the value 2 to indicate that values along the Z axis are being formatted, which allows a single callback routine to be used for both axis labeling and contour labeling.
- *Index* is the contour level index (indices start at 0).
- *Value* is the data value of the current contour level.
- `DATA` is the optional keyword allowing any user-defined value specified through the `LABEL_FRMTDATA` keyword to `ICONTOUR`.

LABEL_FRMTDATA

Set this keyword to a value of any type. It will be passed via the `DATA` keyword to the user-supplied formatting function specified via the `LABEL_FORMAT` keyword, if any. By default, this value is 0, indicating that the `DATA` keyword will not be set (and furthermore, need not be supported by the user-supplied function).

LABEL_UNITS

Set this keyword to a string indicating the units to be used for default contour level labeling.

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the contour levels correspond to time values; IDL will determine the appropriate label format based upon the range of values covered by the contour Z data.
- "" - The empty string is equivalent to the "Numeric" unit. This is the default.

If any of the time units are utilized, then the contour values are interpreted as Julian date/time values.

Note

The singular form of each of the time unit strings is also acceptable (for example, LEVEL_UNITS='Day' is equivalent to LEVEL_UNITS='Days').

LOCATION

Set this keyword to a two-element vector of the form $[x, y]$ to specify the location of the upper left-hand corner of the tool relative to the display screen, in units specified by the UNITS keyword.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the iTool and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

MAX_VALUE

Set this keyword to the maximum value to be plotted. Data values greater than this value are treated as missing data. The default is the maximum value of the input Z data. IDL converts, maintains, and returns this data as double-precision floating-point.

MONTHS

Set this keyword to a vector of 12 strings indicating the names to be used for the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the C_LABEL_FORMAT keyword. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual for more information on format codes.

MIN_VALUE

Set this keyword to the minimum value to be plotted. Data values less than this value are treated as missing data. The default is the minimum value of the input Z data. IDL converts, maintains, and returns this data as double-precision floating-point.

NAME

Set this keyword to a string that specifies the name of this visualization.

NO_SAVEPROMPT

Set this keyword to cause the iTool *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

N_LEVELS

Set this keyword to the number of contour levels to generate. This keyword is ignored if the C_VALUE keyword is set to a vector, in which case, the number of levels is derived from the number of elements in that vector. Set this keyword to zero to indicate that IDL should compute a default number of levels based on the range of data values. This is the default.

OVERPLOT

Set this keyword to an `iToolID` to direct the graphical output of the particular tool to the tool specified by the provided `iToolID`.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

PLANAR

Set this keyword to indicate that the contoured data is to be projected onto a plane. Unlike the underlying `IDLgrContour` object, the default for `ICONTOUR` is planar (`PLANAR = 1`), which displays the contoured data in a plane. See the `ZVALUE` keyword to specify the Z value at which to display the planar Contour plot if it is displayed in a three dimensional data space.

RGB_INDICES

Set this keyword to a vector of indices into the color table to select colors to use for contour level colors. Setting the `RGB_INDICES` keyword activates the palette color mode, which allows colors from a specified color table to be used for the contour levels. The values set for `RGB_INDICES` are indices into the `RGB_TABLE` array of colors. If the number of colors selected using `RGB_INDICES` is less than the number of contour levels, the colors are repeated cyclically. If indices are not specified with the `RGB_INDICES` keyword, a default vector is constructed based on the values of the contour levels within the contour data range scaled to the byte range of `RGB_TABLE`.

See [“Using Palettes”](#) on page 898 for more details on the palette color mode.

RGB_TABLE

Set this keyword to either a 3 by 256 or 256 by 3 array containing color values to use for contour level colors. Setting the `RGB_TABLE` keyword activates the palette color mode, which allows colors from a specified color table to be used for the contour levels. The colors for each level are selected from `RGB_TABLE` using the `RGB_INDICES` vector. If indices are not specified with the `RGB_INDICES` keyword then a default vector is constructed based on the values of the contour levels within the contour data range scaled to the byte range of `RGB_TABLE`.

If the visualization is in palette color mode, but colors have not been specified with the `RGB_TABLE` keyword, the contour plot uses a default grayscale ramp.

See [“Using Palettes”](#) on page 898 for more details on the palette color mode.

SHADE_RANGE

Set this keyword to a two-element array that specifies the range of pixel values (color indices) to use for shading. The first element is the color index for the darkest pixel. The second element is the color index for the brightest pixel. This value is ignored when the contour is drawn to a graphics destination that uses the RGB color model.

SHADING

Set this keyword to an integer representing the type of shading to use:

- 0 = Flat (default): The color has a constant intensity for each face of the contour, based on the normal vector.
- 1 = Gouraud: The colors are interpolated between vertices, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied using the following rules:

If the tool exists and `/OVERPLOT` is specified, then the style is only applied to the newly-created visualizations within the current view. The current tool style is not updated with the new style, nor is the style applied to any other items within the view.

Otherwise, if the tool exists and either `VIEW_NEXT` or `VIEW_NUMBER` is being used to select a different view, then the style is applied to all items within that view. The current tool style is updated with the new style.

Otherwise, if a new tool is being created, then the style is applied to all items within all views. The current tool style is updated with the new style.

TICKINTERVAL

Set this keyword equal to a number indicating the distance between downhill tickmarks, in data units. If `TICKINTERVAL` is not set, or if you explicitly set it to zero, IDL will compute the distance based on the geometry of the contour. IDL converts, maintains, and returns this data as double-precision floating-point.

TICKLEN

Set this keyword equal to a number indicating the length of the downhill tickmarks, in data units. If TICKLEN is not set, or if you explicitly set it to zero, IDL will compute the length based on the geometry of the contour. IDL converts, maintains, and returns this data as double-precision floating-point

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool and is used for tool-related display purposes only – as the root of the hierarchy shown in the Tool Browser, for example.

USE_TEXT_ALIGNMENTS

Set this keyword to indicate that, for any IDLgrText labels (as specified via the C_LABEL_OBJECTS keyword), the ALIGNMENT and VERTICAL_ALIGNMENT property values for the given IDLgrText object(s) are to be used to draw the corresponding labels. By default, this value is zero, indicating that the ALIGNMENT and VERTICAL_ALIGNMENT properties of the label IDLgrText object(s) will be set to default values (0.5 for each, indicating centered labels).

VIEW_GRID

Set this keyword to a two-element vector of the form [columns, rows] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

[XYZ]GRIDSTYLE

The index of the linestyle to be used for plot tickmarks and grids (i.e., when [XYZ]TICKLEN is set to 1.0). See LINSTYLE for a list of linestyles.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely. ZMAJOR is ignored unless PLANAR is set to 0.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely. ZMINOR is ignored unless PLANAR is set to 0.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum. ZRANGE is ignored unless PLANAR is set to 0.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark. ZSUBTICKLEN is ignored unless PLANAR is set to 0.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black). ZTEXT_COLOR is ignored unless PLANAR is set to 0.

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

ZTICKFONT_INDEX is ignored unless PLANAR is set to 0.

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points. ZTICKFONT_SIZE is ignored unless PLANAR is set to 0.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

ZTICKFONT_STYLE is ignored unless PLANAR is set to 0.

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether TICKUNITS is specified.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Note

For more information, see “[[XYZ](#)]TICKFORMAT” on page 4345.

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

ZTICKFORMAT is ignored unless PLANAR is set to 0.

[[XYZ](#)]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [[XYZ](#)]RANGE and the number of major tick marks ([[XYZ](#)]MAJOR). The value of this keyword takes precedence over the value set for the [[XYZ](#)]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

ZTICKINTERVAL is ignored unless PLANAR is set to 0.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

ZTICKLAYOUT is ignored unless PLANAR is set to 0.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKLEN is ignored unless PLANAR is set to 0.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark. ZTICKNAME is ignored unless PLANAR is set to 0.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- "" - Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

ZTICKUNITS is ignored unless PLANAR is set to 0.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 C.E., respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKVALUES is ignored unless PLANAR is set to 0.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis. ZTITLE is ignored unless PLANAR is set to 0.

ZVALUE

For a planar contour plot, the height of the Z plane onto which the contour plot is projected.

Note

This keyword will not have any visual effect unless **PLANAR** is true and the plot is in a 3D dataspace, for example by selecting the **Surface** operation to add a surface plot to the dataspace along with the contour plot.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the **File** menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “[Data Import Methods](#)” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how to use the IDL Command Line to bring contour data into the iContour tool.

At the IDL Command Line, enter:

```
file = FILEPATH('convec.dat', SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [248, 248])
ICONTOUR, data
```

Double-click on a contour to display the contour properties. Change the **Number of levels** setting to 20, change **Use palette color** to **True**, and use the **Levels Color Table** setting to load the **EOS B** predefined color table through the **Load Predefined** button in the Palette Editor. Then, change the **Fill contours** setting to **True**.

The following figure displays the output of this example:

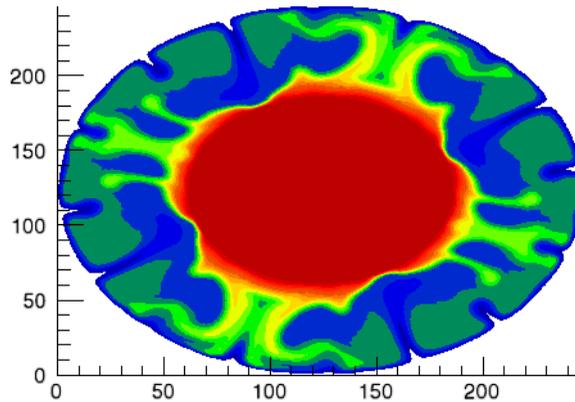


Figure 3-23: Earth Mantle Convection *iContour* Example

Example 2

This example shows how to use the *iTool* **File** → **Open** command to load DICOM data into the *iContour* tool.

At the IDL Command Line, enter:

```
ICONTOUR
```

Select **File** → **Open** to display the Open dialog, then browse to find `mr_brain.dcm` in the `examples/data` directory in the IDL distribution, and click **Open**.

Double-click on a contour to display the contour properties. Then, change **Use palette color** to `True` and the **Fill contours** setting to `True`.

Smooth the data by selecting **Operations** → **Filter** → **Smooth**.

The following figure displays the output of this example:

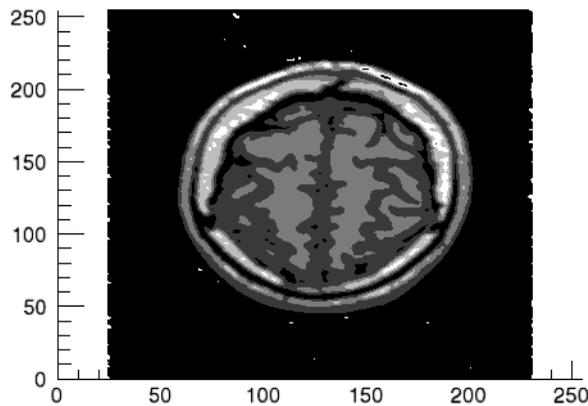


Figure 3-24: Smoothed Brain MRI iContour Example

Example 3

This example shows how to use the **File** → **Import** command to load binary data into the iContour tool.

At the IDL Command Line, enter:

```
ICONTOUR
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `idemosurf.dat` in the `examples/data` directory in the IDL distribution, and click **Next>>**.
3. At Step 3, select **Contour** and click **Finish**.

The Binary Template wizard is displayed. In the Binary Template, change **File's byte ordering** to `Little Endian`. Then, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** data (or a name of your choosing)
- **Type:** Float (32 bit)
- **Number of Dimensions:** 2
- **1st Dimension Size:** 200

- **2nd Dimension Size:** 200

Click **OK** to close the New Field dialog and the Binary Template dialog, and the contours are displayed.

Double-click on a contour to display the contour properties. Change the **Number of levels** setting to 10, change **Use palette color** to `True`, and use the **Levels Color Table** setting to load the `Rainbow18` predefined color table through the **Load Predefined** button in the Palette Editor. Then, change the **Fill contours** setting to `True`.

Change the **Projection** setting from `Planar` to `Three-D`.

The following figure displays the output of this example:

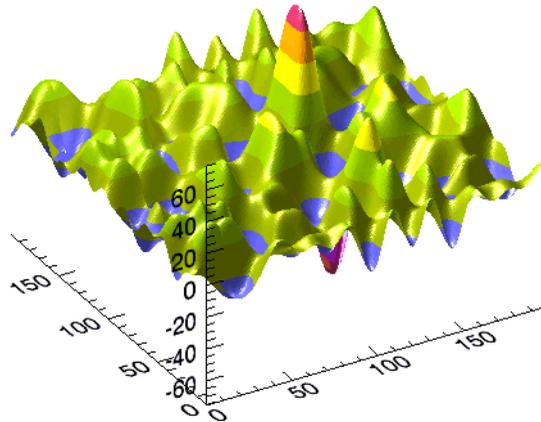


Figure 3-25: Filled Three-Dimensional iContour Example

Version History

Introduced: 6.0

`BACKGROUND_COLOR`, `GRID_UNITS`, `MACRO_NAMES`, `STYLE_NAME`
 keywords added: 6.1

IDENTITY

The `IDENTITY` function returns an identity array (an array with ones along the main diagonal and zeros elsewhere) of the specified dimensions.

This routine is written in the IDL language. Its source code can be found in the file `identity.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = IDENTITY( N [, /DOUBLE] )
```

Return Value

Returns an n by n identity array.

Arguments

N

The desired column and row dimensions.

Keywords

DOUBLE

Set this keyword to return a double-precision identity array.

Examples

```
; Define an array, A:
A = [[ 2.0,  1.0,  1.0, 1.5], $
      [ 4.0, -6.0,  0.0, 0.0], $
      [-2.0,  7.0,  2.0, 2.5], $
      [ 1.0,  0.5,  0.0, 5.0]]

; Compute the inverse of A using the INVERT function:
inverse = INVERT(A)

; Verify the accuracy of the computed inverse using the
; mathematical identity, A x A^-1 - I(4) = 0, where A^-1 is the
; inverse of A and I(4) is the 4 by 4 identity array:
PRINT, A ## inverse - IDENTITY(4)
```

Version History

Introduced: 5.0

See Also

[FINDGEN](#), [FLTARR](#)

IDL_Container Object Class

See “[IDL_Container](#)” in Chapter 9.

IDL_VALIDNAME

The IDL_VALIDNAME function determines whether a string may be used as a valid IDL variable name or structure tag name. Optionally, the routine can convert non-valid characters into underscores, returning a valid name string.

Syntax

Result = IDL_VALIDNAME(*String* [, /CONVERT_ALL] [, /CONVERT_SPACES])

Return Value

Returns the input string, optionally converting all spaces or non-alphanumeric characters to underscores. If the input string cannot be used as a valid variable or structure tag name, a null string is returned.

Arguments

String

A string representing the IDL variable or structure tag name to be checked.

Keywords

CONVERT_ALL

If this keyword is set, then *String* is converted into a valid IDL variable name using the following rules:

- All non-alphanumeric characters (except ‘_’, ‘!’ and ‘\$’) are converted to underscores
- If the first character of *String* is a number or a ‘\$’, then an underscore is prepended to the string
- If the first character of *String* is not a valid character (‘_’, ‘!’, ‘A’...’Z’) then that character is converted to an underscore
- If *String* is an empty string or a reserved word (such as “AND”) then an underscore is prepended to the string

Tip

The `CONVERT_ALL` keyword guarantees that a valid variable name is returned. It is useful in converting user-supplied strings into valid IDL variable names.

CONVERT_SPACES

If this keyword is set, then all spaces within *String* are converted to underscores. If *String* contains any other non-alphanumeric characters, then a null string is returned, indicating that the string cannot be used as a valid variable name.

Note

`CONVERT_SPACES` behaves the same as `CREATE_STRUCT` when checking structure tag names.

Examples

The following table provides `IDL_VALIDNAME` examples and their results.

Example	Result
<code>result = IDL_VALIDNAME('abc')</code>	'abc'
<code>result = IDL_VALIDNAME(' a b c')</code>	''
<code>result = IDL_VALIDNAME(' a b c ', /CONVERT_SPACES)</code>	'_a_b_c_'
<code>result = IDL_VALIDNAME('\$var')</code>	''
<code>result = IDL_VALIDNAME('\$var', /CONVERT_ALL)</code>	'_\$VAR'
<code>result = IDL_VALIDNAME('and')</code>	''
<code>result = IDL_VALIDNAME('and', /CONVERT_ALL)</code>	'_AND'

Table 3-47: `IDL_VALIDNAME` Examples

Version History

Introduced: 6.0

See Also

[CREATE_STRUCT](#)

IDLan* Object Class

The following IDLan* object classes are documented in [Chapter 5, “Analysis Object Classes”](#):

- [IDLanROI](#)
- [IDLanROIGroup](#)

IDLcom* Object Class

The following IDLcom* object classes are documented in [Chapter 9, “Miscellaneous Object Classes”](#):

- [IDLcomActiveX](#)
- [IDLcomIDispatch](#)

IDLff* Object Class

The following IDLff* object classes are documented in [Chapter 6, “File Format Object Classes”](#):

- [IDLffDICOM](#)
- [IDLffDXF](#)
- [IDLffLangCat](#)
- [IDLffShape](#)
- [IDLffXMLSAX](#)

IDLgr* Object Classes

The following IDLgr* object classes are documented in [Chapter 8](#), “Graphics Object Classes”:

- [IDLgrAxis](#)
- [IDLgrBuffer](#)
- [IDLgrClipboard](#)
- [IDLgrColorbar](#)
- [IDLgrColorbar](#)
- [IDLgrFont](#)
- [IDLgrImage](#)
- [IDLgrLegend](#)
- [IDLgrLight](#)
- [IDLgrModel](#)
- [IDLgrMPEG](#)
- [IDLgrPalette](#)
- [IDLgrPattern](#)
- [IDLgrPlot](#)
- [IDLgrPolygon](#)
- [IDLgrPolyline](#)
- [IDLgrPrinter](#)
- [IDLgrROI](#)
- [IDLgrROIgroup](#)
- [IDLgrScene](#)
- [IDLgrSurface](#)
- [IDLgrSymbol](#)
- [IDLgrTessellator](#)
- [IDLgrText](#)
- [IDLgrView](#)
- [IDLgrViewgroup](#)
- [IDLgrVolume](#)
- [IDLgrVRML](#)
- [IDLgrWindow](#)

IDLit* Object Classes

The following IDLit* object classes are documented in [Chapter 7, “iTools Object Classes”](#):

- [IDLitCommand](#)
- [IDLitCommandSet](#)
- [IDLitComponent](#)
- [IDLitContainer](#)
- [IDLitData](#)
- [IDLitDataContainer](#)
- [IDLitDataOperation](#)
- [IDLitIMessaging](#)
- [IDLitManipulator](#)
- [IDLitManipulatorContainer](#)
- [IDLitManipulatorManager](#)
- [IDLitManipulatorVisual](#)
- [IDLitOperation](#)
- [IDLitParameter](#)
- [IDLitParameterSet](#)
- [IDLitReader](#)
- [IDLitTool](#)
- [IDLitUI](#)
- [IDLitVisualization](#)
- [IDLitWindow](#)
- [IDLitWriter](#)

IDLITSYS_CREATETOOL

The IDLITSYS_CREATETOOL function creates an instance of the specified tool registered within the IDL Intelligent Tools system.

This routine is written in the IDL language. Its source code can be found in the file `idlitsys_createtool.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

```
Result = IDLITSYS_CREATETOOL(StrTool [, DIMENSIONS=[width,height]]
    [, IDENTIFIER=variable] [, INITIAL_DATA=data] [, LOCATION=[x,y]]
    [, MACRO_NAMES=string or string array] [, /NO_SAVEPROMPT]
    [, OVERPLOT=iToolID] [, STYLE_NAME=string]
    [, USER_INTERFACE=string] [, VIEW_GRID=vector] [, /VIEW_NEXT]
    [, VIEW_NUMBER=number] [, VISUALIZATION_TYPE=vistype] )
```

Return Value

Returns an `iToolID` that can be used to reference the created tool at a later time.

Arguments

StrTool

The name of a tool that has been registered with the iTools system via the `ITREGISTER` routine.

Keywords

Note

Additional keywords/properties associated with the target visualization at the command line are passed to the underlying system to be applied to the created tool and visualizations.

DIMENSIONS

Set this keyword to a two-element vector of the form `[width, height]` to specify the dimensions of the drawing area of the specific tool in device units. The minimum

width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

IDENTIFIER

Set this keyword to a named IDL variable that will contain the `iToolID` for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

INITIAL_DATA

Set this keyword to the data objects that are used to create the initial visualizations in the created tool.

LOCATION

Set this keyword to a two-element vector of the form $[x, y]$ to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the `iTool` and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

NO_SAVEPROMPT

Set this keyword to cause the `iTool` *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

OVERPLOT

Set this keyword to the `iToolID` of the tool in which the visualization is to be created. This `iToolID` can be obtained during the creation of a previous tool or from the `ITGETCURRENT` routine.

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied to all items within all views and the current tool style is updated with the new style.

USER_INTERFACE

Set this keyword equal to a string containing the name of a custom user interface that was previously registered using the USER_INTERFACE keyword to the ITREGISTER procedure.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created; it is ignored if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified.

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VISUALIZATION_TYPE

Set this keyword to a string containing the name of a registered visualization type that should be used to visualize any data specified by the INITIAL_DATA keyword. If this keyword is not specified, the iTool will select a visualization type based on the data type of the input data.

Obsolete Keywords

The following keywords are obsolete:

- PANEL_LOCATION

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

See [Chapter 5, “Example: Simple iTool”](#) in the *iTool Developer’s Guide* manual.

Version History

Introduced: 6.0

MACRO_NAMES, STYLE_NAME keywords added: 6.1

See Also

[ITREGISTER](#), [Chapter 5, “Creating an iTool Launch Routine”](#) in the *iTool Developer’s Guide* manual

IF...THEN...ELSE

The IF...THEN...ELSE statement conditionally executes a statement or block of statements.

Note

For information on using IF...THEN...ELSE and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

```
IF expression THEN statement [ ELSE statement ]
```

or

```
IF expression THEN BEGIN
```

```
  statements
```

```
ENDIF [ ELSE BEGIN
```

```
  statements
```

```
ENDELSE ]
```

Examples

The following example illustrates the use of the IF statement using the ELSE clause. Notice that the IF statement is ended with ENDIF, and the ELSE statement is ended with ENDELSE. Also notice that the IF statement can be used with or without the BEGIN...END block:

```
A = 2
B = 4

IF (A EQ 2) AND (B EQ 3) THEN BEGIN
  PRINT, 'A = ', A
  PRINT, 'B = ', B
ENDIF ELSE BEGIN
  IF A NE 2 THEN PRINT, 'A <> 2' ELSE PRINT, 'B <> 3'
ENDELSE
```

IDL Prints:

```
B <> 3
```

Version History

Introduced: Original

IGAMMA

The IGAMMA function computes the incomplete gamma function.

$$P_x(a) \equiv \frac{\int_0^x e^{-t} t^{a-1} dt}{\int_0^{\infty} e^{-t} t^{a-1} dt}$$

IGAMMA uses either a power series representation or a continued fractions method. If Z is less than or equal to $A+1$, a power series representation is used. If Z is greater than $A+1$, a continued fractions method is used.

This routine is written in the IDL language. Its source code can be found in the file `igamma.pro` in the `lib` subdirectory of the IDL distribution.

This routine can also be used with the GAMMA function to calculate the following other variations of the incomplete gamma function.

- $\gamma_x(a) = P_x(a)\Gamma(a) = \int_0^x e^{-t} t^{a-1} dt$, which can be calculated with IGAMMA and GAMMA:


```
igmaVariant1 = IGAMMA(A, Z)*GAMMA(A)
```
- $\Gamma_x(a) = \Gamma(a) - \gamma(a) = \int_x^{\infty} e^{-t} t^{a-1} dt$, which can be calculated with IGAMMA and GAMMA:


```
igmaVariant2 = GAMMA(A)*(1 - IGAMMA(A, Z))
```
- $\gamma_x^*(a) = x^{-a}P_x(a) = (x^{-a}/(\Gamma(a)))\int_0^x e^{-t} t^{a-1} dt$, which can be calculated with IGAMMA:


```
igmaVariant3 = x^(-A)*IGAMMA(A, Z)
```

Syntax

```
Result = IGAMMA( A, Z [, /DOUBLE] [, EPS=value] [, ITER=variable]
              [, ITMAX=value] [, METHOD=variable] )
```

Return Value

If both arguments are scalar, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of A and Z , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

If any of the arguments are double-precision or if the `DOUBLE` keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

A

A scalar or array that specifies the parametric exponent of the integrand. A may be complex.

Z

A scalar or array that specifies the upper limit of integration. Z may be complex. If Z is not complex then the values must be greater than or equal to zero.

Keywords

DOUBLE

Set this keyword to return a double-precision result.

EPS

Set this keyword to the desired relative accuracy, or tolerance. The default tolerance is $3.0e-7$ for single precision, and $3.0d-12$ for double precision.

ITER

Set this keyword to a named variable that will contain the actual number of iterations performed.

ITMAX

Set this keyword to specify the maximum number of iterations. The default value is 100,000.

METHOD

This keyword is obsolete. METHOD will still be accepted, but will always return 0.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Example 1

Compute the incomplete gamma function for the corresponding elements of A and Z.

```
; Define an array of parametric exponents:
A = [0.10, 0.50, 1.00, 1.10, 6.00, 26.00]

; Define the upper limits of integration:
Z = [0.0316228, 0.0707107, 5.00000, 1.04881, 2.44949, 25.4951]

; Compute the incomplete gamma functions:
result = IGAMMA(A, Z)

PRINT, result
```

IDL prints:

```
[0.742026, 0.293128, 0.993262, 0.607646, 0.0387318, 0.486387]
```

Version History

Introduced: 4.0

A and Z arguments accepts complex input: 5.6

See Also

[BETA](#), [GAMMA](#), [IBETA](#), [LNGAMMA](#)

IIMAGE

The IIMAGE procedure creates an iTool and associated user interface (UI) configured to display and manipulate image data.

Note

If no arguments are specified, the IIMAGE procedure creates an empty Image tool.

This routine is written in the IDL language. Its source code can be found in the file `iimage.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

IIMAGE[, Image[, X, Y]]

iTool Common Keywords: [, BACKGROUND_COLOR=*value*]

[, DIMENSIONS=[*x, y*] [, IDENTIFIER=*variable*] [, LOCATION=[*x, y*]]
 [, MACRO_NAMES=*string or string array*] [, NAME=*string*]
 [, /NO_SAVEPROMPT] [, OVERPLOT=*iToolID*] [, STYLE_NAME=*string*]
 [, TITLE=*string*] [, VIEW_GRID=[*columns, rows*]] [, /VIEW_NEXT]
 [, VIEW_NUMBER=*integer*]

Image Keywords: [, ALPHA_CHANNEL=*array*] [, BLUE_CHANNEL=*array*]

[, CHANNEL=*bitmask*] [, CLIP_PLANES=*array*]
 [, GREEN_CHANNEL=*array*] [, GRID_UNITS=*value*] [, /HIDE]
 [, IMAGE_DIMENSIONS=[*width, height*]] [, IMAGE_LOCATION=[*x, y*]]
 [, /INTERPOLATE] [, /ORDER] [, RED_CHANNEL=*array*]
 [, RGB_TABLE=*array*]

Axis Keywords: [, [XY]GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]

[, [XY]MAJOR=*integer*] [, [XY]MINOR=*integer*] [, [XY]RANGE=[*min, max*]]
 [, [XY]SUBTICKLEN=*ratio*] [, [XY]TEXT_COLOR=*RGB vector*]
 [, [XY]TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
 [, [XY]TICKFONT_SIZE=*integer*] [, [XY]TICKFONT_STYLE={0 | 1 | 2 | 3}]
 [, [XY]TICKFORMAT=*string or string array*] [, [XY]TICKINTERVAL=*value*]
 [, [XY]TICKLAYOUT={0 | 1 | 2}] [, [XY]TICKLEN=*value*]
 [, [XY]TICKNAME=*string array*] [, [XY]TICKUNITS=*string*]
 [, [XY]TICKVALUES=*vector*] [, [XY]TITLE=*string*]

Arguments

Image

Either a vector, a two-dimensional, or a three-dimensional array representing the sample values to be displayed as an image.

If *Image* is a vector:

- The *X* and *Y* arguments must also be present and contain the same number of elements. In this case, a dialog will be presented that offers the option of gridding the data to a regular grid (the results of which will be displayed as an indexed-color image).

If *Image* is a two-dimensional array:

- If either dimension is 3:

Image represents an array of *x*, *y*, and *z* values (either $[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots, [x_n, y_n, z_n]]$ or $[[x_0, x_1, \dots, x_n], [y_0, y_1, \dots, y_n], [z_0, z_1, \dots, z_n]]$ where *n* is the length of the other dimension). In this case, the *X* and *Y* arguments, if present, will be ignored. A dialog will be presented that allows the option of gridding the data to a regular grid (the results of which will be displayed as an indexed-color image, using the *z* values as the image data values).

- If neither dimension is 3:

Image represents an array of sample values to be displayed as a color-indexed image. If *X* and *Y* are provided, the sample values are defined as a function of the corresponding (*x*, *y*) locations; otherwise, the sample values are implicitly treated as a function of the array indices of each element of *Image*.

If *Image* is a three-dimensional array:

- If one of the dimensions is 3:

Image is a $3 \times n \times m$, $n \times 3 \times m$, or $n \times m \times 3$ array representing the red, green, and blue channels of the image to be displayed.

- If one of the dimensions is 4:

Image is a $4 \times n \times m$, $n \times 4 \times m$, or $n \times m \times 4$ array representing the red, green, blue, and alpha channels of the image to be displayed.

X

Either a vector or a two-dimensional array representing the x -coordinates of the image grid.

If the *Image* argument is a vector:

- X must be a vector with the same number of elements as *Image*.

If the *Image* argument is a two-dimensional array (for which neither dimension is 3):

- If X is a vector:

Each element of X specifies the x -coordinates for a column of *Image* (e.g., $X[0]$ specifies the x -coordinate for *Image*[0, *]).

- If X is a two-dimensional array:

Each element of X specifies the x -coordinate of the corresponding point in *Image* (X_{ij} specifies the x -coordinate of *Image* _{ij}).

Y

Either a vector or a two-dimensional array representing the y -coordinates of the image grid.

If the *Image* argument is a vector:

- Y must be a vector with the same number of elements.

If the *Image* argument is a two-dimensional array:

- If Y is a vector:

Each element of Y specifies the y -coordinates for a column of *Image* (e.g., $Y[0]$ specifies the y -coordinate for *Image*[*, 0]).

- If Y is a two-dimensional array:

Each element of Y specifies the y -coordinate of the corresponding point in *Image* (Y_{ij} specifies the y -coordinate of *Image* _{ij}).

Keywords

Note

Keywords to the IIMAGE routine that correspond to the names of *registered properties* of the iImage tool must be specified in full, without abbreviation.

ALPHA_CHANNEL

Set this keyword to a two-dimensional array representing the alpha channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the RED_CHANNEL, GREEN_CHANNEL, and BLUE_CHANNEL keywords.

BACKGROUND_COLOR

Set this keyword to an RGB value specifying the color to be used as the background color for the view. The default is [255, 255, 255] (white). The BACKGROUND_COLOR keyword can be used when a tool is being created or when a new visualization is being created in an existing tool with the use of the OVERPLOT, VIEW_NUMBER or VIEW_NEXT keywords. The background color is applied to the current view. For example, if multiple views have been created with the VIEW_GRID keyword, and the VIEW_NUMBER keyword is used to create a visualization in the second view, use of the BACKGROUND_COLOR keyword would set the background color in the second view only.

BLUE_CHANNEL

Set this keyword to a two-dimensional array representing the blue channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the RED_CHANNEL, GREEN_CHANNEL, and ALPHA_CHANNEL keywords.

CHANNEL

Set this keyword to a hexadecimal bitmask that defines which color channel(s) to draw. Each bit that is a 1 is drawn; each bit that is a 0 is not drawn. For example, 'ff0000'X represents a Blue channel write. The default is to draw all channels, and is represented by the hexadecimal value 'ffffff'X.

CLIP_PLANES

Set this keyword to an array of dimensions [4, *N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping

plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

DIMENSIONS

Set this keyword to a two-element vector of the form $[\text{width}, \text{height}]$ to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

GREEN_CHANNEL

Set this keyword to a two-dimensional array representing the green channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the RED_CHANNEL, BLUE_CHANNEL, and ALPHA_CHANNEL keywords.

GRID_UNITS

Set this keyword to an integer specifying the units for the image grid. This keyword applies only when there is a map projection inserted. It has the following values:

- 0 – None. The image grid is in arbitrary units that are not tied to a map projection. The image will not be warped to the current map projection.
- 1 – Meters. The image grid is in meters, and is tied to a particular map projection. The image will be warped to the current map projection.
- 2 – Degrees. The image grid is in degrees longitude/latitude, and will be automatically warped to the current map projection.

HIDE

Set this keyword to a boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

IDENTIFIER

Set this keyword to a named IDL variable that will contain the `iToolID` for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

IMAGE_DIMENSIONS

Set this keyword to a 2-element vector, $[width, height]$, to specify the image dimensions (in data units). By default, the dimensions match the pixel width of the image.

IMAGE_LOCATION

Set this keyword to a 2-element vector, $[x, y]$, to specify the image location (in data units). By default, the location is $[0, 0]$.

INTERPOLATE

Set this keyword to one (1) to display the `iImage` tool using bilinear interpolation. The default is to use nearest neighbor interpolation.

LOCATION

Set this keyword to a two-element vector of the form $[x, y]$ to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the `iTool` and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

NAME

Set this keyword to a string to specify the name for this particular tool. The name is used for tool-related display purposes only—as the root of the hierarchy shown in the Tool Browser, for example.

NO_SAVEPROMPT

Set this keyword to cause the iTool *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

ORDER

Set this keyword to force the rows of the image data to be drawn from top to bottom. By default, image data is drawn from the bottom row up to the top row.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

RED_CHANNEL

Set this keyword to a two-dimensional array representing the red channel pixel values for the image to be displayed. This keyword is ignored if the *Image* argument is present, and is intended to be used in conjunction with some combination of the GREEN_CHANNEL, BLUE_CHANNEL, and ALPHA_CHANNEL keywords.

RGB_TABLE

Set this keyword to a 3 by 256 or 256 by 3 byte array of RGB color values. If no color tables are supplied, the tool will provide a default 256-entry linear grayscale ramp.

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied using the following rules:

- If the tool exists and /OVERPLOT is specified, then the style is only applied to the newly-created visualizations within the current view. The current tool style is not updated with the new style, nor is the style applied to any other items within the view.

- Otherwise, if the tool exists and either `VIEW_NEXT` or `VIEW_NUMBER` is being used to select a different view, then the style is applied to all items within that view. The current tool style is updated with the new style.
- Otherwise, if a new tool is being created, then the style is applied to all items within all views. The current tool style is updated with the new style.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if `OVERPLOT`, `VIEW_NEXT`, or `VIEW_NUMBER` are specified then `VIEW_GRID` is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then `/VIEW_NEXT` will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless `/OVERPLOT` is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the `VIEW_NUMBER` before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless `/OVERPLOT` is set.

[XY]GRIDSTYLE

The index of the linestyle to be used for plot tickmarks and grids (i.e., when [XY]TICKLEN is set to 1.0). See **LINestyle** for a list of linestyles.

[XY]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

[XY]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

[XY]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum.

[XY]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XY]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XY]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XY]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XY]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XY]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether TICKUNITS is specified.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Note

For more information, see “[XYZ]TICKFORMAT” on page 4345.

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XY]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

[XY]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XY]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XY]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XY]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- `Numeric`
- `Years`
- `Months`
- `Days`
- `Hours`
- `Minutes`
- `Seconds`
- `Time` - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- `""` - Use an empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the `Numeric` unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 C.E., respectively.

[XY]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XY]TITLE

Set this keyword to a string representing the title of the specified axis.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the **File** menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “[Data Import Methods](#)” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how use the IDL Command Line to load data into the iImage tool.

At the IDL Command Line, enter:

```
file = FILEPATH('mineral.png', $
  SUBDIRECTORY = ['examples', 'data'])
data = READ_PNG(file)
IIMAGE, data, TITLE = 'Electron Image of Mineral Deposits'
```

Double-click the image to display image properties, and use the **Image Palette** setting to load the *Stern Special* predefined color table through the **Load Predefined** button in the Palette Editor.

Use the Text Annotation tool to insert a title at the top of the image. Select **Insert** → **Colorbars** to insert a color bar at the bottom of the image. Double-click on the colorbar to display its properties, and change the **Title** setting to *Stern Special*.

The following figure displays the output of this example:

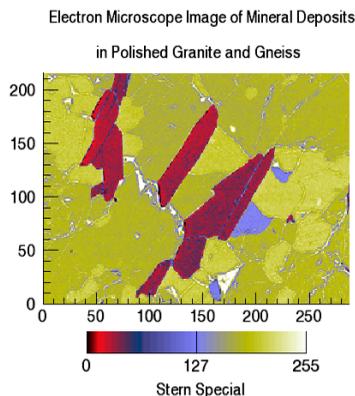


Figure 3-26: Mineral image Example with Sterns Color Table

Example 2

This example shows how to use the iTool **File** → **Open** command to load binary data into the iImage tool.

At the IDL Command Line, enter:

```
IIMAGE
```

Select **File** → **Open** to display the Open dialog, then browse to find `worldelv.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

In the Binary Template dialog, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** data (or a name of your choosing)
- **Type:** Byte (unsigned 8-bits)
- **Number of Dimensions:** 2
- **1st Dimension Size:** 360
- **2nd Dimension Size:** 360

Click **OK** to close the New Field dialog and the Binary Template dialog, and the image is displayed.

Note

For more information on using the Binary Template to import data, see “Using the `BINARY_TEMPLATE` Function” in Chapter 15 of the *Using IDL* manual.

Double-click the image to display image properties, and use the **Image Palette** setting to load the `STD_GAMMA-11` predefined color table through the **Load Predefined** button in the Palette Editor.

The following figure displays the output of this example:

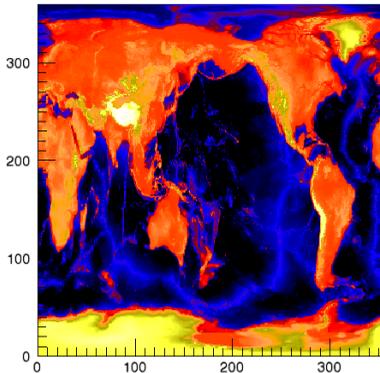


Figure 3-27: World Elevation Image Example

Example 3

This example shows how to use the IDL Import Data Wizard to load image data into the `iImage` tool.

At the IDL Command Line, enter:

```
I IMAGE
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `n_vasinfecta.jpg` in the `examples/data` directory in the IDL distribution, and click **Next>>**.
3. At Step 3, select **Image** and click **Finish**.

Define the edges within the image by selecting **Operations** → **Filter** → **Sobel Filter**.
The following figure displays the output of this example:

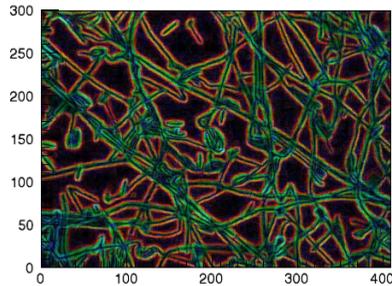


Figure 3-28: Sobel Filtered Neocosmospora Vasinfecta Image Example

Version History

Introduced: 6.0

BACKGROUND_COLOR, GRID_UNITS, MACRO_NAMES, STYLE_NAME
keywords added: 6.1

IMAGE_CONT

The IMAGE_CONT procedure overlays an image with a contour plot.

This routine is written in the IDL language. Its source code can be found in the file `image_cont.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
IMAGE_CONT, A [, /ASPECT] [, /INTERP] [, /WINDOW_SCALE]
```

Arguments

A

The two-dimensional array to display and overlay.

Keywords

ASPECT

Set this keyword to retain the image's aspect ratio. Square pixels are assumed. If WINDOW_SCALE is set, the aspect ratio is automatically retained.

INTERP

If this keyword is set, bilinear interpolation is used if the image is resized.

WINDOW_SCALE

Set this keyword to scale the window size to the image size. Otherwise, the image size is scaled to the window size. This keyword is ignored when outputting to devices with scalable pixels (e.g., PostScript).

Examples

```
; Create an image to display:  
A = BYTSCALE(DIST(356))  
  
; Display image and overplot contour lines:  
IMAGE_CONT, A, /WINDOW
```

Version History

Introduced: Original

See Also

[CONTOUR](#), [ICONTOUR](#), [IIMAGE](#), [TV](#)

IMAGE_STATISTICS

The IMAGE_STATISTICS procedure computes sample statistics for a given array of values. An optional mask may be specified to restrict computations to a spatial subset of the input data.

Syntax

```
IMAGE_STATISTICS, Data [, COUNT=variable] [, DATA_SUM=variable]
  [, /Labeled | [, LUT=array] [, MASK=array] [, MAXIMUM=variable]
  [, MEAN=variable] [, MINIMUM=variable] [, STDDEV=variable]
  [, SUM_OF_SQUARES=variable] [, VARIANCE=variable] [, /VECTOR]
  [, WEIGHT_SUM=variable] [, /WEIGHTED]
```

Arguments

Data

An N -dimensional input data array.

Keywords

COUNT

Set this keyword to a named variable to contain the number of samples that correspond to nonzero values within the mask.

DATA_SUM

Set this keyword to a named variable to contain the sum of the samples that lie within the mask.

Labeled

When set, this keyword indicates values in the mask representing region labels, where each pixel of the mask is set to the index of the region in which that pixel belongs (see the LABEL_REGION function in the *IDL Reference Guide*). If the Labeled keyword is set, each statistic's value is computed for each region index. Thus, a vector containing the results is provided for each statistic with one element per region. By default, this keyword is set to zero, indicating that all samples with a corresponding nonzero mask value are used to form a scalar result for each statistic.

Note

The `LABELED` keyword cannot be used with either the `WEIGHT_SUM` or the `WEIGHTED` keywords.

LUT

Set this keyword to a one-dimensional array. For non-floating point input *Data*, the pixel values are looked up through this table before being used in any of the statistical computations. This allows an integer image array to be calibrated to any user specified intensity range for the sake of calculations. The length of this array must include the range of the input array. This keyword may not be set with floating point input data. When signed input data types are used, they are first cast to the corresponding IDL unsigned type before being used to access this array. For example, the integer value -1 looks up the value 65535 in the LUT array.

MASK

An array of N , or $N-1$ (when the `VECTOR` keyword is used) dimensions representing the mask array. If the `LABELED` keyword is set, `MASK` contains the region indices of each pixel; otherwise statistics are only computed for data values where the `MASK` array is non-zero.

MAXIMUM

Set this keyword to a named variable to contain the maximum value of the samples that lie within the mask.

MEAN

Set this keyword to a named variable to contain the mean of the samples that lie within the mask.

MINIMUM

Set this keyword to a named variable to contain the minimum value of the samples that lie within the mask.

STDDEV

Set this keyword to a named variable to contain the standard deviation of the samples that lie within the mask.

SUM_OF_SQUARES

Set this keyword to a named variable to contain the sum of the squares of the samples that lie within the mask.

VARIANCE

Set this keyword to a named variable to contain the variance of the samples that lie within the mask.

VECTOR

Set this keyword to specify that the leading dimension of the input array is not to be considered spatial but consists of multiple data values at each pixel location. In this case, the leading dimension is treated as a vector of samples at the spatial location determined by the remainder of the array dimensions.

WEIGHT_SUM

Set the WEIGHT_SUM keyword to a named variable to contain the sum of the weights in the mask.

Note

The WEIGHT_SUM keyword cannot be used if the LABELED keyword is specified.

WEIGHTED

If the WEIGHTED keyword is set, the values in the MASK array are used to weight individual pixels with respect to their count value. If a MASK array is not provided, all pixels are assigned a weight of 1.0.

Note

The WEIGHTED keyword cannot be used if the LABELED keyword is specified.

Version History

Introduced: 5.3

IMAGINARY

The IMAGINARY function returns the imaginary part of its complex-valued argument.

Syntax

Result = IMAGINARY(*Complex_Expression*)

Return Value

If the complex-valued argument is double-precision, the result will be double-precision, otherwise the result will be single-precision floating-point.

Arguments

Complex_Expression

The complex-valued expression for which the imaginary part is desired.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

```
; Create an array of complex values:
C = COMPLEX([1,2,3],[4,5,6])

; Print just the imaginary parts of each element in C:
PRINT, IMAGINARY(C)
```

IDL prints:

```
4.00000 5.00000 6.00000
```

Version History

Introduced: Original

See Also

[COMPLEX](#), [DCOMPLEX](#), [REAL_PART](#)

IMAP

The IMAP procedure creates an iTool and associated user interface (UI) configured to display and manipulate image and contour data that is georeferenced, as well as shapefile data that is interactively imported once an iMap tool is created.

Note

If no arguments are specified, the IMAP procedure creates an empty iMap tool.

This procedure is written in the IDL language. Its source code can be found in the file `imap.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

Empty Map Projection

```
IMAP[, MAP_PROJECTION=string]
```

Image Visualization

```
IMAP[, Image[, X, Y]] [, GRID_UNITS=value] [, MAP_PROJECTION=string]
```

Contour Visualization

```
IMAP[, Z[, X, Y]] [, /CONTOUR [, GRID_UNITS=value]  
[, MAP_PROJECTION=string]
```

iTool Common Keywords: [, BACKGROUND_COLOR=*value*]
[, DIMENSIONS=[*x, y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x, y*]]
[, MACRO_NAMES=*string or string array*] [, NAME=*string*]
[, /NO_SAVEPROMPT] [, OVERPLOT=*iToolID*] [, STYLE_NAME=*string*]
[, TITLE=*string*] [, VIEW_GRID=[*columns, rows*]] [, /VIEW_NEXT]
[, VIEW_NUMBER=*integer*]

iTool Image Keywords: This procedure accepts all IIMAGE keywords. For more information, see “IIMAGE” on page 942.

iTool Contour Keywords: If the CONTOUR keyword is set, this procedure accepts all ICONTOUR keywords. For more information, see “ICONTOUR” on page 898.

Map Projection Keywords: [, CENTER_LATITUDE=*value*]
 [, CENTER_LONGITUDE=*value*] [, DATUM=*string*]
 [, FALSE_EASTING=*value*] [, FALSE_NORTHING=*value*] [, HEIGHT=*value*]
 [, HOM_AZIM_LONGITUDE=*value*] [, HOM_AZIM_ANGLE=*value*]
 [, HOM_LATITUDE1=*value*] [, HOM_LATITUDE2=*value*]
 [, HOM_LONGITUDE1=*value*] [, HOM_LONGITUDE2=*value*]
 [, IS_ZONES=*value*] [, IS_JUSTIFY=*value*]
 [, LIMIT=[*latmin, lonmin, latmax, lonmax*]] [, MERCATOR_SCALE=*value*]
 [, OEA_ANGLE=*value*] [, OEA_SHAPEM=*value*] [, OEA_SHAPEP=*value*]
 [, SEMIMAJOR_AXIS=*value*] [, SEMIMINOR_AXIS=*value*]
 [, SOM_INCLINATION=*value*] [, SOM_LONGITUDE=*value*]
 [, SOM_PERIOD=*value*] [, SOM_RATIO=*value*] [, SOM_FLAG=*value*]
 [, SOM_LANDSAT_NUMBER=*value*] [, SOM_LANDSAT_PATH=*value*]
 [, SPHERE_RADIUS=*value*] [, STANDARD_PARALLEL=*value*]
 [, STANDARD_PAR1=*value*] [, STANDARD_PAR2=*value*]
 [, TRUE_SCALE_LATITUDE=*value*] [, ZONE=*value*]

Axis Keywords: [, [XY]GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]
 [, [XY]MAJOR=*integer*] [, [XY]MINOR=*integer*] [, [XY]RANGE=[*min, max*]]
 [, [XY]SUBTICKLEN=*ratio*] [, [XY]TEXT_COLOR=*RGB vector*]
 [, [XY]TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
 [, [XY]TICKFONT_SIZE=*integer*] [, [XY]TICKFONT_STYLE={0 | 1 | 2 | 3}]
 [, [XY]TICKFORMAT=*string or string array*] [, [XY]TICKINTERVAL=*value*]
 [, [XY]TICKLAYOUT={0 | 1 | 2}] [, [XY]TICKLEN=*value*]
 [, [XY]TICKNAME=*string array*] [, [XY]TICKUNITS=*string*]
 [, [XY]TICKVALUES=*vector*] [, [XY]TITLE=*string*]

Image Visualization Arguments

Image

A vector, a two-dimensional array, or a three-dimensional array representing the sample values to be displayed as an image.

If *Image* is a vector:

- The *X* and *Y* arguments must also be present and contain the same number of elements. In this case, a dialog will be presented that offers the option of gridding the data to a regular grid (the results of which will be displayed as an indexed-color image).

If *Image* is a two-dimensional array:

- If either dimension is 3:

Image represents an array of x , y , and z values (either $[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots, [x_n, y_n, z_n]]$ or $[[x_0, x_1, \dots, x_n], [y_0, y_1, \dots, y_n], [z_0, z_1, \dots, z_n]]$ where n is the length of the other dimension). In this case, the X and Y arguments, if present, will be ignored. A dialog will be presented that allows the option of gridding the data to a regular grid (the results of which will be displayed as an indexed-color image, using the z values as the image data values).

- If neither dimension is 3:

Image represents an array of sample values to be displayed as a color-indexed image. If X and Y are provided, the sample values are defined as a function of the corresponding (x, y) locations; otherwise, the sample values are implicitly treated as a function of the array indices of each element of *Image*.

If *Image* is a three-dimensional array:

- If one of the dimensions is 3:

Image is a $3 \times n \times m$, $n \times 3 \times m$, or $n \times m \times 3$ array representing the red, green, and blue channels of the image to be displayed.

- If one of the dimensions is 4:

Image is a $4 \times n \times m$, $n \times 4 \times m$, or $n \times m \times 4$ array representing the red, green, blue, and alpha channels of the image to be displayed.

X

Either a vector or a two-dimensional array representing the x -coordinates of the image grid.

If the *Image* argument is a vector:

- X must be a vector with the same number of elements as *Image*.

If the *Image* argument is a two-dimensional array (for which neither dimension is 3):

- If X is a vector:

Each element of X specifies the x -coordinates for a column of *Image* (e.g., $X[0]$ specifies the x -coordinate for *Image*[0, *]).

- If X is a two-dimensional array:

Each element of X specifies the x -coordinate of the corresponding point in *Image* (X_{ij} specifies the x -coordinate of *Image* _{ij}).

Y

Either a vector or a two-dimensional array representing the y -coordinates of the image grid.

If the *Image* argument is a vector:

- Y must be a vector with the same number of elements.

If the *Image* argument is a two-dimensional array:

- If Y is a vector:

Each element of Y specifies the y -coordinates for a column of *Image* (e.g., $Y[0]$ specifies the y -coordinate for *Image*[*, 0]).

- If Y is a two-dimensional array:

Each element of Y specifies the y -coordinate of the corresponding point in *Image* (Y_{ij} specifies the y -coordinate of *Image* _{ij}).

Contour Visualization Arguments**Z**

A vector or two-dimensional array containing the values to be contoured. If the X and Y arguments are provided, the contour is plotted as a function of the (x, y) locations specified by their contents. Otherwise, the contour is generated as a function of the two-dimensional array index of each element of Z .

X

A vector or two-dimensional array specifying the x -coordinates for the contour surface. If X is a vector, each element of X specifies the x -coordinate for a column of Z (e.g., $X[0]$ specifies the x -coordinate for $Z[0, *]$). If X is a two-dimensional array, each element of X specifies the x -coordinate of the corresponding point in Z (i.e., X_{ij} specifies the x -coordinate for Z_{ij}).

Y

A vector or two-dimensional array specifying the y -coordinates for the contour surface. If Y is a vector, each element of Y specifies the y -coordinate for a row of Z (e.g., $Y[0]$ specifies the y -coordinate for $Z[* , 0]$). If Y is a two-dimensional array, each element of Y specifies the y -coordinate of the corresponding point in Z (Y_{ij} specifies the y -coordinate for Z_{ij}).

Keywords

CONTOUR

Set this keyword to create a contour visualization from the supplied data. By default, the procedure creates an image visualization.

MAP_PROJECTION

Set this keyword to a string specifying the name of an IDL GCTP map projection to use for the initial projection. If this keyword is specified, the allowed map projection keywords are available. [Table 3-48](#) provides the projection names and the allowed keywords associated with them.

These properties are applied to the newly created dataspace, without affecting existing dataspace. If you create an image with IMAP and use this keyword to define a map projection, IDL uses these properties to set up the image's projection. For more information, see [“Registering an Image”](#) in Chapter 15 of the *iTool User's Guide* manual.

Projection Name	Allowed Keywords
UTM	CENTER_LATITUDE, ZONE
State Plane	ZONE
Albers Equal Area	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Lambert Conformal Conic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, TRUE_SCALE_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Polar Stereographic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-48: Map Projections and Their Allowed Keywords

Projection Name	Allowed Keywords
Polyconic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Equidistant Conic A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PARALLEL, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Equidistant Conic B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Transverse Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Stereographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Lambert Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Gnomonic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Orthographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-48: Map Projections and Their Allowed Keywords

Projection Name	Allowed Keywords
Near Side Perspective	SPHERE_RADIUS, HEIGHT, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Sinusoidal	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Equirectangular	SPHERE_RADIUS, CENTER_LONGITUDE, TRUE_SCALE_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Miller Cylindrical	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Van der Grinten	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Hotine Oblique Mercator A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING, HOM_LONGITUDE1, HOM_LATITUDE1, HOM_LONGITUDE2, HOM_LATITUDE2
Hotine Oblique Mercator B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, HOM_AZIM_ANGLE, HOM_AZIM_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
Robinson	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Space Oblique Mercator A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, SOM_INCLINATION, SOM_LONGITUDE, FALSE_EASTING, FALSE_NORTHING, SOM_PERIOD, SOM_RATIO, SOM_FLAG
Space Oblique Mercator B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, SOM_LANDSAT_NUMBER, SOM_LANDSAT_PATH, FALSE_EASTING, FALSE_NORTHING

Table 3-48: Map Projections and Their Allowed Keywords

Projection Name	Allowed Keywords
Alaska Conformal	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, FALSE_EASTING, FALSE_NORTHING
Interrupted Goode	SPHERE_RADIUS
Mollweide	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Interrupted Mollweide	SPHERE_RADIUS
Hammer	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Wagner IV	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Wagner VII	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
Oblated Equal Area	SPHERE_RADIUS, OEA_SHAPEM, OEA_SHAPEN, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING, OEA_ANGLE
Integerized Sinusoidal	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING, IS_ZONES, IS_JUSTIFY

Table 3-48: Map Projections and Their Allowed Keywords

BACKGROUND_COLOR

Set this keyword to an RGB value specifying the color to be used as the background color for the view. The default is [255, 255, 255] (white). The BACKGROUND_COLOR keyword can be used when a tool is being created or when a new visualization is being created in an existing tool with the use of the OVERPLOT, VIEW_NUMBER, or VIEW_NEXT keywords. The background color is applied to the current view. For example, if multiple views have been created with the VIEW_GRID keyword, and the VIEW_NUMBER keyword is used to create a visualization in the second view, use of the BACKGROUND_COLOR keyword would set the background color in the second view only.

DIMENSIONS

Set this keyword to a two-element vector of the form $[width, height]$ to specify the dimensions of the drawing area of the specific tool in units specified by the UNITS keyword. If no value is provided, a default value of one half the screen size is used. The minimum width of the window correlates to the width of the menu bar. The minimum window height is 100 pixels.

GRID_UNITS

Set this keyword to an integer specifying the units for the image or contour grid. This keyword applies only when there is a map projection inserted. It has the following values:

- 0 — None. The image or contour grid is in arbitrary units that are not tied to a map projection. The image or contour will not be warped to the current map projection.
- 1 — Meters. The image or contour grid is in meters, and is tied to a particular map projection. The image or contour will be automatically warped to the current map projection. You are responsible for choosing the dataspace map projection that matches the image or contour's map projection.
- 2 — Degrees. The image or contour grid is in degrees longitude/latitude, and will be automatically warped to the current map projection.

IDENTIFIER

Set this keyword to a named variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LOCATION

Set this keyword to a two-element vector of the form $[x, y]$ to specify the location of the upper left corner of the tool relative to the display screen, in units specified by the UNITS keyword.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the iTool and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

NAME

Set this keyword to a string that specifies the name of this visualization.

NO_SAVEPROMPT

Set this keyword to cause the iTool *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied using the following rules:

- If the tool exists and OVERPLOT is specified, then the style is only applied to the newly-created visualizations within the current view. The current tool style is not updated with the new style, nor is the style applied to any other items within the view.
- Otherwise, if the tool exists and either VIEW_NEXT or VIEW_NUMBER is being used to select a different view, then the style is applied to all items within that view. The current tool style is updated with the new style.
- Otherwise, if a new tool is being created, then the style is applied to all items within all views. The current tool style is updated with the new style.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool and is used for tool-related display purposes only, as the root of the hierarchy shown in the Tool Browser, for example.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created. For example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified, then VIEW_GRID is ignored.

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently selected view before issuing any graphical commands. If the currently selected view is the last one in the layout, then VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless OVERPLOT is set.

[XY]GRIDSTYLE

The index of the linestyle to be used for plot tickmarks and grids (i.e., when [XY]TICKLEN is set to 1.0). See LINSTYLE for a list of linestyles.

[XY]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

[XY]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

[XY]RANGE

Set this keyword to the desired data range of the axis, a two-element vector. The first element is the axis minimum, and the second is the axis maximum.

[XY]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XY]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XY]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XY]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XY]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XY]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether TICKUNITS is specified.

If TICKUNITS are not specified:

The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:

- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Note

For more information, see “[XYZ]TICKFORMAT” on page 4345.

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XY]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XY]RANGE and the number of major tick marks ([XY]MAJOR). The value of this keyword takes precedence over the value set for the [XY]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'] and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

[XY]TICKLAYOUT

Set this keyword to an integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XY]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XY]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XY]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- Numeric
- Years
- Months
- Days
- Hours
- Minutes
- Seconds
- Time — Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- Use the empty string ("") to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the Numeric unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 C.E., respectively.

Note

Note that the singular form of each of the time value strings is also acceptable (e.g., TICKUNITS = "Day" is equivalent to TICKUNITS = "Days").

[XY]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XY]TITLE

Set this keyword to a string representing the title of the specified axis.

Map Projection Keywords:**CENTER_LATITUDE**

Set this keyword to the latitude of the point on the earth's surface to be mapped to the center of the projection plane. Latitude is measured in degrees north of the equator and must be in the range -90 to +90. The default value is zero.

CENTER_LONGITUDE

Set this keyword to the longitude of the point on the earth's surface to be mapped to the center of the map projection. Longitude is measured in degrees east of the Greenwich meridian and must be in the range -360 to +360. The default value is zero.

DATUM

Set this keyword to a scalar string containing the name of the datum to use for the ellipsoid. The default value depends on the projection selected, but is either the Clarke 1866 ellipsoid or a sphere with a radius of 6370.997 kilometers.

[Table 5](#) shows the datums (or spheroids) available for use with the DATUM keyword.

Name	Semimajor Axis (m)	Semiminor Axis (m)
Clarke 1866 (NAD27)	6378206.4	6356583.8

Table 5: Datums Available for Use with the DATUM Keyword

Name	Semimajor Axis (m)	Semiminor Axis (m)
Clarke 1880	6378249.145	6356514.86955
Bessel	6377397.155	6356078.96284
International 1967	6378157.5	6356772.2
International 1909	6378388.0	6356911.94613
WGS 72	6378135.0	6356750.519915
Everest	6377276.3452	6356075.4133
WGS 66	6378145.0	6356759.769356
GRS 1980/WGS 84 (NAD 83)	6378137.0	6356752.31414
Airy	6377563.396	6356256.91
Modified Everest	6377304.063	6356103.039
Modified Airy	6377340.189	6356034.448
Walbeck	6378137.0	6356752.314245
Southeast Asia	6378155.0	6356773.3205
Australian National	6378160.0	6356774.719
Krassovsky	6378245.0	6356863.0188
Hough	6378270.0	6356794.343479
Mercury 1960	6378166.0	6356784.283666
Modified Mercury 1968	6378150.0	6356768.337303
Sphere	6370997.0	6370997.0

Table 5: Datums Available for Use with the DATUM Keyword

Note

For many projections, you can specify your own datum by using either the SEMIMAJOR_AXIS and SEMIMINOR_AXIS keywords or the SPHERE_RADIUS keyword.

FALSE_EASTING

Set this keyword to the false easting value (in meters) to be added to each x -coordinate for the forward transform, or subtracted from each x coordinate for the inverse transform.

FALSE_NORTHING

Set this keyword to the false northing value (in meters) to be added to each y -coordinate for the forward transform, or subtracted from each y -coordinate for the inverse transform.

HEIGHT

Set this keyword to the height (in meters) above the earth's surface for satellite projections.

HOM_AZIM_LONGITUDE

Set this keyword to the longitude in degrees of the central meridian point where the azimuth occurs.

HOM_AZIM_ANGLE

Set this keyword to the azimuth angle, measured in degrees, east of a north-south line that intersects the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LATITUDE1

Set this keyword to the latitude in degrees of the first point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LATITUDE2

Set this keyword to the latitude in degrees of the second point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LONGITUDE1

Set this keyword to the longitude in degrees of the first point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LONGITUDE2

Set this keyword to the longitude in degrees of the second point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

IS_ZONES

Set this keyword to the number of longitudinal zones to include in the projection.

IS_JUSTIFY

Set this keyword to indicate what to do with rows with an odd number of columns. The following values are allowed:

- 0 — Indicates the extra column is on the right of the projection y -axis
- 1 — Indicates the extra column is on the left of the projection y -axis
- 2 — Calculates an even number of columns

LIMIT

Set this keyword to a four-element vector of the form $[latmin, lonmin, latmax, lonmax]$ that specifies the boundaries of the region to be mapped. The points $[lonmin, latmin]$ and $[lonmax, latmax]$ are the longitudes and latitudes of two points diagonal from each other on the region's boundary.

MERCATOR_SCALE

Set this keyword to the scale factor at the central meridian (for the Transverse Mercator projection) or the center of the projection (for the Hotine Oblique Mercator projection). For the Transverse Mercator projection, the default scale is 0.9996.

OEA_ANGLE

Set this keyword to the Oblated Equal Area oval rotation angle in degrees.

OEA_SHAPEM

Set this keyword to the Oblated Equal Area shape parameter m . The value of OEA_SHAPEM determines the horizontal flatness of the oblong region, and is usually set to a value between one and three.

Note

Setting both OEA_SHAPEM and OEA_SHAPEN equal to 2 is equivalent to using the Lambert Azimuthal projection.

OEA_SHAPEN

Set this keyword to the Oblated Equal Area oval shape parameter n . The value of OEA_SHAPEN determines the vertical flatness of the oblong region, and is usually set to a value between one and three.

Note

Setting both OEA_SHAPEM and OEA_SHAPEN equal to 2 is equivalent to using the Lambert Azimuthal projection.

SEMIMAJOR_AXIS

Set this keyword to the length in meters of the semimajor axis of the reference ellipsoid. The default is the semimajor axis length of either the Clarke 1866 datum (6378206.4 meters) or the Sphere radius (6370997.0 meters), depending on the projection.

SEMIMINOR_AXIS

Set this keyword to the length in meters of the semiminor axis of the reference ellipsoid. The default is the semiminor axis length of either the Clarke 1866 datum (6356583.8 meters) or the Sphere radius (6370997.0 meters), depending on the projection.

SOM_INCLINATION

Set this keyword to the orbit inclination angle in degrees of the ascending node, counter-clockwise from the equator.

SOM_LONGITUDE

Set this keyword to the longitude in degrees of the ascending orbit at the equator.

SOM_PERIOD

Set this keyword to the period in minutes of the satellite revolution.

SOM_RATIO

Set this keyword to the Landsat ratio to compensate for confusion at the northern end of orbit. A typical value is 0.5201613.

SOM_FLAG

Set this keyword to the end-of-path flag for Landsat, where 0 is the start and 1 is the end.

SOM_LANDSAT_NUMBER

Set this keyword to the Landsat satellite number.

SOM_LANDSAT_PATH

Set this keyword to the Landsat path number. Use 1 for Landsat 1, 2, and 3; use 2 for Landsat 4, 5 and 6.

SPHERE_RADIUS

Set this keyword to the radius in meters of the reference sphere. The default value is 6370997.

STANDARD_PARALLEL

Set this keyword to the latitude in degrees of the standard parallel along which the scale is true.

STANDARD_PAR1

Set this keyword to the latitude in degrees of the first standard parallel along which the scale is true.

STANDARD_PAR2

Set this keyword to the latitude in degrees of the second standard parallel along which the scale is true.

TRUE_SCALE_LATITUDE

Set this keyword to the latitude in degrees of true scale.

ZONE

Set this keyword to an integer specifying the zone for the UTM projection or the State Plane projection.

Note

For the UTM projection, you can also use the `CENTER_LONGITUDE` and `CENTER_LATITUDE` keywords to set the zone. Internally, the `ZONE` value will be computed from the longitude and latitude.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Examples 2 and 3) or data can be imported via the **File** menu in the iTool window. For detailed information on importing data via the iTool file menu, refer to “[Data Import Methods](#)” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

```
IMAP, MAP_PROJECTION='Lambert'
```

This command opens a new iMap tool with the Lambert Conformal Conic projection loaded (Figure 3-29).

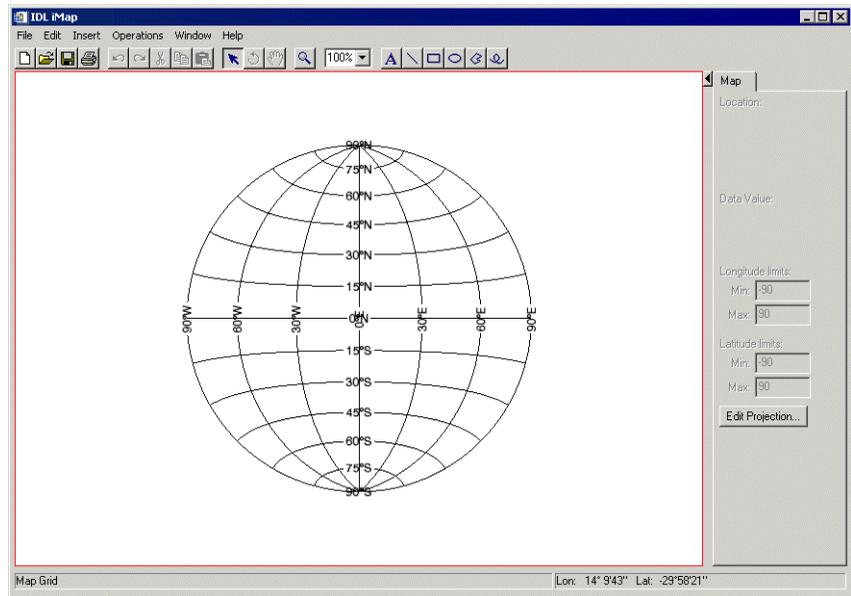


Figure 3-29: iMap Example — Empty Lambert Projection

Example 2

```
file = FILEPATH('avhrr.png', SUBDIRECTORY=['examples','data'])
data = READ_PNG(file, r, g, b)
IMAP, data, LIMIT=[-90,-180,90,180], $
    MAP_PROJECTION='Mollweide', RGB_TABLE=[[r],[g],[b]], $
    IMAGE_TRANSPARENCY=50, GRID_UNITS=2, $
    IMAGE_LOCATION=[-180,-90], IMAGE_DIMENSIONS=[360,180]
```

This series of commands opens a new iMap tool and loads an image of the world, registered in degrees, into a Mollweide map projection (Figure 3-30).

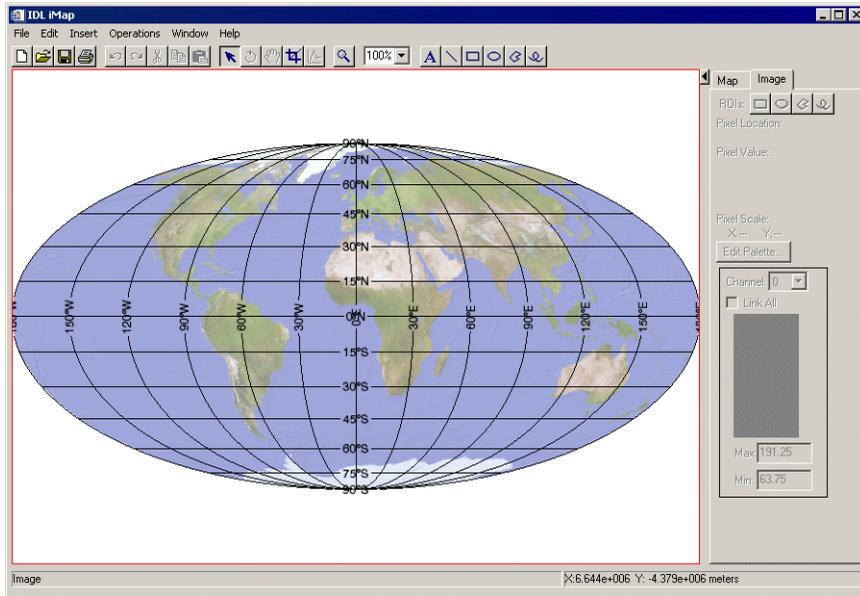


Figure 3-30: iMap Example — World Image with Mollweide Projection

Example 3

This example builds on Example 2, adding contours to the world image in the Mollweide projection.

```
loadct, 39
tv!ct, r, g, b, /get

clons = FINDGEN(360) - 179.5
clats = FINDGEN(180) - 89.5
cdata = SIN(clons/30) # COS(clats/30)
IMAP, cdata, clons, clats, /CONTOUR, /OVERPLOT, $
    RGB_TABLE=[[r],[g],[b]], GRID_UNITS=2, $
    N_LEVELS=10, C_THICK=REPLICATE(2,10)
```

This series of commands plots the contour data atop Example 2's display (Figure 3-30)..

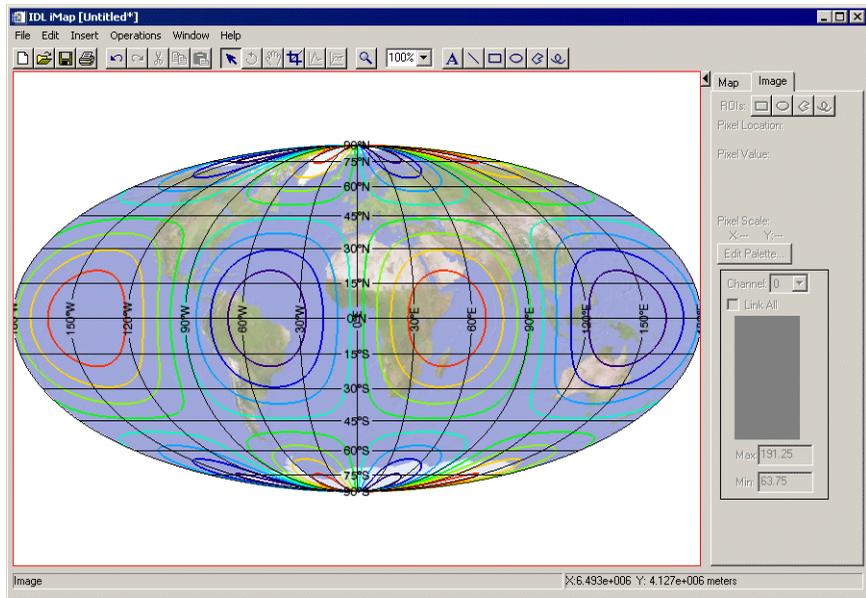


Figure 3-31: iMap Example — Overplotted Contour Data in Rainbow Colors

Version History

Introduced: 6.1

INDGEN

The INDGEN function returns an integer array with the specified dimensions.

Syntax

```
Result = INDGEN(D1 [, ..., D8] [, /BYTE | , /COMPLEX | , /DCOMPLEX | ,  
/DOUBLE | , /FLOAT | , /L64 | , /LONG | , /STRING | , /UINT | , /UL64 | ,  
/ULONG] [, TYPE=value] )
```

Return Value

Each element of the returned integer array is set to the value of its one-dimensional subscript.

Arguments

*D*_{*i*}

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

BYTE

Set this keyword to create a byte array.

COMPLEX

Set this keyword to create a complex, single-precision, floating-point array.

DCOMPLEX

Set this keyword to create a complex, double-precision, floating-point array.

DOUBLE

Set this keyword to create a double-precision, floating-point array.

FLOAT

Set this keyword to create a single-precision, floating-point array.

L64

Set this keyword to create a 64-bit integer array.

LONG

Set this keyword to create a longword integer array.

STRING

Set this keyword to create a string array.

TYPE

The type code to set the type of the result. See the description of the [SIZE](#) function for a list of IDL type codes.

UINT

Set this keyword to create an unsigned integer array.

UL64

Set this keyword to create an unsigned 64-bit integer array.

ULONG

Set this keyword to create an unsigned longword integer array.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the `!CPU` system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Create *I*, a 5-element vector of integer values with each element set to the value of its subscript by entering:

```
I = INDGEN(5)
```

Version History

Introduced: Original

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [LINDGEN](#),
[SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

INT_2D

The INT_2D function computes the double integral of a bivariate function using iterated Gaussian quadrature. The algorithm's transformation data is provided in tabulated form with 15 decimal accuracy.

This routine is written in the IDL language. Its source code can be found in the file `int_2d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = INT_2D( Fxy, AB_Limits, PQ_Limits, Pts [, /DOUBLE] [, /ORDER] )
```

Return Value

Returns a double value containing the integral.

Arguments

Fxy

A scalar string specifying the name of a user-supplied IDL function that defines the bivariate function to be integrated. For dy/dx integration (the default, or if the ORDER keyword is explicitly set to zero), the Fxy function must be able to accept a scalar value for X and a vector for Y, and must return a vector of the same length as Y. For dx/dy integration (if the ORDER keyword is set), the Fxy function must be able to accept a vector for X and a scalar value for Y, and must return a vector of the same length as X.

For example, if we wish to integrate the following function:

$$f(x, y) = e^{-x^2 - y^2}$$

We define a function FXY to express this relationship in the IDL language:

```
FUNCTION fxy, X, Y
  RETURN, EXP(-X^2. -Y^2.)
END
```

AB_Limits

A two-element vector containing the lower (A) and upper (B) limits of integration with respect to the variable x .

PQ_Limits

A scalar string specifying the name of a user-supplied IDL function that defines the lower (P(x)) and upper (Q(x)) limits of integration with respect to the variable y. The function must accept x and return a two-element vector result.

For example, we might write the following IDL function to represent the limits of integration with respect to y:

```
FUNCTION PQ_limits, X
    RETURN, [-SQRT(16.0 - X^2), SQRT(16.0 - X^2)]
END
```

Pts

The number of transformation points used in the computation. Possible values are: 6, 10, 20, 48, or 96.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ORDER

A scalar value of either 0 or 1. If set to 0, the integral is computed using a dy-dx order of integration. If set to 1, the integral is computed using a dx-dy order of integration.

Examples

Example 1

Compute the double integral of the bivariate function.

$$I = \int_{x=0.0}^{x=2.0} \int_{y=0.0}^{y=x^2} y \cdot \cos(x^5) dy dx$$

```
; Define the Fxy function.
FUNCTION Fxy, x, y
    RETURN, y*COS(x^5)
END
```

```

; Define the limits of integration for y as a function of x:
FUNCTION PQ_Limits, x
    RETURN, [0.0, x^2]
END

; Define limits of integration for x:
AB_Limits = [0.0, 2.0]

; Using the function and limits defined above, integrate with 48
; and 96 point formulas using a dy-dx order of integration and
; double-precision arithmetic:
PRINT, INT_2D('Fxy', AB_Limits, 'PQ_Limits', 48, /DOUBLE)
PRINT, INT_2D('Fxy', AB_Limits, 'PQ_Limits', 96, /DOUBLE)

```

INT_2D with 48 transformation points yields: 0.055142668

INT_2D with 96 transformation points yields: 0.055142668

Example 2

Compute the double integral of the bivariate function:

$$I = \int_{x=0.0}^{x=2.0} \int_{y=0.0}^{y=x^2} y \cdot \cos(x^5) dx dy$$

```

; Define the Fxy function.
FUNCTION Fxy, x, y
    RETURN, y*COS(x^5)
END

; Define the limits of integration for y as a function of x:
FUNCTION PQ_Limits, y
    RETURN, [sqrt(y), 2.0]
END

; Define limits of integration for x:
AB_Limits = [0.0, 4.0]

; Using the function and limits defined above, integrate with 48
; and 96 point formulas using a dy-dx order of integration and
; double-precision arithmetic:
PRINT, INT_2D('Fxy', AB_Limits, 'PQ_Limits', 48, /DOUBLE, /ORDER)
PRINT, INT_2D('Fxy', AB_Limits, 'PQ_Limits', 96, /DOUBLE, /ORDER)

```

INT_2D with 48 transformation points yields: 0.055142678

INT_2D with 96 transformation points yields: 0.055142668

The exact solution (7 decimal accuracy) is: 0.055142668

Version History

Introduced: Pre 4.0

See Also

[INT_3D](#), [INT_TABULATED](#), [QROMB](#), [QROMO](#), [QSIMP](#)

INT_3D

The INT_3D function computes the integral of a trivariate function using iterated Gaussian quadrature. The algorithm's transformation data is provided in tabulated form with 15 decimal accuracy.

This routine is written in the IDL language. Its source code can be found in the file `int_3d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = INT_3D( Fxyz, AB_Limits, PQ_Limits, UV_Limits, Pts [, /DOUBLE] )
```

Return Value

Returns the triple integral.

Arguments

Fxyz

A scalar string specifying the name of a user-supplied IDL function that defines the trivariate function to be integrated. The function must accept X, Y, and Z, and return a scalar result.

For example, if we wish to integrate the following function:

$$f(x, y, z) = z \cdot (x^2 + y^2 + z^2)^{3/2}$$

We define a function FXY to express this relationship in the IDL language:

```
FUNCTION fxyz, X, Y, Z
  RETURN, z*(x^2+y^2+z^2)^1.5
END
```

AB_Limits

A two-element vector containing the lower (A) and upper (B) limits of integration with respect to the variable x .

PQ_Limits

A scalar string specifying the name of a user-supplied IDL function that defines the lower ($P(x)$) and upper ($Q(x)$) limits of integration with respect to the variable y . The function must accept x and return a two-element vector result.

For example, we might write the following IDL function to represent the limits of integration with respect to y :

```
FUNCTION PQ_limits, X
    RETURN, [-SQRT(4.0 - X^2), SQRT(4.0 - X^2)]
END
```

UV_Limits

A scalar string specifying the name of a user-supplied IDL function that defines the lower ($U(x,y)$) and upper ($V(x,y)$) limits of integration with respect to the variable z . The function must accept x and y and return a two-element vector result.

For example, we might write the following IDL function to represent the limits of integration with respect to z :

```
FUNCTION UV_limits, X, Y
    RETURN, [0, SQRT(4.0 - X^2 - Y^2)]
END
```

Pts

The number of transformation points used in the computation. Possible values are: 6, 10, 20, 48, or 96.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

Compute the triple integral of the trivariate function

$$I = \int_{x=-2}^{x=2} \int_{y=-\sqrt{4-x^2}}^{y=\sqrt{4-x^2}} \int_{z=0}^{z=\sqrt{4-x^2-y^2}} z \cdot (x^2 + y^2 + z^2)^{3/2} dz dy dx$$

Using the functions and limits defined above, integrate with 10, 20, 48, and 96 point formulas (using double-precision arithmetic):

```
PRINT, INT_3D('Fxyz', [-2.0, 2.0], 'PQ_Limits', 'UV_Limits', 10,$
/D)
PRINT, INT_3D('Fxyz', [-2.0, 2.0], 'PQ_Limits', 'UV_Limits', 20,$
/D)
PRINT, INT_3D('Fxyz', [-2.0, 2.0], 'PQ_Limits', 'UV_Limits', 48,$
/D)
PRINT, INT_3D('Fxyz', [-2.0, 2.0], 'PQ_Limits', 'UV_Limits', 96,$
/D)
```

INT_3D with 10 transformation points yields: 57.444248

INT_3D with 20 transformation points yields: 57.446201

INT_3D with 48 transformation points yields: 57.446265

INT_3D with 96 transformation points yields: 57.446266

The exact solution (6 decimal accuracy) is: 57.446267

Version History

Introduced: Pre 4.0

See Also

[INT_2D](#), [INT_TABULATED](#), [QROMB](#), [QROMO](#), [QSIMP](#)

INT_TABULATED

The INT_TABULATED function integrates a tabulated set of data $\{ x_i, f_i \}$ on the closed interval $[\text{MIN}(x), \text{MAX}(x)]$, using a five-point Newton-Cotes integration formula.

Warning

Data that is highly oscillatory requires a sufficient number of samples for an accurate integral approximation.

This routine is written in the IDL language. Its source code can be found in the file `int_tabulated.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = INT_TABULATED(*X*, *F* [, /DOUBLE] [, /SORT])

Return Value

Returns the area under the curve represented by the function.

Arguments

X

The tabulated single- or double-precision floating-point x -value data. Data may be irregularly gridded and in random order. (If the data is randomly ordered, set the SORT keyword.)

Warning

Each X value must be unique; if duplicate X values are detected, the routine will exit and display a warning message.

F

The tabulated single- or double-precision floating-point f -value data. Upon input to the function, x_i and f_i must have corresponding indices for all values of i . If x is reordered, f is also reordered.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

SORT

Set this keyword to sort the tabulated x -value data into ascending order. If SORT is set, both x and f values are sorted.

Examples

Define 11 x -values on the closed interval [0.0 , 0.8]:

```
X = [0.0, .12, .22, .32, .36, .40, .44, .54, .64, .70, .80]
```

Define 11 f -values corresponding to x_i :

```
F = [0.200000, 1.30973, 1.30524, 1.74339, 2.07490, 2.45600, $
      2.84299, 3.50730, 3.18194, 2.36302, 0.231964]
result = INT_TABULATED(X, F)
```

In this example, the f -values are generated from a known function

$$f = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

which allows the determination of an exact solution. A comparison of methods yields the following results:

- The Multiple Application Trapezoid Method yields: 1.5648
- The Multiple Application Simpson's Method yields: 1.6036
- INT_TABULATED yields: 1.6271

The exact solution (4 decimal accuracy) is: 1.6405

Version History

Introduced: Pre 4.0

See Also

[INT_2D](#), [INT_3D](#), [QROMB](#), [QROMO](#), [QSIMP](#)

INTARR

The INTARR function returns an integer vector or array.

Syntax

```
Result = INTARR(  $D_1$  [, ...,  $D_8$ ] [, /NOZERO] )
```

Return Value

Returns the integer array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, INTARR sets every element of the result to zero. If NOZERO is nonzero, this zeroing is not performed and INTARR executes faster.

Examples

Create I, a 3-element by 3-element integer array with each element set to 0 by entering:

```
I = INTARR(3, 3)
```

Version History

Introduced: Original

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [LON64ARR](#),
[LONARR](#), [MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

INTERPOL

The INTERPOL function performs linear, quadratic, or spline, interpolation on vectors with a regular or irregular grid.

This routine is written in the IDL language. Its source code can be found in the file `interp.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

For regular grids: $Result = \text{INTERPOL}(V, N [, /LSQUADRATIC] [, /QUADRATIC] [, /SPLINE])$

For irregular grids: $Result = \text{INTERPOL}(V, X, U [, /LSQUADRATIC] [, /QUADRATIC] [, /SPLINE])$

Return Value

The result is a single- or double-precision floating-point vector, or a complex vector if the input vector is complex.

Arguments

V

An input vector of any type except string.

N

The number of points in the result when both input and output grids are regular. The abscissa values for the output grid will contain the same endpoints as the input.

X

The abscissa values for *V*, in the irregularly-gridded case. *X* must have the same number of elements as *V*, and the values *must* be strictly ascending or descending.

U

The abscissa values for the result. The result will have the same number of elements as *U*. *U* does not need to be monotonic.

Keywords

LSQUADRATIC

If set, interpolate using a least squares quadratic fit to the equation $y = a + bx + cx^2$, for each 4 point neighborhood $(x[i-1], x[i], x[i+1], x[i+2])$ surrounding the interval of the interpolate, $x[i] \leq u < x[i+1]$.

QUADRATIC

If set, interpolate by fitting a quadratic $y = a + bx + cx^2$, to the three point neighborhood $(x[i-1], x[i], x[i+1])$ surrounding the interval $x[i] \leq u < x[i+1]$.

SPLINE

If set, interpolate by fitting a cubic spline to the 4 point neighborhood $(x[i-1], x[i], x[i+1], x[i+2])$ surrounding the interval, $x[i] \leq u < x[i+1]$.

Note

If LSQUADRATIC or QUADRATIC or SPLINE is not set, the default is to use linear interpolation.

Examples

Create a floating-point vector of 61 elements in the range [-3, 3].

```
X = FINDGEN(61)/10 - 3

; Evaluate V[x] at each point:
V = SIN(X)

; Define X-values where interpolates are desired:
U = [-2.50, -2.25, -1.85, -1.55, -1.20, -0.85, -0.50, -0.10, $
     0.30, 0.40, 0.75, 0.85, 1.05, 1.45, 1.85, 2.00, 2.25, 2.75 ]

; Interpolate:
result = INTERPOL(V, X, U)

; Plot the function:
PLOT, X, V

; Plot the interpolated values:
OPLOT, U, result
```

Version History

Introduced: Original

See Also

[BILINEAR](#), [INTERPOLATE](#), [KRIG2D](#)

INTERPOLATE

The INTERPOLATE function returns an array of linear, bilinear or trilinear interpolates, depending on the dimensions of the input array *P*.

Interpolates outside the bounds of *P* can be set to a user-specified value by using the MISSING keyword.

Syntax

```
Result = INTERPOLATE( P, X [, Y [, Z]] [, CUBIC=value{-1 to 0}] [, /GRID]
  [, MISSING=value] )
```

Return Value

Linear interpolates are returned in the one-dimensional case, bilinear in the two-dimensional case and trilinear interpolates in the three-dimensional case. The returned array has the same type as *P* and its dimensions depend on those of the location parameters *X*, *Y*, and *Z*, as explained below.

Arguments

P

The array of data values. *P* can be an array of any dimensions. Interpolation occurs in the *M* rightmost indices of *P*, where *M* is the number of interpolation arrays.

X, Y, Z

Arrays of numeric type containing the locations for which interpolates are desired. For linear interpolation (*P* is a vector), the result has the same dimensions as *X*. The *i*-th element of the result is *P* interpolated at location X_i . The *Y* and *Z* parameters should be omitted.

For bilinear interpolation *Z* should not be present.

Note

INTERPOLATE considers location points with values between zero and *n*, where *n* is the number of values in the input array *P*, to be valid. Location points outside this range are considered missing data. Location points *x* in the range $n-1 \leq x < n$ return the last data value in the array *P*.

If the keyword GRID is not set, all location arrays must have the same number of elements. See the description of the GRID keyword below for more details on how interpolates are computed from P and these arrays.

Keywords

CUBIC

Set this keyword to a value between -1 and 0 to use the cubic convolution interpolation method with the specified value as the interpolation parameter. Setting this keyword equal to a value greater than zero specifies a value of -1 for the interpolation parameter. Park and Schowengerdt (see reference below) suggest that a value of -0.5 significantly improves the reconstruction properties of this algorithm.

Cubic convolution is an interpolation method that closely approximates the theoretically optimum sinc interpolation function using cubic polynomials. According to sampling theory, details of which are beyond the scope of this document, if the original signal, f , is a band-limited signal, with no frequency component larger than ω_0 , and f is sampled with spacing less than or equal to $1/(2\omega_0)$, then f can be reconstructed by convolving with a sinc function: $\text{sinc}(x) = \sin(\pi x) / (\pi x)$.

The number of neighboring points used varies according to the dimension:

- 1-dimensional: 4 points
- 2-dimensional: 16 points
- 3-dimensional: not supported

Note

Cubic convolution interpolation is significantly slower than bilinear interpolation. Also note that cubic interpolation is not supported for three-dimensional data.

For further details see:

Rifman, S.S. and McKinnon, D.M., "Evaluation of Digital Correction Techniques for ERTS Images; Final Report", Report 20634-6003-TU-00, TRW Systems, Redondo Beach, CA, July 1974.

S. Park and R. Schowengerdt, 1983 "Image Reconstruction by Parametric Cubic Convolution", Computer Vision, Graphics & Image Processing 23, 256.

GRID

The GRID keyword controls how the location arrays specify where interpolates are desired. This keyword has no effect in the case of linear interpolation.

If GRID is not set: The location arrays, X , Y , and, if present, Z must have the same number of elements. The result has the same structure and number of elements as X .

In the case of bilinear interpolation, the result is obtained as follows: Let $l = \lfloor X_i \rfloor$ and $k = \lfloor Y_i \rfloor$. Element i of the result is computed by interpolating between $P(l, k)$, $P(l+1, k)$, $P(l, k+1)$, and $P(l+1, k+1)$ to obtain the estimated value at (X_i, Y_i) . Trilinear interpolation is a direct extension of the above.

If GRID is set: Let N_x be the number of elements in X , let N_y be the number of elements in Y , and N_z be the number of elements in Z . The result has dimensions (N_x, N_y) for bilinear interpolation, and (N_x, N_y, N_z) for trilinear interpolation. For bilinear interpolation, element (i, j) of the result contains the value of P interpolated at position (X_i, Y_i) . For trilinear interpolation, element (i, j, k) of the result is P interpolated at (X_i, Y_i, Z_i) .

MISSING

The value to return for elements outside the bounds of P . If this keyword is not specified, interpolated positions that fall outside the bounds of the array P —that is, elements of the X , Y , or Z arguments that are either less than zero or greater than the largest subscript in the corresponding dimension of P —are set equal to the value of the nearest element of P .

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

The example below computes bilinear interpolates with the keyword GRID set:

```
p = FINDGEN(4,4)
PRINT, INTERPOLATE(p, [.5, 1.5, 2.5], [.5, 1.5, 2.5], /GRID)
```

and prints the 3 by 3 array:

```
2.50000  3.50000  4.50000
6.50000  7.50000  8.50000
10.50000 11.50000 12.50000
```

corresponding to the locations:

```
(.5,.5), (1.5, .5), (2.5, .5),
(.5,1.5), (1.5, 1.5), (2.5, 1.5),
(.5,2.5), (1.5, 2.5), (2.5, 2.5)
```

Another example computes interpolates, with GRID not set and a parameter outside the bounds of *P*:

```
PRINT, INTERPOLATE(p, [.5, 1.5, 2.5, 3.1], [.5, 1.5, 2.5, 2])
```

and prints the result:

```
2.50000  7.50000  12.5000  11.0000
```

corresponding to the locations (.5,.5), (1.5, 1.5), (2.5, 2.5) and (3.1, 2.0). Note that the last location is outside the bounds of *P* and is set from the value of the last column.

The following command uses the MISSING keyword to set such values to -1:

```
PRINT, INTERPOLATE(p, [.5, 1.5, 2.5, 3.1], [.5, 1.5, 2.5, 2], $
MISSING = -1)
```

and gives the result:

```
2.50000  7.50000  12.5000  -1.00000
```

Version History

Introduced: Pre 4.0

See Also

[BILINEAR](#), [INTERPOL](#), [KRIG2D](#)

INTERVAL_VOLUME

The INTERVAL_VOLUME procedure is used to generate a tetrahedral mesh from volumetric data. The generated mesh spans the portion of the volume where the volume data samples fall between two constant data values. This can also be thought of as a mesh constructed to fill the volume between two isosurfaces which are drawn according to the two supplied constant data values. The algorithm is very similar to the ISOSURFACE algorithm and expands upon the SHADE_VOLUME algorithm. A topologically-consistent tetrahedral mesh is returned by decomposing the volume into oriented tetrahedra. This also allows the algorithm to find the interval volume of any tetrahedral mesh.

If an auxiliary array is provided, its data is interpolated onto the output vertices and is returned. This auxiliary data array may have multiple values at each vertex. Any size-leading dimension is allowed as long as the number of values in the subsequent dimensions matches the number of elements in the input data array.

For more information on the INTERVAL_VOLUME algorithm, see the paper, “Interval Volume Tetrahedrization”, Nielson and Sung, *Proceedings: IEEE Visualization*, 1997.

Syntax

```
INTERVAL_VOLUME, Data, Value0, Value1, Outverts, Outconn
  [, AUXDATA_IN=array, AUXDATA_OUT=variable]
  [, GEOM_XYZ=array, TETRAHEDRA=array]
  [, PROGRESS_CALLBACK=string] [, PROGRESS_METHOD=string]
  [, PROGRESS_OBJECT=objref] [, PROGRESS_PERCENT=percent{0 to 100}]
  [, PROGRESS_USERDATA=value]
```

Arguments

Data

Input three-dimensional array of scalars that define the volume data.

Value0

Input scalar iso-value. This value specifies one of the limits for the interval volume. The generated interval volume encloses all volume samples between and including Value0 and Value1. Value0 may be greater than or less than Value1, but the two values may not be exactly equal. This value also cannot be a NaN, but can be +/- INF.

Value1

Input scalar iso-value. This value specifies the other limit for the interval volume. The generated interval volume encloses all volume samples between and including *Value0* and *Value1*. *Value1* may be greater than or less than *Value0*, but the two values may not be exactly equal. This value also cannot be a NaN, but can be +/- INF.

Outverts

A named variable to contain an output [3, n] array of floating point vertices making up the tetrahedral mesh.

Outconn

A named variable to contain an output array of tetrahedral mesh connectivity values. This array is one-dimensional and consists of a series of four vertex indices, where each group of four indices describes a tetrahedron. The connectivity values are indices into the vertex array returned in *Outverts*. If no tetrahedra are extracted, this argument returns the array [-1].

Keywords

AUXDATA_IN

This keyword defines the input array of auxiliary data with trailing dimensions being the number of values in *Data*.

Note

If you specify the AUXDATA_IN then you must specify AUXDATA_OUT.

AUXDATA_OUT

Set this keyword to a named variable that will contain an output array of auxiliary data sampled at the locations in *Outverts*.

Note

If you specify AUXDATA_OUT then you must specify AUXDATA_IN.

GEOM_XYZ

This keyword defines a [3, n] input array of vertex coordinates (one for each value in the *Data* array). This array is used to define the spatial location of each scalar. If this

keyword is omitted, *Data* must be a three-dimensional array and the scalar locations are assumed to be on a uniform grid.

Note

If you specify GEOM_XYZ then you must specify TETRAHEDRA.

PROGRESS_CALLBACK

Set this keyword to a scalar string containing the name of the IDL function that the INTERVAL_VOLUME procedure calls at PROGRESS_PERCENT intervals as it generates the interval volume.

The PROGRESS_CALLBACK function returns a zero to signal INTERVAL_VOLUME to stop generating the interval volume. This causes INTERVAL_VOLUME to return a single vertex and a connectivity array of [-1], which specifies an empty mesh. If the callback function returns any non-zero value, INTERVAL_VOLUME continues to generate the interval volume.

The PROGRESS_CALLBACK function must specify a single argument, *Percent*, which INTERVAL_VOLUME sets to an integer between 0 and 100, inclusive.

The PROGRESS_CALLBACK function may specify an optional USERDATA keyword parameter, which INTERVAL_VOLUME sets to the variable provided in the PROGRESS_USERDATA keyword.

The following code shows an example of a progress callback function:

```
FUNCTION myProgressCallback, $
    percent, USERDATA = myStruct

    oProgressBar = myStruct.oProgressBar

    ; This method updates the progress bar
    ; graphic and returns TRUE if the user
    ; has NOT pressed the cancel button.
    keepGoing = oProgressBar->$
        UpdateProgressValue(percent)

    RETURN, keepGoing

END
```

PROGRESS_METHOD

Set this keyword to a scalar string containing the name of a function method that the INTERVAL_VOLUME procedure calls at PROGRESS_PERCENT intervals as it generates the interval. If this keyword is set, then the PROGRESS_OBJECT keyword

must be set to an object reference that is an instance of a class that defines the specified method.

The `PROGRESS_METHOD` function method callback has the same specification as the callback described in the `PROGRESS_CALLBACK` keyword, except that it is defined as an object class method:

```
FUNCTION myClass::myProgressCallback, $
    percent, USERDATA = myStruct
```

PROGRESS_OBJECT

Set this keyword to an object reference that is an instance of a class that defines the method specified with the `PROGRESS_METHOD` keyword. If this keyword is set, then the `PROGRESS_METHOD` keyword must also be set.

PROGRESS_PERCENT

Set this keyword to a scalar in the range [1, 100] to specify the interval between invocations of the callback function. The default value is 5 and IDL silently clamps other values to the range [1, 100].

For example, a value of 5 (the default) specifies `INTERVAL_VOLUME` will call the callback function when the interval volume process is 0% complete, 5% complete, 10% complete, ..., 95% complete, and 100% complete.

PROGRESS_USERDATA

Set this property to any IDL variable that `INTERVAL_VOLUME` passes to the callback function in the callback function's `USERDATA` keyword parameter. If this keyword is specified, then the callback function must be able to accept keyword parameters.

TETRAHEDRA

This keyword defines an input array of tetrahedral connectivity values. If this array is not specified, the connectivity is assumed to be a rectilinear grid over the input three-dimensional array. If this keyword is specified, the input data array need not be a three-dimensional array. Each tetrahedron is represented by four values in the connectivity array. Every four values in the array correspond to the vertices of a single tetrahedron.

Note

If you specify `TETRAHEDRA` then you must specify `GEOM_XYZ`.

Examples

The following example generates an interval volume and displays the surface of the volume:

```
RESTORE, FILEPATH('clouds3d.dat', $
  SUBDIRECTORY=['examples','data'])
INTERVAL_VOLUME, rain, 0.1, 0.6, verts, conn
conn = TETRA_SURFACE(verts, conn)
oRain = OBJ_NEW('IDLgrPolygon', verts, POLYGONS=conn, $
  COLOR=[255,255,255], SHADING=1)
XOBJVIEW, oRain, BACKGROUND=[150,200,255]
```

Version History

Introduced: 5.5

PROGRESS_CALLBACK, PROGRESS_METHOD, PROGRESS_OBJECT,
PROGRESS_PERCENT, and PROGRESS_USERDATA keywords added: 6.0

See Also

[ISOSURFACE](#), [SHADE_VOLUME](#), [XVOLUME](#)

INVERT

The INVERT function uses the Gaussian elimination method to compute the inverse of a square array. Errors from singular or near-singular arrays are accumulated in the optional *Status* argument.

Note

If you are working with complex inputs, instead use the LA_INVERT function.

Syntax

Result = INVERT(*Array* [, *Status*] [, /DOUBLE])

Return Value

The result is a single- or double-precision array of floating or complex values.

Arguments

Array

The array to be inverted. *Array* must have two dimensions of equal size (i.e., a square array) and can be of any type except string. Note that the resulting array will be composed of single- or double-precision floating-point or complex values, depending on whether the DOUBLE keyword is set.

Status

A named variable to receive the status of the operation. Possible status values are:

- 0 = Successful completion.
- 1 = Singular array (which indicates that the inversion is invalid).
- 2 = Warning that a small pivot element was used and that significant accuracy was probably lost.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```
; Create an array A:  
A = [[ 5.0, -1.0, 3.0], $  
      [ 2.0,  0.0, 1.0], $  
      [ 3.0,  2.0, 1.0]]  
result = INVERT(A)
```

```
; We can check the accuracy of the inversion by multiplying the  
; inverted array by the original array. The result should be a 3 x  
; 3 identity array.  
PRINT, result # A
```

IDL prints:

```
1.00000      0.00000      0.00000  
0.00000      1.00000      0.00000  
0.00000  9.53674e-07      1.00000
```

Version History

Introduced: Original

See Also

[COND](#), [DETERM](#), [LA_INVERT](#), [REVERSE](#), [ROTATE](#), [TRANSPOSE](#)

IOCTL

The IOCTL function provides a thin wrapper over the UNIX `ioctl(2)` system call. IOCTL performs special functions on the specified file. The set of functions actually available depends on your version of UNIX and the type of file (tty, tape, disk file, etc.) referred to.

To use IOCTL, read the C programmer's documentation describing the `ioctl(2)` function for the desired device and convert all constants and data to their IDL equivalents.

Syntax

```
Result = IOCTL( File_Unit [, Request, Arg] [, /BY_VALUE] [, /MT_OFFLINE]
  [, /MT_REWIND] [, MT_SKIP_FILE=[-]number_of_files]
  [, MT_SKIP_RECORD=[-]number_of_records] [, /MT_WEOF]
  [, /SUPPRESS_ERROR] )
```

Return Value

The value returned by the system `ioctl` function is returned as the value of the IDL IOCTL function.

Arguments

File_Unit

The IDL logical unit number (LUN) for the open file on which the `ioctl` request is made.

Request

A longword integer that specifies the `ioctl` request code. These codes are usually contained in C language header files provided by the operating system, and are not generally portable between UNIX versions. If one of the "MT" keywords is used, this argument can be omitted.

Arg

A named variable through which data is passed to and from `ioctl`. IOCTL requests usually request data from the system or supply the system with information. The user must make *Arg* the correct type and size. Errors in typing or sizing *Arg* can corrupt

the IDL address space and/or make IDL crash. If one of the MT keywords is used, this argument can be omitted.

Keywords

Note that the keywords below that start with “MT” can be used to issue commonly used magnetic tape `ioctl()` calls. When these keywords are used, the *Request* and *Arg* arguments are ignored and can be omitted. Magnetic tape operations not available via these keywords can still be executed by supplying the appropriate *Request* and *Arg* values. When issuing magnetic tape IOCTL calls, be aware that different devices have different rules for which `ioctl` calls are allowed, and when. The documentation for your computer system explains those rules.

BY_VALUE

If this keyword is set, *Arg* is converted to a scalar longword and this longword is passed by value. Normally, *Arg* is passed to `ioctl` by reference (i.e., by address).

MT_OFFLINE

Set this keyword to rewind and unload a tape.

MT_REWIND

Set this keyword to rewind a tape.

MT_SKIP_FILE

Use this keyword to skip files on a tape. A positive value skips forward that number of files. A negative value skips backward.

MT_SKIP_RECORD

Use this keyword to skip records on tape. A positive value skips forward that number of files. A negative value skips backward.

MT_WEOF

Set this keyword to write an EOF (“tape mark”) on the tape at the current location.

SUPPRESS_ERROR

Set this keyword to log errors quietly and cause a value of -1 to be returned. The default is for IDL to notice any failures associated with the use of `ioctl` and issue the appropriate IDL error and halt execution.

Examples

The following example prints the size of the terminal being used by the current IDL session. It is known to work under SunOS 4.1.2. Changes may be necessary for other operating systems or even other versions of SunOS.

```

; Variable to receive result. This structure is described in
; Section 4 of the SunOS manual pages under termios(4):
winsize = { row:0, col:0, xpixel:0, ypixel:0 }

; The request code for obtaining the tty size, as determined by
; reading the termios(4) documentation, and reading the system
; include files in the /usr/include/sys directory:
TIOCGWINSZ = 1074295912L

; Make the information request. -1 is the IDL logical file unit for
; the standard output:
ret = IOCTL(-1, TIOCGWINSZ, winsize)

; Output the results:
PRINT,winsize.row, winsize.col, $
    format='("TTY has ", I0," rows and ", I0," columns.")'
```

The following points should be noted in this example:

- Even though we only want the number of rows and columns, we must include all the fields required by the `TIOCGWINSZ ioctl` in the `winsize` variable (as documented in the `termio(4)` manual page). Not providing a large enough result buffer would cause IDL's memory to be corrupted.
- The value of `TIOCGWINSZ` was determined by examining the system header files provided in the `/usr/include/sys` directory. Such values are not always portable between major operating system releases.

Version History

Introduced: Pre 4.0

See Also

[OPEN](#)

IPlot

The IPlot procedure creates an iTool and the associated user interface (UI) configured to display and manipulate plot data.

Note

If no arguments are specified, the IPlot procedure creates an empty Plot tool.

This routine is written in the IDL language. Its source code can be found in the file `iplot.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

Cartesian

IPlot, [X,] Y

or

IPlot, X, Y, Z

Polar

IPlot[, R], Theta, /POLAR

iTool Common Keywords: [, BACKGROUND_COLOR=*value*]
 [, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]]
 [, MACRO_NAMES=*string* or *string array*] [, NAME=*string*]
 [, /NO_SAVEPROMPT] [, OVERPLOT=*iToolID*] [, STYLE_NAME=*string*]
 [, TITLE=*string*] [, VIEW_GRID=[*columns*, *rows*]] [, /VIEW_NEXT]
 [, VIEW_NUMBER=*integer*]

Plot Keywords: [, CLIP_PLANES=*array*] [, COLOR = *RGB vector*]
 [, ERRORBAR_COLOR=*vector*] [, ERRORBAR_CAPSIZE=*float* {0.0 to 1.0}]
 [, /FILL_BACKGROUND] [, FILL_COLOR=*vector*] [, FILL_LEVEL=*value*]
 [, /HIDE] [, /HISTOGRAM] [, LINESSTYLE=*integer*] [, MAX_VALUE=*value*]
 [, MIN_VALUE=*value*] [, NSUM=*value*] [, /POLAR]
 [, RGB_TABLE=*byte array*] [, /SCATTER] [, SYM_COLOR=*RGB vector*]
 [, SYM_INCREMENT=*integer*] [, SYM_INDEX=*integer*]
 [, SYM_SIZE=*float* {0.0 to 1.0}] [, SYM_THICK=*float* {1.0 to 10.0}]
 [, THICK=*float* {1.0 to 10.0}] [, TRANSPARENCY=*float* {0.0 to 100.0}]
 [, /USE_DEFAULT_COLOR] [, VERT_COLORS=*byte vector*]
 [, /XY_SHADOW] [, /[XYZ]_ERRORBARS] [, /[XYZ]_LOG]
 [, [XYZ]ERROR=*vector* or *array*] [, /XZ_SHADOW] [, /YZ_SHADOW]

Axis Keywords: [, [XYZ]GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]
 [, [XYZ]MAJOR=*integer*] [, [XYZ]MINOR=*integer*]
 [, [XYZ]RANGE=[*min, max*] [, [XYZ]SUBTICKLEN=*ratio*]
 [, [XYZ]TEXT_COLOR=*RGB vector*]
 [, [XYZ]TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
 [, [XYZ]TICKFONT_SIZE=*integer*]
 [, [XYZ]TICKFONT_STYLE={0 | 1 | 2 | 3}]
 [, [XYZ]TICKFORMAT=*string or string array*]
 [, [XYZ]TICKINTERVAL=*value*] [, [XYZ]TICKLAYOUT={0 | 1 | 2}]
 [, [XYZ]TICKLEN=*value*] [, [XYZ]TICKNAME=*string array*]
 [, [XYZ]TICKUNITS=*string*] [, [XYZ]TICKVALUES=*vector*]
 [, [XYZ]TITLE=*string*]

Arguments

R

If the POLAR keyword is set, *R* is a vector representing the radius of the polar plot. If *R* is specified, *Theta* is plotted as a function of *R*. If *R* is not specified, *Theta* is plotted as a function of the vector index of *Theta*.

Theta

If the POLAR keyword is set, *Theta* is a vector representing the angle (in radians) of the polar plot.

X

A vector representing the *x*-coordinates of the plot.

Y

A vector or a two-dimensional array. If *Y* is:

- a vector, it represents the *y*-coordinates of the plot. If *X* is not specified, *Y* is plotted as a function of the vector index of *Y*. If *X* is specified, *Y* is plotted as a function of *X*.
- a 2-by-*n* array, *Y*[0, *] represents the *x*-coordinates and *Y*[1, *] represents the *y*-coordinates of the plot.
- a 3-by-*n* array, *Y*[0, *] represents the *x*-coordinates, *Y*[1, *] represents the *y*-coordinates, and *Y*[2, *] represents the *z*-coordinates of the plot.

Z

A vector representing the z -coordinates of the plot.

Keywords

Note

Keywords to the IPLOT routine that correspond to the names of *registered properties* of the iPlot tool must be specified in full, without abbreviation.

BACKGROUND_COLOR

Set this keyword to an RGB value specifying the color to be used as the background color for the view. The default is [255, 255, 255] (white). The BACKGROUND_COLOR keyword can be used when a tool is being created or when a new visualization is being created in an existing tool with the use of the OVERPLOT, VIEW_NUMBER or VIEW_NEXT keywords. The background color is applied to the current view. For example, if multiple views have been created with the VIEW_GRID keyword, and the VIEW_NUMBER keyword is used to create a visualization in the second view, use of the BACKGROUND_COLOR keyword would set the background color in the second view only.

CLIP_PLANES

Set this keyword to an array of dimensions [4, N] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COLOR

Set this keyword to an RGB value specifying the color to be used as the foreground color for this plot. The default is [0, 0, 0] (black).

DIMENSIONS

Set this keyword to a two-element vector of the form $[width, height]$ to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

ERRORBAR_COLOR

Set this keyword to an RGB value specifying the color for the error bar. The default value is $[0, 0, 0]$ (black).

ERRORBAR_CAPSIZE

Set this keyword to a floating-point value specifying the size of the error bar endcaps. This value ranges from 0 to 1.0, where a value of 1.0 results in an endcap that is 10% of the data range.

FILL_BACKGROUND

This keyword is only available for 2D plots.

Set this keyword to fill the area under the plot. This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

FILL_COLOR

This keyword is only available for 2D plots.

Set this keyword to an RGB value specifying the color for the filled area. The default value is $[255, 255, 255]$ (white). This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

FILL_LEVEL

This keyword is only available for 2D plots.

Set this keyword to a floating-point value specifying the y-value for the lower boundary of the fill region. This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

HIDE

Set this keyword to a boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

HISTOGRAM

This keyword is only available for 2D plots.

Set this keyword to force only horizontal and vertical lines to be used to connect the plotted points. By default, the points are connected using a single straight line. This keyword is only available for two-dimensional plots. This keyword is ignored for three-dimensional plots.

IDENTIFIER

Set this keyword to a named IDL variable that will contain the `iToolID` for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LINestyle

Set this keyword to indicate the line style that should be used to draw the plot lines. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the `LINestyle` keyword equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq \textit{repeat} \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINestyle = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

LOCATION

Set this keyword to a two-element vector of the form $[x, y]$ to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the iTool and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

MAX_VALUE

The maximum value to be plotted. If this keyword is present, data values greater than the value of MAX_VALUE are treated as missing data and are not plotted.

Note

The IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

MIN_VALUE

The minimum value to be plotted. If this keyword is present, data values less than the value of MIN_VALUE are treated as missing data and are not plotted.

Note

The IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

NAME

Set this keyword to a string to specify the name for this visualization.

NO_SAVEPROMPT

Set this keyword to cause the iTool *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

NSUM

Set this keyword to the number of data points to average when plotting. If NSUM is larger than 1, every group of NSUM points is averaged to produce one plotted point. If there are M data points, then $M/NSUM$ points are plotted.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

POLAR

Set this keyword to display the plot as a polar plot. If this keyword is set, the arguments will be interpreted as R and θ or simply θ for a single argument. If R is not supplied the plot will use a vector of indices for the R argument.

RGB_TABLE

Set this keyword to either a 3 by 256 or 256 by 3 byte array containing color values to use for vertex colors. If the values supplied are not of type byte, they are scaled to the byte range using BYTSCL. Use the VERT_COLORS keyword to specify indices that select colors from the values specified with RGB_TABLE.

SCATTER

Set this keyword to generate a scatter plot. This action is equivalent to setting `LINESTYLE = 6` (no line) and `SYM_INDEX = 3` (Period symbol).

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied using the following rules:

- If the tool exists and /OVERPLOT is specified, then the style is only applied to the newly-created visualizations within the current view. The current tool style

is not updated with the new style, nor is the style applied to any other items within the view.

- Otherwise, if the tool exists and either `VIEW_NEXT` or `VIEW_NUMBER` is being used to select a different view, then the style is applied to all items within that view. The current tool style is updated with the new style.
- Otherwise, if a new tool is being created, then the style is applied to all items within all views. The current tool style is updated with the new style.

SYM_COLOR

Set this keyword to an RGB value specifying the color for the plot symbol.

Note

This color is applied to the symbol only if the `USE_DEFAULT_COLOR` property is set.

SYM_INCREMENT

Set this keyword to an integer value specifying the number of vertices to increment between symbol instances. The default value is 1, which places a symbol on every vertex.

SYM_INDEX

Set this keyword to one of the following scalar-represented internal default symbols:

- 0 = No symbol
- 1 = Plus sign, '+' (default)
- 2 = Asterisk
- 3 = Period (Dot)
- 4 = Diamond
- 5 = Triangle
- 6 = Square
- 7 = X
- 8 = Arrow Head

SYM_SIZE

Set this keyword to a floating-point value from 0.0 to 1.0 specifying the size of the plot symbol. A value of 1.0 results in an symbol that is 10% of the width/height of the plot.

SYM_THICK

Set this keyword to floating-point value from 1 to 10 points specifying the thickness of the plot symbol.

THICK

Set this keyword to a value between 1.0 and 10.0, specifying the line thickness to be used to draw the plotted lines, in points. The default is 1.0 points.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool and is used for tool-related display purposes only—as the root of the hierarchy shown in the Tool Browser, for example.

TRANSPARENCY

Set this keyword to floating-point value specifying the transparency of the filled area. Valid values range from 0.0 to 100.0. The default value is 0.0 (opaque).

USE_DEFAULT_COLOR

Set this keyword to have the color of the symbols match the plot color. If this keyword is set to 0 (`USE_DEFAULT_COLOR = 0`), the color specified by the `SYM_COLOR` keyword is used for symbols instead of matching the color of the plot.

VERT_COLORS

Set this keyword to a vector of indices into the color table to select colors to use for each vertex (plot data point). Alternately, set this keyword to a 3 by N byte array containing color values to use for each vertex. If the values supplied are not of type byte, they are scaled to the byte range using `BYTSCL`. If indices are supplied but no colors are provided with the `RGB_TABLE` keyword, then a default grayscale ramp is used. If a 3 by N array of colors is provided, the colors are used directly and the color values provided with `RGB_TABLE` are ignored. If the number of indices or colors specified is less than the number of vertices, the colors are repeated cyclically.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

XY_SHADOW

This keyword is only available for 3D plots.

Set this keyword to display a shadow of the plot in a three-dimensional plot. The shadow lies in the XY plane at the minimum value of the data space range of the z-axis. This keyword has no effect for two-dimensional plots.

[XYZ]_ERRORBARS

Set this keyword to show error bars. The Z_ERRORBARS keyword is for three-dimensional plots only.

[XYZ]GRIDSTYLE

The index of the linestyle to be used for plot tickmarks and grids (i.e., when [XYZ]TICKLEN is set to 1.0). See **LINestyle** for a list of linestyles.

[XYZ]_LOG

Set this keyword to specify a logarithmic axis. The minimum value of the axis range must be greater than zero. The **Z_LOG** keyword is for three-dimensional plots only.

[XYZ]ERROR

Set this keyword to either a vector or a 2 by N array of error values to be displayed as error bars for the [XYZ] dimension of the plot. The length of this array must be equal in length to the number of vertices of the plot or it will be ignored. If this keyword is set to a vector, the value will be applied as both a negative and positive error and the error bar will be symmetric about the plot vertex. If this keyword is set to a 2 by N array the [0, *] values define the negative error and the [1, *] values define the positive error, allowing asymmetric error bars. The **ZERROR** keyword is for three-dimensional plots only.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting **MAJOR** equal to zero suppresses major tickmarks entirely. **ZMAJOR** is for three-dimensional plots only.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting **MINOR** equal to zero suppresses minor tickmarks entirely. **ZMINOR** is for three-dimensional plots only.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum. **ZRANGE** is for three-dimensional plots only.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the

minor tick mark is one-half the length of the major tick mark. ZSUBTICKLEN is for three-dimensional plots only.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black). ZTEXT_COLOR is for three-dimensional plots only.

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

ZTICKFONT_INDEX is for three-dimensional plots only.

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points. ZTICKFONT_SIZE is for three-dimensional plots only.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

ZTICKFONT_STYLE is for three-dimensional plots only.

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an

array is provided, each string corresponds to a level of the axis. The `TICKUNITS` keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether `TICKUNITS` is specified.

If `TICKUNITS` are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If `TICKUNITS` are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Note

For more information, see “[\[XYZ\]TICKFORMAT](#)” on page 4345.

Used with the `LABEL_DATE` function, this property can easily create axes with date/time labels.

`ZTICKFORMAT` is for three-dimensional plots only.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis `[XYZ]RANGE` and the number of major tick marks (`[XYZ]MAJOR`). The value of this keyword takes precedence over the value set for the `[XYZ]MAJOR` keyword.

For example, if `TICKUNITS = ['S', 'H', 'D']`, and `TICKINTERVAL = 30`, then the interval between major ticks for the first axis level will be 30 seconds.

ZTICKINTERVAL is for three-dimensional plots only.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

ZTICKLAYOUT is for three-dimensional plots only.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKLEN is for three-dimensional plots only.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark. ZTICKNAME is for three-dimensional plots only.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- "" - Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

ZTICKUNITS is for three-dimensional plots only.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 C.E., respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point. ZTICKVALUES is for three-dimensional plots only.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis. ZTITLE is for three-dimensional plots only.

XZ_SHADOW

This keyword is only available for 3D plots.

Set this keyword to display a shadow of the plot in a three-dimensional plot. The shadow lies in the XZ plane at the minimum value of the data space range of the y-axis. This keyword has no effect for two-dimensional plots.

YZ_SHADOW

This keyword is only available for 3D plots.

Set this keyword to display a shadow of the plot in a three-dimensional plot. The shadow lies in the YZ plane at the minimum value of the data space range of the x-axis. This keyword has no effect for two-dimensional plots.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the File menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “[Data Import Methods](#)” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how to use the IDL Command Line to load data and variables into the iPlot tool.

At the IDL Command Line, enter:

```
file = FILEPATH('elnino.dat', SUBDIRECTORY = ['examples','data'])
data = READ_BINARY(file, DATA_TYPE = 4, DATA_DIMS = [500, 1], $
    ENDIAN = 'little')
time = DINDGEN(500)*0.25d + 1871
I PLOT, time, data, TITLE = 'El Nino', COLOR = [255, 128, 0]
```

Place a title on the time axis of your plot by selecting the axis, right-clicking to display the context menu, selecting **Properties** to display the property sheet, and typing `Year` in the **Title** field.

Place a title on the temperature axis of your plot by selecting the axis, displaying the property sheet, and entering the following in the **Title** field:

```
Temperature Anomaly (!Uo!NC)
```

Annotate your plot by selecting the Text Annotation tool, clicking near the top of the plot, and typing El Nino.

Add the special character to the annotation by selecting the annotation text, displaying the property sheet, selecting the lower-case n in Nino in the **Title** field, and replacing it with the following:

```
!Z(U+0F1)
```

Note

U+0F1 is unicode for the ñ character.

The following figure displays the output of this example:

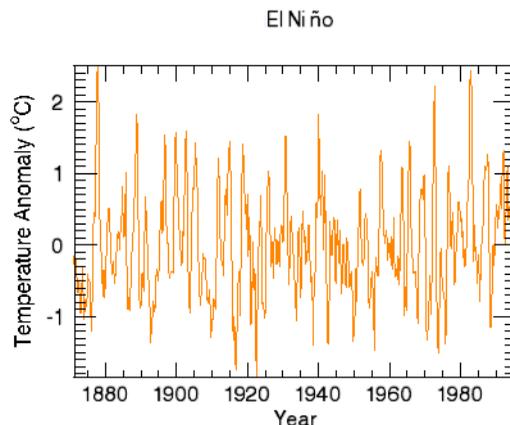


Figure 3-32: El Niño iPlot Example

Example 2

This example shows how to use the **File** → **Open** command to load binary data into the iPlot tool.

At the IDL Command Line, enter:

```
I PLOT
```

Select **File** → **Open** command to display the Open dialog, then browse to find `dirty_sine.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

In the Binary Template dialog, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** Byte (unsigned 8-bits)
- **Number of Dimensions:** 1
- **1st Dimension Size:** 256

Click **OK** to close the New Field dialog and the Binary Template dialog, and the surface is displayed.

Note

For more information on using the Binary Template to import data, see “Using the `BINARY_TEMPLATE` Function” in Chapter 15 of the *Using IDL* manual.

Annotate your plot by selecting the Text Annotation tool, clicking near the curve, and typing `Noisy Sine Wave`.

The following figure displays the output of this example:

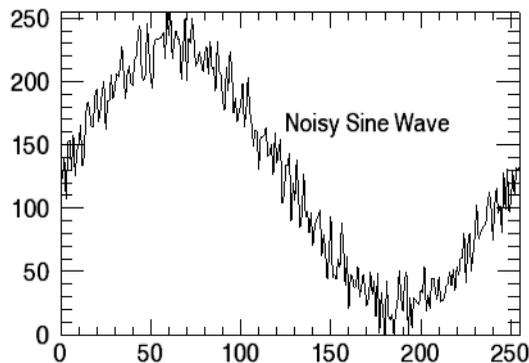


Figure 3-33: Noisy Sine Data *iPlot* Example

Example 3

This example shows how to use the **File** → **Import** command to load ASCII data into the iPlot tool.

At the IDL Command Line, enter:

```
I PLOT
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `sine_waves.txt` in the `examples/data` directory in the IDL distribution, and click **Next>>**.
3. At Step 3, select **Plot** and click **Finish**.

Then, the ASCII Template wizard is displayed.

1. At Step 1, click **Next>>** to accept the displayed data type/range definition.
2. At Step 2, click **Next>>** to accept the displayed delimiter/fields definition.
3. At Step 3, click **Finish** to accept the displayed field specification. The `sine_waves.txt` plot is displayed in the iPlot window.

The plot consists of two overlapping sine waves. To make it easier to distinguish between the two, change the appearance of the noisy sine wave to a dotted line pattern by selecting the noisy sine wave, right-clicking to display the context menu, selecting **Properties**, and changing the **Linestyle** property to a dotted line.

The following figure displays the output of this example:

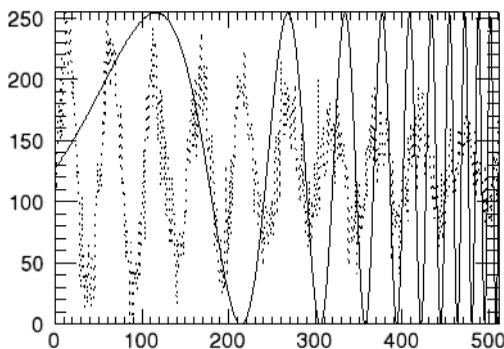


Figure 3-34: Overlapping Sine Waves iPlot Example

Version History

Introduced: 6.0

BACKGROUND_COLOR, MACRO_NAMES, STYLE_NAME keywords added:
6.1

ISHFT

The ISHFT function performs the bit shift operation on bytes, integers and longwords.

Syntax

$$Result = ISHFT(P_1, P_2)$$

Return Value

If P_2 is positive, P_1 is left shifted P_2 bit positions with 0 bits filling vacated positions. If P_2 is negative, P_1 is right shifted with 0 bits filling vacated positions.

Arguments

P_1

The scalar or array to be shifted.

P_2

The scalar or array containing the number of bit positions and direction of the shift.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Bit shift each element of the integer array [1, 2, 3, 4, 5] three bits to the left and store the result in B by entering:

```
B = ISHFT([1,2,3,4,5], 3)
```

The resulting array B is [8, 16, 24, 32, 40].

Version History

Introduced: Original

See Also

[SHIFT](#)

ISOCONTOUR

The ISOCONTOUR procedure interprets the contouring algorithm found in the IDLgrContour object. The algorithm allows for contouring on arbitrary meshes and returns line or orientated tessellated polygonal output. The interface will also allow secondary data values to be interpolated and returned at the output vertices as well.

Syntax

```
ISOCONTOUR, Values, Outverts, Outconn
  [, AUXDATA_IN=array, AUXDATA_OUT=variable]
  [, C_LABEL_INTERVAL=vector of values] [, C_LABEL_SHOW=vector of
  integers] [, C_VALUE=scalar or vector] [, /DOUBLE] [, /FILL]
  [, GEOMX=vector] [, GEOMY=vector] [, GEOMZ=vector]
  [, LEVEL_VALUES=variable] [, N_LEVELS=levels]
  [, OUT_LABEL_OFFSETS=variable] [, OUT_LABEL_POLYLINES=variable]
  [, OUT_LABEL_STRINGS=variable] [, OUTCONN_INDICES=variable]
  [, POLYGONS=array of polygon descriptions]
```

Arguments

Values

A vector or two-dimensional array specifying the values to be contoured. If the data is of double-precision floating-point type, or if the DOUBLE keyword is set, computations are done in double precision and the result is a double-precision value. If the data is of any other type, it is converted (if necessary) to single-precision floating-point; all computations are then done in single precision and the result is a single-precision value.

Outconn

Output variable to contain the connectivity information of the contour geometry in the form: [n0, i(0, 0), i(0, 1)..., i(0, n0-1), n1, i(1, 0), ...].

Outverts

Output variable to contain the contour vertices. The vertices are returned in double-precision floating point if the DOUBLE keyword is specified with a non-zero value. Otherwise, the vertices are returned in single-precision floating point.

Keywords

AUXDATA_IN

The auxiliary values to be interpolated at contour vertices. If p is the dimensionality of the auxiliary values, set this argument to a p -by- n array (if the *Values* argument is a vector of length n), or to a p -by- m -by- n array (if the *Values* argument is an m -by- n two-dimensional array).

AUXDATA_OUT

If the AUXDATA_IN keyword was specified, set this keyword to a named output variable to contain the interpolated auxiliary values at the contour vertices. If p is the dimensionality of the auxiliary values, the output is a p -by- n array of values, where n is the number of vertices in *Outverts*.

C_LABEL_INTERVAL

Set this keyword to a vector of values indicating the distance (measured parametrically relative to the length of each contour path) between labels for each contour level. If the number of contour levels exceeds the number of provided intervals, the C_LABEL_INTERVAL values will be repeated cyclically. The default is 0.4.

C_LABEL_SHOW

Set this keyword to a vector of integers. For each contour value, if the corresponding value in the C_LABEL_SHOW vector is non-zero, the contour line for that contour value will be labeled (with the corresponding label information returned via the OUT_LABEL_POLYS, OUT_LABEL_OFFSETS, and OUT_LABEL_STRINGS keywords). If the number of contour levels exceeds the number of elements in this vector, the C_LABEL_SHOW values will be repeated cyclically. The default is 0 indicating that no contour levels will be labeled.

C_VALUE

Set this keyword to a scalar value or a vector of values for which contour levels are to be generated. If this keyword is set to 0, contour levels will be evenly sampled across the range of the *Values* argument, using the value of the N_LEVELS keyword to determine the number of samples.

DOUBLE

Set this keyword to use double-precision to compute the contours. IDL converts any data supplied by the *Values* argument or GEOMX, GEOMY, and GEOMZ keywords to double precision and returns the *Outverts* argument in double precision. The default behavior is to convert the input to single precision and return the *Outverts* in single precision.

FILL

Set this keyword to generate an output connectivity as a set of polygons (Outconn is in the form used by the IDLgrPolygon POLYGONS keyword). The resulting representation is as a set of filled contours. The default is to generate line contours (Outconn is in the form used by the IDLgrPolyline POLYLINES keyword).

GEOMX

Set this keyword to a vector or two-dimensional array specifying the *X* coordinates of the geometry with which the contour values correspond. If *X* is a vector, it must match the number of elements in the *Values* argument, or it must match the first of the two dimensions of the *Values* argument (in which case the *X* coordinates will be repeated for each row of data values).

If necessary, data in the supplied vector or array will be converted to match the data type of the *Values* argument.

GEOMY

Set this keyword to a vector or two-dimensional array specifying the *Y* coordinates of the geometry with which the contour values correspond. If *Y* is a vector, it must match the number of elements in the *Values* argument, or it must match the second of the two dimensions of the *Values* argument (in which case the *Y* coordinates will be repeated for each column of data values).

If necessary, data in the supplied vector or array will be converted to match the data type of the *Values* argument.

GEOMZ

Set this keyword to a vector or two-dimensional array specifying the *Z* coordinates of the geometry with which the contour values correspond.

If GEOMZ is a vector or an array, it must match the number of elements in the *Values* argument.

If GEOMZ is not set, the geometry will be derived from the *Values* argument (if it is set to a two-dimensional array). In this case connectivity is implied. The X and Y coordinates match the row and column indices of the array, and the Z coordinates match the data values.

If necessary, data in the supplied vector or array will be converted to match the data type of the *Values* argument.

LEVEL_VALUES

Set this keyword to a named output variable to receive a vector of values corresponding to the values used to generate the contours. The length of this vector is equal to the number of contour levels generated. This vector is returned in double precision floating point.

N_LEVELS

Set this keyword to the number of contour levels to generate. This keyword is ignored if the C_VALUE keyword is set to a vector, in which case the number of levels is derived from the number of elements in that vector. Set this keyword to 0 to indicate that IDL should compute a default number of levels based on the range of data values. This is the default.

OUT_LABEL_OFFSETS

Set this keyword to a named variable that upon return will contain a vector of offsets (parameterized to the corresponding contour line) indicating the positions of the contour labels.

Note

The C_LABEL_SHOW keyword should be specified if this keyword is used.

OUT_LABEL_POLYLINES

Set this keyword to a named variable that upon return will contain a vector of polyline indices, [P0, P1, ...], that indicate which contour lines are labeled. Pi corresponds to the ith polyline specified via the Outconn argument. Note that if a given contour line has more than one label along its perimeter, then the corresponding polyline index may appear more than once in the LABEL_POLYS vector.

Note

The C_LABEL_SHOW keyword should be specified if this keyword is used.

OUT_LABEL_STRINGS

Set this keyword to a named variable that upon return will contain a vector of strings, [str0, str1, ...], that indicate the label strings.

Note

The C_LABEL_SHOW keyword should be specified if this keyword is used.

OUTCONN_INDICES

Set this keyword to a named output variable to receive an array of beginning and ending indices of connectivity for each contour level.

The output array is of the form: [start₀, end₀, start₁, end₁, ..., start_{nc-1}, end_{nc-1}], where *nc* is the number of contour levels. If a level has no contour lines, the start and stop pair is set to 0 and 0 for that level.

POLYGONS

Set this keyword to an array of polygonal descriptions that represents the connectivity information for the data to be contoured (as specified in the *Values* argument). A polygonal description is an integer or long array of the form: [n, i₀, i₁, ..., i_{n-1}], where *n* is the number of vertices that define the polygon, and i₀...i_{n-1} are indices into the GEOMX, GEOMY, and GEOMZ keywords that represent the polygonal vertices. To ignore an entry in the POLYGONS array, set the vertex count, *n* to 0. To end the drawing list, even if additional array space is available, set *n* to -1.

Version History

Introduced: 5.5

C_LABEL_INTERVAL, C_LABEL_SHOW, OUT_LABEL_OFFSETS, OUT_LABEL_POLYLINES, and OUT_LABEL_STRINGS keywords added: 5.6

ISOSURFACE

The ISOSURFACE procedure algorithm expands on the SHADE_VOLUME algorithm. It returns topologically consistent triangles by using oriented tetrahedral decomposition internally. This also allows the algorithm to isosurface any arbitrary tetrahedral mesh. If the user provides an optional auxiliary array, the data in this array is interpolated onto the output vertices and is returned as well. This auxiliary data array is allowed to have more than one value at each vertex. Any size leading dimension is allowed as long as the number of values in the subsequent dimensions matches the number of elements in the input Data array.

Syntax

```
ISOSURFACE, Data, Value, Outverts, Outconn
[, GEOM_XYZ=array, TETRAHEDRA=array]
[, AUXDATA_IN=array, AUXDATA_OUT=variable]
[, PROGRESS_CALLBACK=string] [, PROGRESS_METHOD=string]
[, PROGRESS_OBJECT=objref] [, PROGRESS_PERCENT=percent{0 to 100}]
[, PROGRESS_USERDATA=value]
```

Arguments

Data

Input three-dimensional array of scalars which are to be contoured.

Value

Input scalar contour value. This value specifies the constant-density surface (also called an iso-surface) to be extracted.

Outverts

A named variable to contain an output $[3, n]$ array of floating point vertices making up the triangle surfaces.

Outconn

A named variable to contain an output array of polygonal connectivity values (see IDLgrPolygon, POLYGONS keyword). If no polygons were extracted, this argument returns the array $[-1]$.

Keywords

AUXDATA_IN

Input array of auxiliary data with trailing dimensions being the number of values in Data.

Note

If AUXDATA_IN is specified then AUXDATA_OUT must also be specified.

AUXDATA_OUT

Set this keyword to a named variable that will contain an output array of auxiliary data sampled at the locations in Outverts.

Note

If AUXDATA_OUT is specified then AUXDATA_IN must also be specified.

GEOM_XYZ

A [3,n] input array of vertex coordinates (one for each value in the Data array). This array is used to define the spatial location of each scalar. If this keyword is omitted, Data must be a three-dimensional array and the scalar locations are assumed to be on a uniform grid.

Note

If GEOM_XYZ is specified then TETRAHEDRA must also be specified if either is to be specified.

PROGRESS_CALLBACK

Set this keyword to a scalar string containing the name of the IDL function that ISOSURFACE calls at PROGRESS_PERCENT intervals as it generates the isosurface.

The PROGRESS_CALLBACK function returns a zero to signal ISOSURFACE to stop generating the isosurface. This causes ISOSURFACE to return a single vertex and a connectivity array of [-1], which specifies an empty polygon. If the callback function returns any non-zero value, ISOSURFACE continues to generate the isosurface.

The `PROGRESS_CALLBACK` function must specify a single argument, *Percent*, which `ISOSURFACE` sets to an integer between 0 and 100, inclusive.

The `PROGRESS_CALLBACK` function may specify an optional `USERDATA` keyword parameter, which `ISOSURFACE` sets to the variable provided in the `PROGRESS_USERDATA` keyword.

The following code shows an example of a progress callback function:

```
FUNCTION myProgressCallback, percent,$      USERDATA = myStruct

oProgressBar = myStruct.oProgressBar

; This method updates the progress bar
; graphic and returns TRUE if the user has
; NOT pressed the cancel button.
keepGoing = oProgressBar-> $
    UpdateProgressValue(percent)

RETURN, keepGoing

END
```

PROGRESS_METHOD

Set this keyword to a scalar string containing the name of a function method that `ISOSURFACE` calls at `PROGRESS_PERCENT` intervals as it generates the isosurface. If this keyword is set, then the `PROGRESS_OBJECT` keyword must be set to an object reference that is an instance of a class that defines the specified method.

The `PROGRESS_METHOD` function method callback has the same specification as the callback described in the `PROGRESS_CALLBACK` keyword, except that it is defined as an object class method:

```
FUNCTION myClass::myProgressCallback, $
    percent, USERDATA = myStruct
```

PROGRESS_OBJECT

Set this keyword to an object reference that is an instance of a class that defines the method specified with the `PROGRESS_METHOD` keyword. If this keyword is set, then the `PROGRESS_METHOD` keyword must also be set.

PROGRESS_PERCENT

Set this keyword to a scalar in the range [1, 100] to specify the interval between invocations of the callback function. The default value is 5 and IDL silently clamps other values to the range [1, 100].

For example, a value of 5 (the default) specifies ISOSURFACE will call the callback function when the isosurface process is 0% complete, 5% complete, 10% complete, ..., 95% complete, and 100% complete.

PROGRESS_USERDATA

Set this property to any IDL variable that ISOSURFACE passes to the callback function in the callback function's USERDATA keyword parameter. If this keyword is specified, then the callback function must be able to accept keyword parameters.

TETRAHEDRA

An input array of tetrahedral connectivity values. If this array is not specified, the connectivity is assumed to be a rectilinear grid over the input three-dimensional array. If this keyword is specified, the input data array need not be a three-dimensional array. Each tetrahedron is represented by four values in the connectivity array. Every four values in the array correspond to the vertices of a single tetrahedron.

Version History

Introduced: 5.5

PROGRESS_CALLBACK, PROGRESS_METHOD, PROGRESS_OBJECT, PROGRESS_PERCENT, and PROGRESS_USERDATA keywords added: 6.0

See Also

[INTERVAL_VOLUME](#), [SHADE_VOLUME](#), [XVOLUME](#)

ISURFACE

The ISURFACE procedure creates an iTool and the associated user interface (UI) configured to display and manipulate surface data.

Note

If no arguments are specified, the ISURFACE procedure creates an empty Surface tool.

This routine is written in the IDL language. Its source code can be found in the file `isurface.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ISURFACE[, Z [, X, Y]]

iTool Common Keywords: [, [BACKGROUND_COLOR=](#)*value*] [, [DIMENSIONS](#)=[*x, y*] [, [IDENTIFIER](#)=*variable*] [, [LOCATION](#)=[*x, y*] [, [MACRO_NAMES](#)=*string or string array*] [, [NAME](#)=*string*] [, [/NO_SAVEPROMPT](#)] [, [OVERPLOT](#)=*iToolID*] [, [STYLE_NAME](#)=*string*] [, [TITLE](#)=*string*] [, [VIEW_GRID](#)=[*columns, rows*] [, [/VIEW_NEXT](#)] [, [VIEW_NUMBER](#)=*integer*]

Surface Keywords: [, [BOTTOM](#)=*index or RGB vector*] [, [CLIP_PLANES](#)=*array*] [, [COLOR](#)=*RGB vector*] [, [DEPTH_OFFSET](#)=*value*] [, [/EXTENDED_LEGO](#)] [, [/HIDDEN_LINES](#)] [, [/HIDE](#)] [, [LINESTYLE](#)=*value*] [, [RGB_TABLE](#)=*array*] [, [SHADING](#)={0 | 1}] [, [/SHOW_SKIRT](#)] [, [SKIRT](#)=*Z value*] [, [STYLE](#)={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, [TEXTURE_ALPHA](#)=*array*] [, [TEXTURE_BLUE](#)=*array*] [, [TEXTURE_GREEN](#)=*array*] [, [TEXTURE_HIGHRES](#)={0 | 1 | 2}] [, [TEXTURE_IMAGE](#)=*array*] [, [/TEXTURE_INTERP](#)] [, [TEXTURE_RED](#)=*array*] [, [THICK](#)=*float {1.0 to 10.0}*] [, [/USE_TRIANGLES](#)] [, [VERT_COLORS](#)=*vector or array*] [, [ZERO_OPACITY_SKIP](#)={0 | 1}]

Axis Keywords: [, [\[XYZ\]GRIDSTYLE](#)={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, [\[XYZ\]MAJOR](#)=*integer*] [, [\[XYZ\]MINOR](#)=*integer*] [, [\[XYZ\]RANGE](#)=[*min, max*] [, [\[XYZ\]SUBTICKLEN](#)=*ratio*] [, [\[XYZ\]TEXT_COLOR](#)=*RGB vector*] [, [\[XYZ\]TICKFONT_INDEX](#)={0 | 1 | 2 | 3 | 4}] [, [\[XYZ\]TICKFONT_SIZE](#)=*integer*] [, [\[XYZ\]TICKFONT_STYLE](#)={0 | 1 | 2 | 3}] [, [\[XYZ\]TICKFORMAT](#)=*string or string array*]

```
[, [XYZ]TICKINTERVAL=value] [, [XYZ]TICKLAYOUT={0 | 1 | 2}]
[, [XYZ]TICKLEN=value] [, [XYZ]TICKNAME=string array]
[, [XYZ]TICKUNITS=string] [, [XYZ]TICKVALUES=vector]
[, [XYZ]TITLE=string]
```

Arguments

X

A vector or two-dimensional array specifying the x -coordinates of the grid.

If X is a vector:

- If Y and Z are vectors and have the same length as X :
Each element of X specifies the x -coordinates of a point in space (e.g., $X[0]$ specifies the x -coordinate for $Y[0]$ and $Z[0]$). The gridding wizard will automatically launch in this case.
- If Z is a two-dimensional array:
Each element of X specifies the x -coordinates for a column of Z (e.g., $X[0]$ specifies the x -coordinate for $Z[0, *]$).

If X is a two-dimensional array, each element of X specifies the x -coordinate of the corresponding point in Z (X_{ij} specifies the x -coordinate of Z_{ij}).

Y

A vector or two-dimensional array specifying the y -coordinates of the grid.

If Y is a vector:

- If X and Z are vectors and have the same length as Y :
Each element of Y specifies the y -coordinates of a point in space (e.g., $Y[0]$ specifies the y -coordinate for $X[0]$ and $Z[0]$). The gridding wizard will automatically launch in this case.
- If Z is a two-dimensional array:
Each element of Y specifies the y -coordinates for a column of Z (e.g., $Y[0]$ specifies the y -coordinate for $Z[* , 0]$).

If Y is a two-dimensional array, each element of Y specifies the y -coordinate of the corresponding point in Z (Y_{ij} specifies the y -coordinate of Z_{ij}).

Z

A vector or two-dimensional array specifying the data to be displayed.

If Z is a vector,

- If X and Y are vectors and have the same length as Z:

Each element of Z specifies the z-coordinates of a point in space (e.g., Z[0] specifies the z-coordinate for X[0] and Y[0]). The gridding wizard will automatically launch in this case.

If Z is a two-dimensional array,

- If X and Y are provided:

The surface is defined as a function of the (x, y) locations specified by their contents.

- If X and Y are not provided:

The surface is generated as a function of the array indices of each element of Z.

Keywords

Note

Keywords to the ISURFACE routine that correspond to the names of *registered properties* of the iSurface tool must be specified in full, without abbreviation.

BACKGROUND_COLOR

Set this keyword to an RGB value specifying the color to be used as the background color for the view. The default is [255, 255, 255] (white). The BACKGROUND_COLOR keyword can be used when a tool is being created or when a new visualization is being created in an existing tool with the use of the OVERPLOT, VIEW_NUMBER or VIEW_NEXT keywords. The background color is applied to the current view. For example, if multiple views have been created with the VIEW_GRID keyword, and the VIEW_NUMBER keyword is used to create a visualization in the second view, use of the BACKGROUND_COLOR keyword would set the background color in the second view only.

BOTTOM

Set this keyword to an RGB color for drawing the bottom of the surface. Set this keyword to a scalar to draw the bottom with the same color as the top.

CLIP_PLANES

Set this keyword to an array of dimensions $[4, N]$ specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COLOR

Set this keyword to the color to be used as the foreground color for this model. The color is specified as an RGB vector. The default is $[225, 184, 0]$.

DIMENSIONS

Set this keyword to a two-element vector of the form $[\text{width}, \text{height}]$ to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

DEPTH_OFFSET

Set this keyword to an integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.

The units are “Z-Buffer units,” where a value of 1 is used to specify a distance that corresponds to a single step in the device’s Z-Buffer.

Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms. This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for “layering” complex scenes with multiple DEPTH_OFFSET values. It is safest to

use only a `DEPTH_OFFSET` value of 0, the default, and one other non-zero value, such as 1. Many system-level graphics drivers are not consistent in their handling of `DEPTH_OFFSET` values, particularly when multiple non-zero values are used. This can lead to portability problems because a set of `DEPTH_OFFSET` values may produce better results on one machine than on another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use `DEPTH_OFFSET`.

Note

`DEPTH_OFFSET` has no effect unless the value of the `STYLE` keyword is 2 or 6 (Filled or LegoFilled).

EXTENDED_LEGO

Set this keyword to force the iSurface tool to display the last row and column of data when lego display styles are selected.

HIDDEN_LINES

Set this keyword to draw point and wireframe surfaces using hidden line (point) removal. By default, hidden line removal is disabled.

HIDE

Set this keyword to a boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

IDENTIFIER

Set this keyword to a named IDL variable that will contain the iToolID for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

LINestyle

Set this keyword to indicate the line style that should be used to draw the surface lines. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the `LINestyle` keyword equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq \textit{repeat} \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINestyle = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

LOCATION

Set this keyword to a two-element vector of the form [*x*, *y*] to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the *iTool* and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

NAME

Set this keyword to a string to specify the name for this particular tool. The name is used for tool-related display purposes only—as the root of the hierarchy shown in the Tool Browser, for example.

NO_SAVEPROMPT

Set this keyword to cause the iTool *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword equal to one to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

RGB_TABLE

Set this keyword to a two-dimensional array containing RGB triplets defining the colors to be used in a color indexed texture image or by vertex colors. The values should be within the range of $0 \leq \text{value} \leq 255$. The array must be a 3 by N array where m must not exceed 256.

SHADING

Set this keyword to an integer representing the type of shading to use if STYLE is set to 2 (Filled).

- 0 = Flat (default): The color has a constant intensity for each face of the surface, based on the normal vector.
- 1 = Gouraud: The colors are interpolated between vertices, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

SHOW_SKIRT

Set this keyword to enable skirt drawing. The default is to disable skirt drawing.

SKIRT

Set this keyword to the Z value at which a skirt is to be defined around the array. The Z value is expressed in data units; the default is 0.0. If a skirt is defined, each point on the four edges of the surface is connected to a point on the skirt which has the given Z value, and the same X and Y values as the edge point. In addition, each point on the skirt is connected to its neighbor. The skirt value is ignored if skirt drawing is disabled (see `SHOW_SKIRT` above). IDL converts, maintains, and returns this data as double-precision floating-point.

STYLE

Set this keyword to an integer value that indicates the style to be used to draw the surface. Valid values are:

- 0 = Points
- 1 = Wire mesh
- 2 = Filled (the default)
- 3 = RuledXZ
- 4 = RuledYZ
- 5 = Lego
- 6 = LegoFilled: for outline or shaded and stacked histogram-style plots.

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied using the following rules:

- If the tool exists and `/OVERPLOT` is specified, then the style is only applied to the newly-created visualizations within the current view. The current tool style is not updated with the new style, nor is the style applied to any other items within the view.
- Otherwise, if the tool exists and either `VIEW_NEXT` or `VIEW_NUMBER` is being used to select a different view, then the style is applied to all items within that view. The current tool style is updated with the new style.
- Otherwise, if a new tool is being created, then the style is applied to all items within all views. The current tool style is updated with the new style.

TEXTURE_ALPHA

Set the keyword to a two-dimensional array containing the alpha channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_RED, TEXTURE_GREEN, and TEXTURE_BLUE be set to arrays of identical size and type.

TEXTURE_BLUE

Set the keyword to a two-dimensional array containing the blue channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_RED and TEXTURE_GREEN be set to arrays of identical size and type.

TEXTURE_GREEN

Set the keyword to a two-dimensional array containing the green channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_RED and TEXTURE_BLUE be set to arrays of identical size and type.

TEXTURE_HIGHRES

An integer value that determines the type of texture tiling, if any, that will be used to maintain the full pixel resolution of the original texture image. Set this property to one of the following options:

- 0 = If the texture map is larger than the maximum resolution supported by the 3-D hardware, the texture is scaled down to this resolution.
- 1 = If the texture map is larger than the maximum resolution supported by the hardware, texture tiling is used to ensure no loss of data. As a large image could easily consume all available video memory, this mode employs texture Level of Detail (LoD). Rather than loading the entire image into the texture tiles at the full resolution, the image tiles are downsampled based on the level of zoom. If the entire image is visible the texture tiles will contain highly downsampled data from the image to conserve video memory. As a portion of the image is zoomed in on, progressively higher resolution image data is loaded into the visible tiles, until the maximum resolution is reached. The tiles that are not visible during zooming are not loaded with high resolution data.

Note

This mode is recommended if IDL is running on modern 3-D hardware and resolution loss due to downscaling becomes problematic when TEXTURE_HIGHRES=0.

- 2 = If the texture map is larger than the maximum resolution supported by the hardware, texture tiling is used to ensure no loss of data. However, the highest resolution data is loaded into all tiles. LoD tiling is disabled. This mode should only be used if the video memory is sufficient for the image size.

If not set, and the texture map is larger than the maximum resolution supported by the 3-D hardware, the texture is scaled down to the maximum resolution supported by the 3-D hardware on your system. The default value is 0.

Note

TEXTURE_HIGHRES is supported only when STYLE=2 (Filled).

TEXTURE_IMAGE

Set this keyword to an array containing an image to be texture mapped onto the surface. If this keyword is omitted or set to a null object reference, no texture map is applied and the surface is filled with the color specified by the COLOR or VERTEX_COLORS property. The image array can be a two-dimensional array of color indexes or a three-dimensional array specifying RGB values at each pixel ($3 \times n \times m$, $n \times 3 \times m$, or $n \times m \times 3$). Setting TEXTURE_IMAGE to a three-dimensional array contains an alpha channel ($4 \times n \times m$, $n \times 4 \times m$, or $n \times m \times 4$) allows you to create a transparent iSurface object. The TEXTURE_IMAGE keyword will override any values passed to TEXTURE_RED, TEXTURE_GREEN, TEXTURE_BLUE, or TEXTURE_ALPHA.

TEXTURE_INTERP

Set this keyword to a nonzero value to indicate that bilinear sampling is to be used with texture mapping. The default method is nearest-neighbor sampling.

TEXTURE_RED

Set the keyword to a two-dimensional array containing the red channel of an image to be used as a texture image. Use of this keyword requires that TEXTURE_GREEN and TEXTURE_BLUE be set to arrays of identical size and type.

THICK

Set this keyword to a value between 1.0 and 10.0, specifying the line thickness to use to draw surface lines, in points. The default is 1.0 points.

TITLE

Set this keyword to a string to specify a title for the tool. The title is displayed in the title bar of the tool.

USE_TRIANGLES

Set this keyword to force the iSurface tool to use triangles instead of quads to draw the surface and skirt.

VERT_COLORS

Set this keyword to a vector, two-dimensional array of equal size to *Z*, or a two-dimensional array containing RGB triplets representing colors to be used at each vertex. If this keyword is set to a vector or a two-dimensional array of equal size to *Z*, these values are indices into a color table that can be specified by the `RGB_TABLE` keyword. If the `RGB_TABLE` keyword is not set, a grayscale color is used. If more vertices exist than elements in `VERT_COLORS`, the elements of `VERT_COLORS` are cyclically repeated. If this keyword is omitted, the surface is drawn in the color specified by the `COLOR` keyword or the default color.

VIEW_GRID

Set this keyword to a two-element vector of the form [*columns*, *rows*] to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if `OVERPLOT`, `VIEW_NEXT`, or `VIEW_NUMBER` are specified then `VIEW_GRID` is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then `/VIEW_NEXT` will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless `/OVERPLOT` is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the `VIEW_NUMBER` before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not

necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

[XYZ]GRIDSTYLE

The index of the linestyle to be used for plot tickmarks and grids (i.e., when [XYZ]TICKLEN is set to 1.0). See LINSTYLE for a list of linestyles.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica

- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether TICKUNITS is specified.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)

- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Note

For more information, see “[XYZ]TICKFORMAT” on page 4345.

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if TICKUNITS = ['S', 'H', 'D'], and TICKINTERVAL = 30, then the interval between major ticks for the first axis level will be 30 seconds.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.

- "" - Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 C.E., respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis.

ZERO_OPACITY_SKIP

Set this keyword to gain finer control over the rendering of textured surface pixels (texels) with an opacity of 0 in the texture map. Texels with zero opacity do not affect the color of a screen pixel since they have no opacity. If this keyword is set to 1, any texels are "skipped" and not rendered at all. If this keyword is set to zero, the Z-buffer is updated for these pixels and the display image is not affected as noted above. By updating the Z-buffer without updating the display image, the surface can be used as a *clipping* surface for other graphics primitives drawn after the current graphics object. The default value for this keyword is 1.

Note

This keyword has no effect if no texture map is used or if the texture map in use does not contain an opacity channel.

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the File menu in the

iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “[Data Import Methods](#)” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how to use the IDL Command Line to load data into the iSurface tool.

At the IDL Command Line, enter:

```
file = FILEPATH('surface.dat', $
  SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [350, 450], DATA_TYPE = 2, $
  ENDIAN = 'little')
ISURFACE, data, TITLE = 'Maroon Bells Elevation', $
  COLOR = [255, 128, 0]
```

Place a title on the elevation axis of your plot by selecting the axis, right-clicking to display the context menu, selecting **Properties** to display the property sheet, and typing `Elevation (m)` in the **Title** field.

Use the **Operations** → **Statistics...** option to display the iTools Statistics dialog. Within this dialog, observe the Z value’s Maximum, which is 4241 at [29, 253]. Close the iTools Statistics dialog by selecting **File** → **Close**.

Annotate your plot by selecting the Text Annotation tool, clicking near the top of the highest peak in the display, and typing `Highest Point (4241 m)`. Draw a line annotation between the text annotation and the highest peak on the surface.

The following figure displays the output of this example:

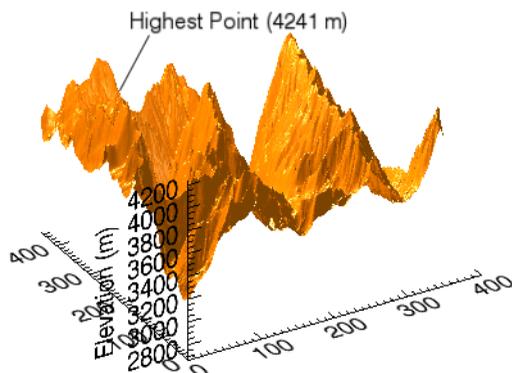


Figure 3-35: Maroon Bells iSurface Example

Example 2

This example shows how to use the **File** → **Open** command to load binary data into the iSurface tool.

At the IDL Command Line, enter:

```
ISURFACE
```

Select **File** → **Open** to display the Open dialog, then browse to find `idemosurf.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

The Binary Template wizard is displayed. In the Binary Template, change **File's byte ordering** to `Little Endian`. Then, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Float (32 bit)`
- **Number of Dimensions:** `2`
- **1st Dimension Size:** `200`
- **2nd Dimension Size:** `200`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the surface is displayed.

Note

For more information on using the Binary Template to import data, see “Using the BINARY_TEMPLATE Function” in Chapter 15 of the *Using IDL* manual.

Insert a contour onto the surface by clicking the **Surface Contour** button on the toolbar, then clicking and dragging on the surface to position the contour at the desired height.

The following figure displays the output of this example:

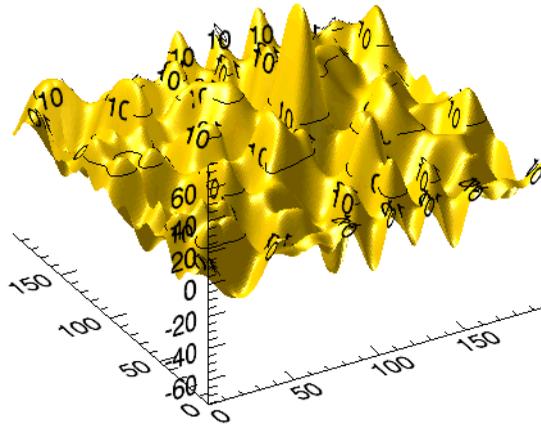


Figure 3-36: Binary Surface Data iSurface Example

Example 3

This example shows how to use the **File** → **Import** command to load ASCII data into the iSurface tool.

At the IDL Command Line, enter:

```
ISURFACE
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `irreg_grid1.txt` in the `examples/data` directory in the IDL distribution, and click **Next>>**.
3. At Step 3, select **Surface** and click **Finish**.

Then, the ASCII Template wizard is displayed.

1. At Step 1, click **Next>>** to accept the displayed Data Type/Range definitions.
2. At Step 2, click **Next>>** to accept the displayed Delimiter/Fields definitions.
3. At Step 3, click **Finish** to accept the displayed Field Specifications.

Note

For more information on using the ASCII Template to import data, see “Using the ASCII_TEMPLATE Function” in Chapter 14 of the *Using IDL* manual.

At the iTool’s Create Visualization window, you have the option of launching the Gridding wizard or not creating a visualization. Choose **Launch the gridding wizard** and click **Ok**.

4. At Step 1, click **Next>>** to accept the interpolation of data values and locations.
5. At Step 2, click **Next>>** to accept the dimensions, start and spacing.
6. At Step 3, select **Inverse Distance** as the gridding method, click **Preview** to preview the possible results, and click **Finish** to display the surface.

Double-click the surface to display the Properties sheet, and change the **Fill shading** setting from `Flat` to `Gouraud`.

Use the Rotate tool on the Toolbar to rotate the surface slightly forward to better display the surface convolutions.

The following figure displays the output of this example.

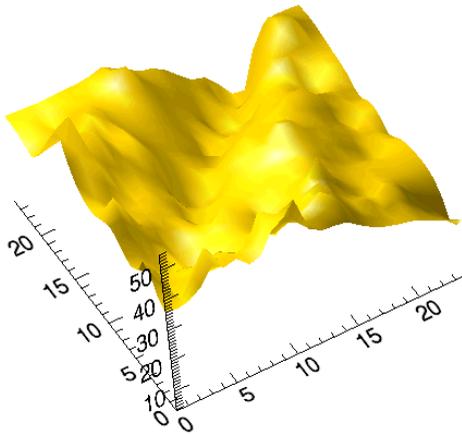


Figure 3-37: ASCII Surface Data iSurface Example

Version History

Introduced: 6.0

BACKGROUND_COLOR, MACRO_NAMES, STYLE_NAME keywords added:
6.1

ITCURRENT

The ITCURRENT procedure is used to set the current tool in the IDL Intelligent Tools system. This routine is used with the identifier of the tool to make it current in the system. If the identifier is valid, the specified tool becomes current.

When a tool is set as current, the visible display or the focus state of the tool does not change. Only the internal setting of the current tool changes.

Besides using this procedure to set the current tool, a tool is made current when it is created or when it is placed in focus in the current windowing system.

This routine is written in the IDL language. Its source code can be found in the file `itcurrent.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

ITCURRENT, *iToolID*

Arguments

iToolID

The identifier of the existing iTool to be set as current.

Keywords

None.

Example

Enter the following at the IDL Command Line:

```
IPlot, IDENTIFIER = PlotID1
current1 = ITGETCURRENT()
PRINT, 'The current tool is ', current1
```

An iPlot tool is created, and the newly created iPlot tool becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/IPlot_8
```

Enter the following at the IDL Command Line:

```
I PLOT, IDENTIFIER = PlotID2
current2 = ITGETCURRENT()
PRINT, 'The current tool is ', current2
```

A second iPlot tool is created, and this newly created iPlot tool becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/I PLOT_9
```

Enter the following at the IDL Command Line:

```
I SURFACE, IDENTIFIER = SurfaceID1
current3 = ITGETCURRENT()
PRINT, 'The current tool is ', current3
```

An iSurface tool is created, and the newly created iSurface tool becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/I SURFACE_5
```

Enter the following at the IDL Command Line:

```
I TCURRENT, PlotID1
current = ITGETCURRENT()
PRINT, 'The current tool is ', current
END
```

The iPlot tool created at the beginning of the example (PlotID1) becomes the current tool. Output similar to the following appears in the IDL Output Log:

```
The current tool is /TOOLS/I PLOT_8
```

Note that the system ID of the current tool (I PLOT_8) is the same as that of the current tool at the beginning of the exercise.

Version History

Introduced: 6.0

See Also

[ITDELETE](#), [ITGETCURRENT](#), [ITRESET](#)

ITDELETE

The ITDELETE procedure is used to delete a tool in the IDL Intelligent Tools system. If a valid identifier is provided, the tool represented by the identifier is destroyed. If no identifier is provided, the current tool is destroyed.

When a tool is destroyed, all resources specific to that tool are released and the tool ceases to exist.

This routine is written in the IDL language. Its source code can be found in the file `itdelete.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

```
ITDELETE[, iToolID]
```

Arguments

iToolID

This optional argument contains the identifier for the specific iTool to delete. If not provided, the current tool is destroyed.

Keywords

None.

Example

Enter the following at the IDL Command Line:

```
I PLOT, IDENTIFIER = PlotID1  
I SURFACE, IDENTIFIER = SurfaceID1
```

Two tools are created: an iPlot tool and an iSurface tool.

Next, enter the following at the IDL Command Line:

```
ITDELETE, plotID1
```

The iPlot tool is deleted, leaving only the iSurface tool.

Version History

Introduced: 6.0

See Also

[ITCURRENT](#), [ITGETCURRENT](#), [ITRESET](#)

ITGETCURRENT

The ITGETCURRENT function is used to get the identifier of, and optionally and object reference to, the current tool in the IDL Intelligent Tools system.

Note

You can also retrieve the identifier of an iTool by specifying the IDENTIFIER keyword in the call to the routine that creates the tool (*e.g.* IPLOT or IIMAGE).

This routine is written in the IDL language. Its source code can be found in the file `itgetcurrent.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

Result = ITGETCURRENT([**TOOL**=*variable*])

Return Value

Returns the identifier of the current tool in the iTool system. If no tool exists, an empty string (' ') is returned.

Arguments

None.

Keywords

TOOL

Set this keyword to a named variable in which to return the object reference to the current tool object, or a null object if no tool exists.

Example

Suppose you have several iPlot tools running in your IDL session, and want to retrieve the iTool identifier for one of them. Select the iPlot tool using the mouse, and issue the following IDL command:

```
idPlot = ITGETCURRENT()
```

The `idPlot` variable would contain the iTool identifier for the selected tool.

Controlling the contents of an existing iTool from the IDL command line sometimes requires the use of the iTool's *object reference* rather than its identifier. For example, suppose you have created an iPlot tool using the following command:

```
IPLOT, SIN(FINDGEN(361) * !DTOR), COLOR=[0, 0, 255], THICK=2
```

To rotate the plot from the IDL command line, you could use the following statements:

```
idPlot = ITGETCURRENT(TOOL=oPlot)
void = oPlot->DoAction('OPERATIONS/OPERATIONS/ROTATE/ROTATERIGHT')
```

The process of controlling an iTool from the IDL command line is discussed in detail in [Chapter , “Controlling iTools from the IDL Command Line”](#) in the *iTool Developer’s Guide* manual.

Version History

Introduced: 6.0

TOOL keyword added: 6.1

See Also

[ITCURRENT](#), [ITDELETE](#), [ITRESET](#)

[Chapter , “Controlling iTools from the IDL Command Line”](#) in the *iTool Developer’s Guide* manual

ITREGISTER

The ITREGISTER procedure is used to register iTool object classes or other iTool functionality with the IDL Intelligent Tools system. Functionality that is registered with the iTools system is available to all iTools. See the Register methods of the [IDLitTool](#) class for information on how to register functionality with an individual iTool.

This routine is written in the IDL language. Its source code can be found in the file `itregister.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

```
ITREGISTER, Name, ItemName [, /ANNOTATION] [, /DEFAULT]  
          [, /FILE_READER] [, /FILE_WRITER] [, TYPES=string] [, /UI_PANEL]  
          [, /UI_SERVICE] [, /USER_INTERFACE] [, /VISUALIZATION]
```

Arguments

Name

A string containing the name used to refer to the associated class once registration is completed. Subsequent calls to create items of this type will use this name to identify the associated class.

Note

If no keywords are supplied, *Name* is the name of an iTool.

ItemName

A string containing the class name of the object class or user interface routine that is to be associated with *Name*. When an item of name *Name* is requested from the system, an object of this class is created or the specified routine is called.

Note

If no keywords are supplied, *ItemName* is the name of an iTool class.

Keywords

Note

Keywords supplied in the call to ITREGISTER but not listed here are passed directly to the underlying objects' registration routines.

ANNOTATION

Set this keyword to indicate that an annotation is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the annotation type, and *ItemName* is the class name of the annotation type's object class.

DEFAULT

Set this keyword to specify that the item being registered should be the *default item* of its type. Making an item the default places the item first in the list of items of its type. When the iTool system chooses an item, it will use the first item in the list that matches the data type of the selected data. When items are displayed in a list for the user to select from, the default item will appear at the top of the list.

For example, if you register a visualization type with the DEFAULT keyword set, your visualization type will be used by the iTool (assuming the selected data is of the proper type) unless you specifically choose a different visualization type.

Note

If you register more than one item with the DEFAULT keyword set, the *last* item registered will appear *first* in the list of items of that type.

FILE_READER

Set this keyword to indicate that a file reader is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the file reader, and *ItemName* is the class name of the file reader's object class.

FILE_WRITER

Set this keyword to indicate that a file writer is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the file writer, and *ItemName* is the class name of the file writer's object class.

TYPES

This keyword is only used in conjunction with the UI_PANEL keyword.

Set this keyword equal to a string or string array containing iTool types with which the UI panel should be associated. When the registered type of a UI panel matches the registered type of an iTool, the panel will be displayed as part of the iTool's interface.

UI_PANEL

Set this keyword to indicate that a UI panel is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the panel and *ItemName* is the routine that should be called when the panel is created.

To specify that the UI panel is associated with a particular iTool or iTools, set the TYPES keyword to the iTool types that should expose this panel.

UI_SERVICE

Set this keyword to indicate that a UI service is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the UI service and *ItemName* is the routine that should be called to execute the service.

USER_INTERFACE

Set this keyword to indicate that a user interface is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the user interface, and *ItemName* is the name of the user interface procedure.

VISUALIZATION

Set this keyword to indicate that a visualization is being registered with the system. When this keyword is set, the value of *Name* is the string used to refer to the visualization type, and *ItemName* is the class name of the visualization type's object class.

Examples

Suppose you have an iTool class definition file named `myTool__define.pro`, located in a directory included in IDL's `!PATH` system variable. Register this class with the iTool system with the following command:

```
ITREGISTER, 'My First Tool', 'myTool'
```

Tools defined by the `myTool` class definition file can now be created by the iTool system by specifying the tool name `My First Tool`.

Similarly, suppose you have a user interface service defined in a file named `myUIFileOpen.pro`. Register this UI service with the iTool system with the following command:

```
ITREGISTER, 'My File Open', 'myUIFileOpen', /UI_SERVICE
```

Finally, suppose you have a user interface panel defined in a file named `myPanel.pro`, and that you want this panel to be added to the user interface of iTools registered with the `TYPES` property set to `MYTOOL`. Register this UI panel with the iTool system with the following command:

```
ITREGISTER, 'My Panel', 'myPanel', /UI_PANEL, TYPES = 'MYTOOL'
```

Version History

Introduced: 6.0

`ANNOTATION`, `DEFAULT`, `FILE_READER`, `FILE_WRITER`, `USER_INTERFACE`
keywords added: 6.1

See Also

[Chapter 5, “Creating an iTool”](#) in the *iTool Developer’s Guide* manual

ITRESET

The ITRESET procedure resets the IDL iTools session. When called, all active tools and overall system management is destroyed and associated resources released.

This routine is written in the IDL language. Its source code can be found in the file `itreset.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

```
ITRESET[, /NO_PROMPT]
```

Arguments

None

Keywords

NO_PROMPT

Set this keyword to disable prompting the user before resetting the system. If this keyword is set, the user is not presented with a prompt and the reset is performed immediately.

Examples

The iTool Data Manager system maintains your data during the entire IDL session, unless ITRESET is used. This example shows how the data is maintained and how ITRESET is used to clear the iTool Data Manager.

Read in plot data and load it into an iPlot tool at the IDL Command Line:

```
file = FILEPATH('dirty_sine.dat', $
  SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [256, 1])
IPLOT, data
```

Delete this tool with the ITDELETE procedure at the IDL Command Line:

```
ITDELETE
```

Read in surface data and load it into an iSurface tool at the IDL Command Line:

```
file = FILEPATH('elevbin.dat', $
  SUBDIRECTORY = ['examples', 'data'])
data = READ_BINARY(file, DATA_DIMS = [64, 64])
```

```
ISURFACE, data
```

Use **Window** → **Data Manager...** to access the Data Manager Browser. The browser contains both plot and surface parameters. Although the iPlot tool was deleted, its data remains in the Data Manager. Click **Dismiss**.

Use **File** → **New** → **iPlot** to create an empty iPlot tool. If you want to load the plot data in the Data Manager into this tool, use **Insert** → **Visualization** to access the Insert Visualization dialog, which allows you to specify the plot data to be displayed.

At the IDL Command Line, enter:

```
ITRESET, /NO_PROMPT
```

The two iTools are deleted and the data in the Data Manager is released. To verify the data in released, create an empty iSurface tool at the IDL Command Line:

```
ISURFACE
```

Use **Window** → **Data Manager...** to access the Data Manager Browser. No data appears in the browser. The iTool Data Manger in empty. Click **Dismiss**.

At the IDL Command Line, enter:

```
ITRESET, /NO_PROMPT
```

Version History

Introduced: 6.0

See Also

[ITCURRENT](#), [ITDELETE](#), [ITGETCURRENT](#)

ITRESOLVE

The ITRESOLVE procedure resolves all IDL code within the iTools directory, as well as all other IDL code required for the iTools framework. This procedure is useful for constructing SAVE files containing user code that requires the iTools framework.

This routine is written in the IDL language. Its source code can be found in the file `itresolve.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

```
ITRESOLVE [, PATH=string]
```

Arguments

None.

Keywords

PATH

Set this keyword to a string containing the full path to the iTools directory. The default is to use the `lib/itools` subdirectory within which the ITRESOLVE procedure resides.

Examples

Suppose you wish to create a SAVE file that contains all of the code necessary to run an iTool you have created with the name `mytool`. First compile all of your own code:

```
.COMPILE mytool
```

Now compile all of the iTools code:

```
ITRESOLVE
```

Finally, create the SAVE file:

```
SAVE, FILE='mytool.sav', /ROUTINES, /COMPRESS
```

Version History

Introduced: 6.1

See Also

[RESOLVE_ALL](#), [SAVE](#)

IVOLUME

The IVOLUME procedure creates an iTool and associated user interface (UI) configured to display and manipulate volume data.

Note

If no arguments are specified, the IVOLUME procedure creates an empty Volume tool.

This routine is written in the IDL language. Its source code can be found in the file `ivolume.pro` in the `lib/itools` subdirectory of the IDL distribution.

Syntax

IVOLUME[, Vol₀[, Vol₁][, Vol₂, Vol₃]]

iTool Common Keywords: [, BACKGROUND_COLOR=*value*]

[, DIMENSIONS=[*x*, *y*]] [, IDENTIFIER=*variable*] [, LOCATION=[*x*, *y*]]
 [, MACRO_NAMES=*string or string array*] [, NAME=*string*]
 [, /NO_SAVEPROMPT] [, OVERPLOT=*iToolID*] [, STYLE_NAME=*string*]
 [, TITLE=*string*] [, VIEW_GRID=[*columns*, *rows*]] [, /VIEW_NEXT]
 [, VIEW_NUMBER=*integer*]

Volume Keywords: [, AMBIENT=*RGB vector*] [, /AUTO_RENDER]

[, BOUNDS=[*xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax*]] [, CLIP_PLANES=*array*]
 [, COMPOSITE_FUNCTION={0 | 1 | 2 | 3}]
 [, EXTENTS_TRANSPARENCY=*integer*] [, DEPTH_CUE=[*zbright*, *zdim*]]
 [, /HIDE] [, HINTS={0 | 1 | 2 | 3}] [, /INTERPOLATE] [, /LIGHTING_MODEL]
 [, OPACITY_TABLE0=*byte array*] [, OPACITY_TABLE1=*byte array*]
 [, RENDER_EXTENTS={0 | 1 | 2}] [, RENDER_STEP=[*x*, *y*, *z*]]
 [, RENDER_QUALITY={1 | 2}] [, RGB_TABLE0=*byte array*]
 [, RGB_TABLE1=*byte array*]
 [, SUBVOLUME=[*xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax*]] [, /TWO_SIDED]
 [, VOLUME_DIMENSIONS=[*width*, *height*, *depth*]]
 [, VOLUME_LOCATION=[*x*, *y*, *z*]] [, /ZBUFFER]
 [, ZERO_OPACITY_SKIP={0 | 1}]

Axis Keywords: [, [XYZ]GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}]
 [, [XYZ]MAJOR=*integer*] [, [XYZ]MINOR=*integer*]
 [, [XYZ]RANGE=[*min, max*] [, [XYZ]SUBTICKLEN=*ratio*]
 [, [XYZ]TEXT_COLOR=*RGB vector*]
 [, [XYZ]TICKFONT_INDEX={0 | 1 | 2 | 3 | 4}]
 [, [XYZ]TICKFONT_SIZE=*integer*]
 [, [XYZ]TICKFONT_STYLE={0 | 1 | 2 | 3}]
 [, [XYZ]TICKFORMAT=*string or string array*]
 [, [XYZ]TICKINTERVAL=*value*] [, [XYZ]TICKLAYOUT={0 | 1 | 2}]
 [, [XYZ]TICKLEN=*value*] [, [XYZ]TICKNAME=*string array*]
 [, [XYZ]TICKUNITS=*string*] [, / [XYZ]TICKVALUES=*vector*]
 [, [XYZ]TITLE=*string*]

Arguments

Note

The volume data provided in the Vol_0 , Vol_1 , Vol_2 , and Vol_3 arguments are scaled into byte values (ranging from 0 to 255) with the BYTSCL function to facilitate using the volume data as indices into the RGB and OPACITY tables. This scaling is done for display purposes only; the iVolume tool maintains the original data as supplied with the arguments for use in other operations. The minimum and maximum values used by the BYTSCL function may be adjusted in the volume's property sheet. By default, the tool uses the minimum and maximum values of all volume parameters to uniformly byte-scale the data.

Vol_0 , Vol_1 , Vol_2 , Vol_3

A three-dimensional array of any numeric type containing volume data. Arrays of strings, structures, object references, and pointers are not allowed. If more than one volume is specified, they must all have the same dimensions.

The number of volumes present and the value of the COMPOSITE_FUNCTION keyword determine how the volume data is rendered by the iVolume tool. The number of volume arguments determine how the `src` and `srcalpha` values for the COMPOSITE_FUNCTION are computed:

- If Vol_0 is the only argument present, the values of `src` and `srcalpha` are taken directly from the RGB and OPACITY tables, as indexed by each volume data sample:

```
src = RGB_TABLE0[VOL0]
srcalpha = OPACITY_TABLE0[VOL0]
```

- If Vol_0 and Vol_1 are the only arguments present, the two volumes are blended together using independent tables:

```
src = (RGB_TABLE0[VOL0]*RGB_TABLE1[VOL1])/256
srcalpha = (OPACITY_TABLE0[VOL0]*OPACITY_TABLE1[VOL1])/256
```

- If all the arguments are present, Vol_0 indexes the red channel of RGB_TABLE0, Vol_1 indexes the green channel of RGB_TABLE0, and Vol_2 indexes the blue channel of RGB_TABLE0. The Vol_3 argument indexes OPACITY_TABLE0:

```
src = (RGB_TABLE[VOL0, 0], RGB_TABLE[VOL1, 1], $
      RGB_TABLE[VOL2, 2])/256
srcalpha = (OPACITY_TABLE0[VOL3])/256.
```

Note

If all the arguments are present, the composite function cannot be set to the average-intensity projection (COMPOSITE_FUNCTION = 3).

Keywords

Note

Keywords to the IVOLUME routine that correspond to the names of *registered properties* of the iVolume tool must be specified in full, without abbreviation.

AMBIENT

Use this keyword to set the color and intensity of the volume's base ambient lighting. Color is specified as an RGB vector. The default is [255, 255, 255]. AMBIENT is applicable only when LIGHTING_MODEL is set.

AUTO_RENDER

Set this keyword to 1 to always render the volume. The default is to not render the volume each time the tool window is drawn.

BACKGROUND_COLOR

Set this keyword to an RGB value specifying the color to be used as the background color for the view. The default is [255, 255, 255] (white). The BACKGROUND_COLOR keyword can be used when a tool is being created or when a new visualization is being created in an existing tool with the use of the OVERPLOT, VIEW_NUMBER or VIEW_NEXT keywords. The background color

is applied to the current view. For example, if multiple views have been created with the `VIEW_GRID` keyword, and the `VIEW_NUMBER` keyword is used to create a visualization in the second view, use of the `BACKGROUND_COLOR` keyword would set the background color in the second view only.

BOUNDS

Set this keyword to a six-element vector of the form $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$, which represents the sub-volume to be rendered. This keyword is the same as the `SUBVOLUME` keyword.

CLIP_PLANES

Set this keyword to an array of dimensions $[4, N]$ specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this keyword is a scalar (-1) indicating that no clipping planes are to be applied.

Note

Clipping planes are equivalent to the obsolete cutting planes. If defined, `CUTTING_PLANES` will be applied first, then the `CLIP_PLANES` (until a maximum number of planes is reached).

Note

A window is only able to support a limited number of clipping planes. Some of these clipping planes may already be in use by the tool to support specific data display features. If the total number of clipping planes exceeds the limit, an informational message is displayed.

COMPOSITE_FUNCTION

The composite function determines the value of a pixel on the viewing plane by analyzing the voxels falling along the corresponding ray, according to one of the following compositing functions:

- `0 = Alpha (default): Alpha-blending.` The recursive equation $dest' = src * sr calpha + dest * (1 - sr calpha)$ is used to compute the final pixel color.
- `1 = MIP: Maximum intensity projection.` The value of each pixel on the viewing plane is set to the brightest voxel, as determined by its opacity. The

most opaque voxel's color appropriation is then reflected by the pixel on the viewing plane.

- 2 = Alpha sum: Alpha-blending. The recursive equation

$$\text{dest}' = \text{src} + \text{dest} * (1 - \text{srcalpha})$$

is used to compute the final pixel color. This equation assumes that the color tables have been pre-multiplied by the opacity tables. The accumulated values can be no greater than 255.

- 3 = Average: Average-intensity projection. The resulting image is the average of all voxels along the corresponding ray.

Note

This option (COMPOSITE_FUNCTION = 3) is not supported for 4-channel volumes.

COMPOSITE_FUNCTION settings are visible only when RENDER_QUALITY is set to 2 (High).

DEPTH_CUE

Set this keyword to a two-element floating-point array [*zbright*, *zdim*] specifying the near and far Z planes between which depth cueing is in effect.

Depth cueing causes an object to appear to fade into the background color of the view object with changes in depth. If the depth of an object is further than *zdim* (that is, if the object's location in the Z direction is farther from the origin than the value specified by *zdim*), the object will be painted in the background color.

Similarly, if the object is closer than the value of *zbright*, the object will appear in its "normal" color. Anywhere in-between, the object will be a blend of the background color and the object color. For example, if the DEPTH_CUE property is set to [-1, 1], an object at the depth of 0.0 will appear as a 50% blend of the object color and the view color.

The relationship between Z_{bright} and Z_{dim} determines the result of the rendering:

- $Z_{bright} < Z_{dim}$: Rendering darkens with depth.
- $Z_{bright} > Z_{dim}$: Rendering brightens with depth.
- $Z_{bright} = Z_{dim}$: Disables depth cueing.

You can disable depth cueing by setting $z_{bright} = z_{dim}$. The default is [0.0, 0.0].

DIMENSIONS

Set this keyword to a two-element vector of the form [width, height] to specify the dimensions of the drawing area of the specific tool in device units. The minimum width of the window correlates to the width of the menubar. The minimum window height is 100 pixels.

EXTENTS_TRANSPARENCY

Set this keyword to an integer that specifies the percent transparency of the volume's boundary wire frame or walls.

HIDE

Set this keyword to a boolean value indicating whether the volume should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

HINTS

Set this keyword to specify one of the following acceleration hints:

- 0 = Disables all acceleration hints (default).
- 1 = Enables Euclidean distance map (EDM) acceleration. This option generates a volume map containing the distance from any voxel to the nearest non-zero opacity voxel. The map is used to speed ray casting by allowing the ray to jump over open spaces. It is most useful with sparse volumes. After setting the EDM hint, the draw operation generates the volume map; this process can take some time. Subsequent draw operations will reuse the generated map and may be much faster, depending on the volume's sparseness. A new map is not automatically generated to match changes in opacity tables or volume data (for performance reasons). The user may force recomputation of the EDM map by setting the HINTS property to 1 again.
- 2 = Enables the use of multiple CPUs for volume rendering if the platforms used support such use. If HINTS is set to 2, IDL will use all the available (up to 8) CPUs to render portions of the volume in parallel.
- 3 = Selects the two acceleration options described above.

IDENTIFIER

Set this keyword to a named IDL variable that will contain the `iToolID` for the created tool. This value can then be used to reference this tool during overplotting operations or command-line-based tool management operations.

INTERPOLATE

Set this keyword to indicate that trilinear interpolation is to be used to determine the data value for each step on a ray. Setting this keyword improves the quality of images produced, at the cost of more computing time, especially when the volume has low resolution with respect to the size of the viewing plane. Nearest neighbor sampling is used by default.

LIGHTING_MODEL

Set this keyword to use the current lighting model during rendering in conjunction with a local gradient evaluation.

Note

Only `DIRECTIONAL` light sources are honored by the volume object. Because normals must be computed for all voxels in a lighted view, enabling light sources increases the rendering time.

LOCATION

Set this keyword to a two-element vector of the form $[x, y]$ to specify the location of the upper left-hand corner of the tool relative to the display screen, in device units.

Note

Some X Window managers explicitly ignore any request from the client for window placement. See [“Positioning Top-Level Bases”](#) on page 2254 for additional information.

MACRO_NAMES

Set this keyword to a scalar string or an array of strings that specifies the names of one or more macros to run. The macro names are retrieved and the macros are run sequentially after the `iTool` and (if applicable) any visualizations have been created. If a macro of the specified name does not exist, IDL generates an error and the routine exits.

NAME

Set this keyword to a string to specify the name for this particular tool. The name is used for tool-related display purposes only—as the root of the hierarchy shown in the Tool Browser, for example.

NO_SAVEPROMPT

Set this keyword to cause the iTool *not* to prompt the user to save changes when closing the tool. The default is to prompt the user to save changes.

OPACITY_TABLE0

Set this keyword to a 256-element byte array to specify an opacity table for Vol_0 if Vol_0 or Vol_0 and Vol_1 are present. If all the volume arguments are present, this keyword represents the opacity of the resulting RGBA volume. A value of 0 indicates complete transparency and a value of 255 indicates complete opacity. The default table is a linear ramp.

OPACITY_TABLE1

Set this keyword to a 256-element byte array to specify an opacity table for Vol_1 when Vol_0 and Vol_1 are present. A value of 0 indicates complete transparency and a value of 255 indicates complete opacity. The default table is a linear ramp.

OVERPLOT

Set this keyword to an iToolID to direct the graphical output of the particular tool to the tool specified by the provided iToolID.

Set this keyword to 1 (one) to place the graphical output for the command in the current tool. If no current tool exists, a new tool is created.

RENDER_EXTENTS

Set this keyword to draw a boundary around the rendered volume. The default (RENDER_EXTENTS = 2) is to draw a translucent boundary box. Possible values for this keyword are:

- 0 = Do not draw anything around the volume.
- 1 = Draw a wireframe around the volume.
- 2 = Draw a translucent box around the volume

RENDER_STEP

Set this keyword to a three element vector of the form $[x, y, z]$ to specify the stepping factor through the voxel matrix. This keyword is only valid if render quality is set to high (`RENDER_QUALITY = 2`). The default render step is $[1, 1, 1]$.

RENDER_QUALITY

Set this keyword to determine the quality of the rendered volume. The default (`RENDER_QUALITY = 1`) is low quality. Possible values for this keyword are:

- 1 = Low - Renders volume with a stack of two-dimensional texture maps.
- 2 = High - Use ray-casting rendering, see the `COMPOSITE_FUNCTION` for more details.

RGB_TABLE0

Set this keyword to a 3 by 256 or 256 by 3 byte array of RGB color values to specify a color table for Vol_0 if Vol_0 or Vol_0 and Vol_1 are present. If all the arguments are present, this keyword represents the RGB color values of all of these volumes. The default is a linear ramp

RGB_TABLE1

Set this keyword to a 3 by 256 or 256 by 3 byte array of RGB color values to specify a color table for Vol_1 when Vol_0 and Vol_1 are present. The default is a linear ramp.

STYLE_NAME

Set this keyword equal to a string that specifies the name of a user-defined or a system style. If that style does not exist then no style is applied. The style is applied using the following rules:

- If the tool exists and `/OVERPLOT` is specified, then the style is only applied to the newly-created visualizations within the current view. The current tool style is not updated with the new style, nor is the style applied to any other items within the view.
- Otherwise, if the tool exists and either `VIEW_NEXT` or `VIEW_NUMBER` is being used to select a different view, then the style is applied to all items within that view. The current tool style is updated with the new style.
- Otherwise, if a new tool is being created, then the style is applied to all items within all views. The current tool style is updated with the new style.

SUBVOLUME

Set this keyword to a six-element vector of the form $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$, which represents the sub-volume to be rendered. This keyword is the same as the BOUNDS keyword.

TITLE

Set this keyword to a string to specify the title for this particular tool. The title is displayed in the title bar of the tool.

TWO_SIDED

Set this keyword to force the lighting model to use a two-sided voxel gradient. The two-sided gradient is different from the one-sided gradient (default) in that the absolute value of the inner product of the light direction and the surface gradient is used instead of clamping to 0.0 for negative values.

VIEW_GRID

Set this keyword to a two-element vector of the form $[columns, rows]$ to specify the view layout within the new tool. This keyword is only used if a new tool is being created (for example, if OVERPLOT, VIEW_NEXT, or VIEW_NUMBER are specified then VIEW_GRID is ignored).

VIEW_NEXT

Set this keyword to change the view selection to the next view following the currently-selected view before issuing any graphical commands. If the currently-selected view is the last one in the layout, then /VIEW_NEXT will cause the first view in the layout to become selected. This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless /OVERPLOT is set.

VIEW_NUMBER

Set this keyword to change the currently-selected view to the view specified by the VIEW_NUMBER before issuing any graphical commands. The view number starts at 1, and corresponds to the position of the view within the graphics container (not necessarily the position on the screen). This keyword is ignored if no current tool exists.

Note

The contents of the newly-selected view will be emptied unless `/OVERPLOT` is set.

VOLUME_DIMENSIONS

A 3-element vector specifying the volume dimensions in terms of user data units. For example, specifying `[0.1, 0.1, 0.1]` would cause the volume to be rendered into a region that is 0.1 data units long on each side of the volume cube. If this parameter is not specified, the volume is rendered into a region the same size as the number of samples, with an origin of `[0, 0, 0]`. In this case, a volume with sample size of `[20, 25, 20]` would render into the region `[0:19, 0:24, 0:19]` in user data units. Use the `VOLUME_LOCATION` keyword to specify a different origin.

VOLUME_LOCATION

A 3-element vector specifying the volume location in user data units. Use this keyword to render the volume so that the first sample voxel appears at the specified location, instead of at `[0, 0, 0]`, the default. Specify the location in terms of coordinates after the application of the `VOLUME_DIMENSIONS` values. For example, if the value of the `VOLUME_DIMENSIONS` keyword is `[0.1, 0.1, 0.1]` and you want the volume to be centered at the origin, set the `VOLUME_LOCATION` keyword to `[-0.05, -0.05, -0.05]`.

[XYZ]GRIDSTYLE

The index of the linestyle to be used for plot tickmarks and grids (i.e., when `[XYZ]TICKLEN` is set to 1.0). See `LINestyle` for a list of linestyles.

[XYZ]MAJOR

Set this keyword to an integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting `MAJOR` equal to zero suppresses major tickmarks entirely.

[XYZ]MINOR

Set this keyword to an integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting `MINOR` equal to zero suppresses minor tickmarks entirely.

[XYZ]RANGE

Set this keyword to the desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum.

[XYZ]SUBTICKLEN

Set this keyword to a floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

[XYZ]TEXT_COLOR

Set this keyword to an RGB value specifying the color for the axis text. The default value is [0, 0, 0] (black).

[XYZ]TICKFONT_INDEX

Set this keyword equal to one of the following integers, which represent the type of font to be used for the axis text:

- 0 = Helvetica
- 1 = Courier
- 2 = Times
- 3 = Symbol
- 4 = Hershey

[XYZ]TICKFONT_SIZE

Set this keyword to an integer representing the point size of the font used for the axis text. The default is 12.0 points.

[XYZ]TICKFONT_STYLE

Set this keyword equal to one of the following integers, which represent the style of font to be used for the axis text:

- 0 = Normal
- 1 = Bold
- 2 = Italic
- 3 = Bold Italic

[XYZ]TICKFORMAT

Set this keyword to a string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. The TICKUNITS keyword determines the number of levels for an axis.

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether TICKUNITS is specified.

If TICKUNITS are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If TICKUNITS are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Note

For more information, see “[[XYZ](#)]TICKFORMAT” on page 4345.

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

[XYZ]TICKINTERVAL

Set this keyword to a floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis [XYZ]RANGE and the number of major tick marks ([XYZ]MAJOR). The value of this keyword takes precedence over the value set for the [XYZ]MAJOR keyword.

For example, if `TICKUNITS = ['S', 'H', 'D']`, and `TICKINTERVAL = 30`, then the interval between major ticks for the first axis level will be 30 seconds.

[XYZ]TICKLAYOUT

Set this keyword to integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Set this keyword to a floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TICKNAME

Set this keyword to a string array of up to 30 elements that controls the annotation of each tick mark.

[XYZ]TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling. If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- "" - Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS = 'Day' is equivalent to TICKUNITS = 'Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 C.E., respectively.

[XYZ]TICKVALUES

Set this keyword to a floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

[XYZ]TITLE

Set this keyword to a string representing the title of the specified axis.

ZBUFFER

Set this keyword to clip the rendering to the current Z-buffer and then update the buffer.

ZERO_OPACITY_SKIP

Set this keyword to skip voxels with an opacity of 0. This keyword can increase the output contrast of MIP (MAXIMUM_INTENSITY) projections by allowing the background to show through. If this keyword is set, voxels with an opacity of zero will not modify the Z-buffer. The default (not setting the keyword) continues to render voxels with an opacity of zero.

Obsolete Keywords

The following keyword is obsolete:

- CUTTING_PLANE

For information on obsolete keywords, see [Appendix I, “Obsolete Features”](#).

Examples

In the IDL Intelligent Tools system, data can be imported from the IDL Command Line (as described in Example 1), or data can be imported via the **File** menu in the iTool window (as described in Examples 2 and 3). For detailed information on importing data via the iTool file menu, refer to “[Data Import Methods](#)” in Chapter 2 of the *iTool User’s Guide* manual.

Example 1

This example shows how to use the IDL Command Line to bring data into the iVolume tool.

At the IDL Command Line, enter:

```
file = FILEPATH('clouds3d.dat', $
  SUBDIRECTORY = ['examples', 'data'])
RESTORE, file
IVOLUME, clouds
```

Derive an interval volume by selecting **Operations** → **Volume** → **Interval Volume**. In the Interval Volume Value Selector dialog, change the minimum value to 0.2 and the **Decimate: % of original surface** slider to 20, then click **OK**.

The following figure displays the output of this example:

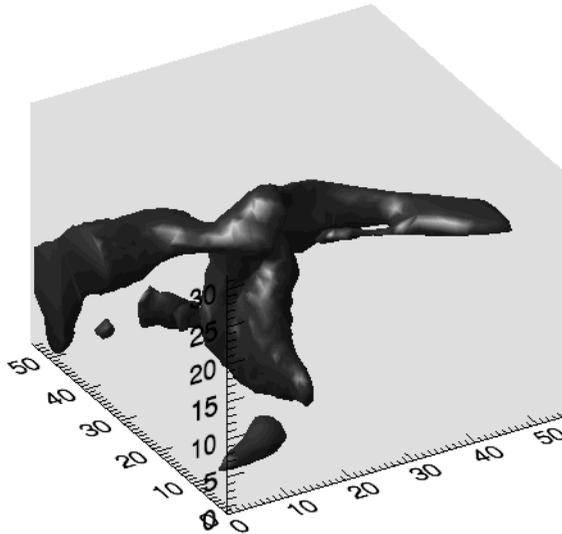


Figure 3-38: Cloud Interval Volume *iVolume* Example

Example 2

This example shows how to use the *iTool* **File** → **Open** command to load binary data into the *iVolume* tool.

At the IDL Command Line, enter:

```
IVOLUME
```

Select **File** → **Open** to display the Open dialog, then browse to find `head.dat` in the `examples/data` directory in the IDL distribution, and click **Open**.

In the Binary Template dialog, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** data (or a name of your choosing)
- **Type:** Byte (unsigned 8-bits)
- **Number of Dimensions:** 3
- **1st Dimension Size:** 80

- **2nd Dimension Size:** 100
- **3rd Dimension Size:** 57

Click **OK** to close the New Field dialog and the Binary Template dialog, and the image is displayed.

Note

For more information on using the Binary Template to import data, see “Using the BINARY_TEMPLATE Function” in Chapter 15 of the *Using IDL* manual.

Select **Operations** → **Volume** → **Isosurface**, and insert an isosurface with a value of 60, decimated to 20% of the original surface.

The following figure displays the output of this example:

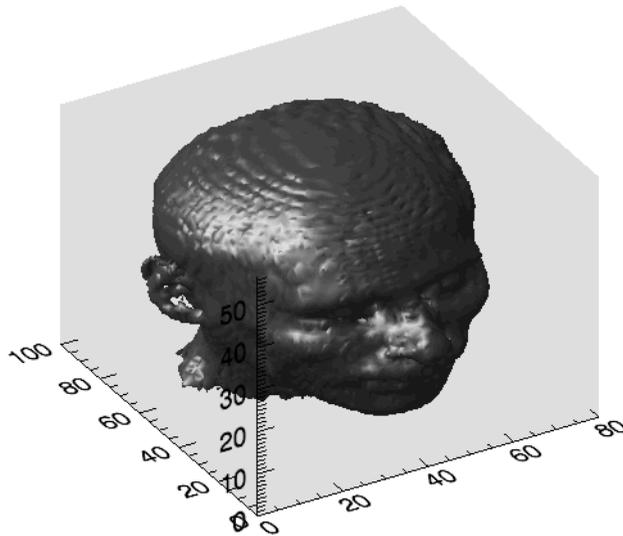


Figure 3-39: Human Head MRI Isosurface iVolume Example

Example 3

This example shows how to use the **File** → **Import** command to load binary data into the iVolume tool.

At the IDL Command Line, enter:

```
IVOLUME
```

Select **File** → **Import** to display the IDL Import Data wizard.

1. At Step 1, select **From a File** and click **Next>>**.
2. At Step 2, under **File Name:**, browse to find `jet.dat` in the `examples/data` directory in the IDL distribution, and click **Next>>**.
3. At Step 3, select **Volume** and click **Finish**.

The Binary Template wizard is displayed. In the Binary Template, change **File's byte ordering** to `Little Endian`. Then, click **New Field**, and enter the following information in the New Field dialog:

- **Field Name:** `data` (or a name of your choosing)
- **Type:** `Byte (unsigned 8-bits)`
- **Number of Dimensions:** `3`
- **1st Dimension Size:** `81`
- **2nd Dimension Size:** `40`
- **3rd Dimension Size:** `101`

Click **OK** to close the New Field dialog and the Binary Template dialog, and the volume is displayed.

Select **Operations** → **Volume** → **Image Plane** to display a plane in the x -direction. Double-click on the plane to access its properties through the property sheet. Change the **Orientation** setting to `z`. You can drag the image to see it at different z values by clicking on the edge of the image plane.

The following figure displays the output of this example:

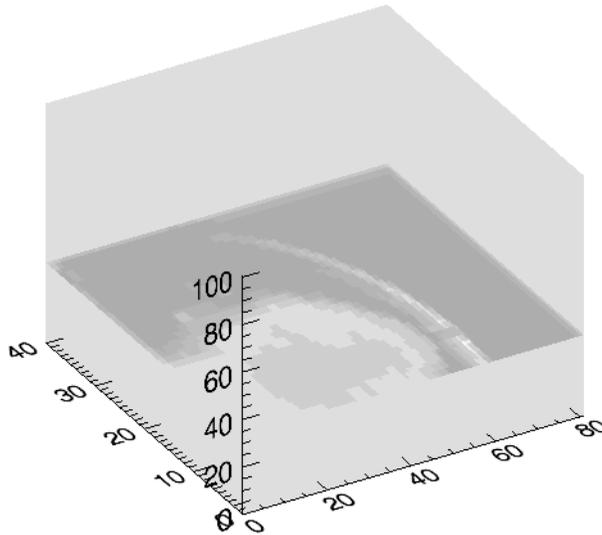


Figure 3-40: Plasma Jet Image Plane *i*Volume Example

Example 4

This example shows how to use a second volume argument to cut away a section of the first volume argument.

First, load the MRI head data into IDL. At the IDL Command Line, enter:

```
file = FILEPATH('head.dat', SUBDIRECTORY = ['examples', 'data'])
data0 = READ_BINARY(file, DATA_DIMS = [80, 100, 57])
```

Then, create the second volume that will cut away the upper left corner of the head.

At the IDL Command Line, enter:

```
data1 = BYTARR(80, 100, 57) + 1B
data1[0:39, *, 28:56] = 0B
```

Derive the color and opacity tables for the second volume. At the IDL Command Line, enter:

```
rgbTable1 = [[BYTARR(256)], [BYTARR(256)], [BYTARR(256)]]
rgbTable1[1, *] = [255, 255, 255]
opacityTable1 = BYTARR(256)
opacityTable1[1] = 255
```

Now, display the two volumes. At the IDL Command Line, enter:

```
IVOLUME, data0, data1, RGB_TABLE1 = rgbTable1, $
      OPACITY_TABLE1 = opacityTable1, /AUTO_RENDER
```

The following figure displays the output of this example:

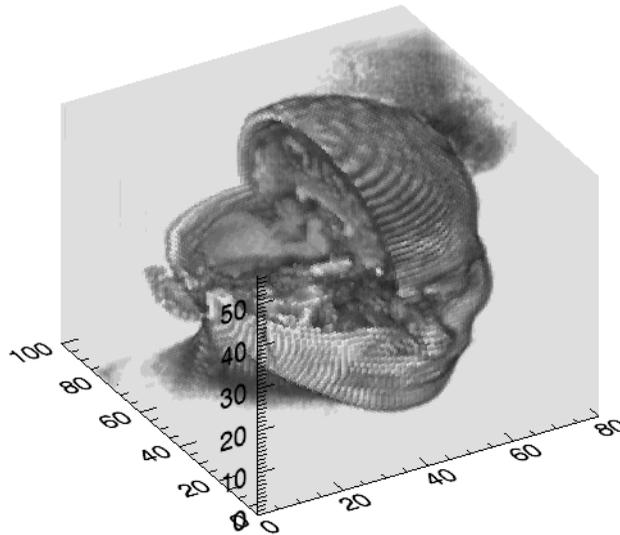


Figure 3-41: Cut Away iVolume Example

Example 5

This example shows how to use all the volume arguments to display an RGB (Red, Green, Blue) volume.

First, create the volumes to contain primary colors (black, red, green, blue, yellow, cyan, magenta, and white) in each corner. At the IDL Command Line, enter:

```
vol0 = BYTARR(32, 32, 32)
vol1 = BYTARR(32, 32, 32)
vol2 = BYTARR(32, 32, 32)
vol3 = BYTARR(32, 32, 32)
vol0[0:15, *, *] = 255
vol1[* , 0:15, *] = 255
vol2[* , *, 0:15] = 255
vol3[* , *, *] = 128
```

Then, derive the color and opacity tables. At the IDL Command Line, enter:

```
loadct,15
tv!ct,r,g,b,/get
rgbtable=BINDGEN(3,256)
rgbtable[0,*]=r
rgbtable[1,*]=g
rgbtable[2,*]=b
opacityTable = BINDGEN(256)
```

Now, display the two volumes. At the IDL Command Line, enter:

```
IVOLUME, vol0, vol1, vol2, vol3, RGB_TABLE0 = rgbTable, $
    OPACITY_TABLE0 = opacityTable, /AUTO_RENDER
```

The following figure displays the output of this example:

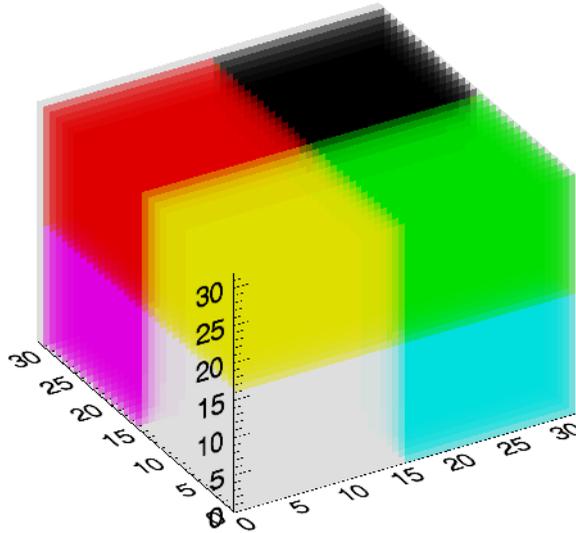


Figure 3-42: RGB iVolume Example

Note

The white corner of this example volume is actually gray to distinguish it from the white background.

Version History

Introduced: 6.0

BACKGROUND_COLOR, EXTENTS_TRANSPARENCY, MACRO_NAMES,
STYLE_NAME keywords added: 6.1

CUTTING_PLANE obsoleted: 6.1

JOURNAL

The JOURNAL procedure provides a record of an interactive session by saving, in a file, all text entered from the terminal in response to the IDL prompt. The first call to JOURNAL starts the logging process. The read-only system variable !JOURNAL is set to the file unit used. To stop saving commands and close the file, call JOURNAL with no parameters. If logging is in effect and JOURNAL is called with a parameter, the parameter is simply written to the journal file.

Syntax

```
JOURNAL [, Arg]
```

Arguments

Arg

A string containing the name of the journal file to be opened or text to be written to an open journal file. If *Arg* is not supplied, and a journal file is not already open, the file `idlsave.pro` is used. Once journaling is enabled, a call to JOURNAL with *Arg* supplied causes *Arg* to be written into the journal file. Calling JOURNAL without *Arg* while journaling is in progress closes the journal file and ends the logging process.

Keywords

None.

Examples

To begin journaling to the file `myjournal.pro`, enter:

```
JOURNAL, 'myjournal.pro'
```

Any commands entered at the IDL prompt are recorded in the file until IDL is exited or the JOURNAL command is entered without an argument.

Version History

Introduced: Original

See Also

[RESTORE](#), [SAVE](#)

JULDAY

The JULDAY function calculates the Julian Day Number (which begins at noon) for the specified date. This is the inverse of the CALDAT procedure.

Note

The Julian calendar, established by Julius Caesar in the year 45 BCE, was corrected by Pope Gregory XIII in 1582, excising ten days from the calendar. The CALDAT procedure reflects the adjustment for dates after October 4, 1582. See the example below for an illustration.

Note

A small offset is added to the returned Julian date to eliminate roundoff errors when calculating the day fraction from hours, minutes, seconds. This offset is given by the larger of EPS and EPS*Julian, where Julian is the integer portion of the Julian date, and EPS is the EPS field from MACHAR. For typical Julian dates, this offset is approximately 6×10^{-1} (which corresponds to 5×10^{-5} seconds). This offset ensures that if the Julian date is converted back to hour, minute, and second, then the hour, minute, and second will have the same integer values as were originally input.

Note

Calendar dates must be in the range 1 Jan 4716 B.C.E. to 31 Dec 5000000, which corresponds to Julian values -1095 and 1827933925, respectively.

This routine is written in the IDL language. Its source code can be found in the file `julday.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = JULDAY(*Month*, *Day*, *Year*, *Hour*, *Minute*, *Second*)

Return Value

Result is of type double-precision if Hour, Minute, or Second is specified, otherwise *Result* is of type long integer. If all arguments are scalar, the function returns a scalar. If all arguments are arrays, the function matches up the corresponding elements of the arrays, returning an array with the same dimensions as the smallest array. If the inputs contain both scalars and arrays, the function uses the scalar value with each element of the arrays, and returns an array with the same dimensions as the smallest input array.

Arguments

Month

Number of the desired month (1 = January, ..., 12 = December). *Month* can be either a scalar or an array.

Day

Number of the day of the month (1-31). *Day* can be either a scalar or an array.

Year

Number of the desired year (e.g., 1994). *Year* can be either a scalar or an array.

Hour

Number of the hour of the day (0-23). *Hour* can be either a scalar or an array.

Minute

Number of the minute of the hour (0-59). *Minute* can be either a scalar or an array.

Second

Number of the second of the minute (0-59). *Second* can be either a scalar or an array.

Examples

In 1582, Pope Gregory XIII adjusted the Julian calendar to correct for its inaccuracy of slightly more than 11 minutes per year. As a result, the day following October 4, 1582 was October 15, 1582. `JULDAY` follows this convention, as illustrated by the following commands:

```
PRINT, JULDAY(10,4,1582), JULDAY(10,5,1582), JULDAY(10,15,1582)
```

IDL prints:

```
2299160      2299161      2299161
```

Using arrays, this can also be calculated as follows:

```
PRINT, JULDAY(10, [4, 5, 15], 1582)
```

If you are using `JULDAY` to calculate an absolute number of days elapsed, be sure to account for the Gregorian adjustment.

Version History

Introduced: Original

See Also

[BIN_DATE](#), [CALDAT](#), [SYSTIME](#)

KEYWORD_SET

The `KEYWORD_SET` function returns a Boolean value based on the value of the specified expression. It returns a True (1) if its argument is defined and nonzero, and False (0) otherwise. The exact rules used to determine this are given in [Return Value](#) below.

Syntax

Result = `KEYWORD_SET(Expression)`

Return Value

This function returns True (1) if:

- *Expression* is a scalar or 1-element array with a non-zero value.
- *Expression* is a structure.
- *Expression* is an ASSOC file variable.

`KEYWORD_SET` returns False (0) if:

- *Expression* is undefined.
- *Expression* is a scalar or 1-element array with a zero value.

Arguments

Expression

The expression to be tested. *Expression* is usually a named variable.

Keywords

None.

Examples

Suppose that you are writing an IDL procedure that has the following procedure definition line:

```
PRO myproc, KEYW1 = keyw1, KEYW2 = keyw2
```

The following command could be used to execute a set of commands only if the keyword `KEYW1` is set (i.e., it is present and nonzero):

```
IF KEYWORD_SET(keyw1) THEN BEGIN
```

The commands to be executed only if `KEYW1` is set would follow.

Version History

Introduced: Original

See Also

[ARG_PRESENT](#), [LOGICAL_TRUE](#), [N_ELEMENTS](#), [N_PARAMS](#)

KRIG2D

The KRIG2D function interpolates a regularly- or irregularly-gridded set of points $z = f(x, y)$ using kriging.

The parameters of the data model – the range, nugget, and sill – are highly dependent upon the degree and type of spatial variation of your data, and should be determined statistically. Experimentation, or preferably rigorous analysis, is required.

For n data points, a system of $n+1$ simultaneous equations are solved for the coefficients of the surface. For any interpolation point, the interpolated value is:

$$f(x, y) = \sum w_i \cdot C(x_i, y_i, x, y)$$

The following formulas are used to model the variogram functions:

$d(i, j)$ = the distance from point i to point j .

V = the variance of the samples.

$C(i, j)$ = the covariance of sample i with sample j .

$C(x_0, y_0, x_1, y_1)$ = the covariance of point (x_0, y_0) with point (x_1, y_1) .

Exponential covariance:

$$C(d) = \begin{cases} C_1 \cdot e^{(-3 \cdot d/A)} & \text{if } d \neq 0 \\ C_1 + C_0 & \text{if } d = 0 \end{cases}$$

Spherical covariance:

$$C(d) = \begin{cases} 1.0 - (1.5 \cdot d/A) + (0.5 \cdot (d/A)^3) & \text{if } d < a \\ C_1 + C_0 & \text{if } d = 0 \\ 0 & \text{if } d > a \end{cases}$$

Note

The accuracy of this function is limited by the single-precision floating-point accuracy of the machine.

This routine is written in the IDL language. Its source code can be found in the file `krig2d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = KRIG2D( Z [, X, Y] [, EXPONENTIAL=vector] [, SPHERICAL=vector]
[, /REGULAR] [, XGRID=[xstart, xspacing]] [, XVALUES=array]
[, YGRID=[ystart, yspacing]] [, YVALUES=array] [, GS=[xspacing, yspacing]]
[, BOUNDS=[xmin, ymin, xmax, ymax]] [, NX=value] [, NY=value] )
```

Return Value

Returns a two dimensional floating-point array containing the interpolated surface, sampled at the grid points.

Arguments

Z, X, Y

Arrays containing the Z, X, and Y coordinates of the data points on the surface. Points need not be regularly gridded. For regularly gridded input data, X and Y are not used: the grid spacing is specified via the XGRID and YGRID (or XVALUES and YVALUES) keywords, and Z must be a two dimensional array. For irregular grids, all three parameters must be present and have the same number of elements.

Keywords

Model Parameter Keywords:

EXPONENTIAL

Set this keyword to a two- or three-element vector of model parameters [A,C0, C1] to use an exponential semivariogram model. The model parameters are as follows:

- A — The *range*. At distances beyond A, the semivariogram or covariance remains essentially constant.
- C0 — The *nugget*, which provides a discontinuity at the origin.
- C1 — If specified, C1 is the covariance value for a zero distance, and the variance of the random sample z variable. If only a two element vector is supplied, C1 is set to the sample variance. (C0 + C1) = the *sill*, which is the variogram value for very large distances.

SPHERICAL

Set this keyword to a two- or three-element vector of model parameters [A, C0, C1] to use a spherical semivariogram model. The model parameters are as follows:

- A — The *range*. At distances beyond A, the semivariogram or covariance remains essentially constant.
- C0 — The *nugget*, which provides a discontinuity at the origin.
- C1 — If specified, C1 is the covariance value for a zero distance, and the variance of the random sample *z* variable. If only a two element vector is supplied, C1 is set to the sample variance. $(C0 + C1)$ = the *sill*, which is the variogram value for very large distances.

Input Grid Keywords:

REGULAR

If set, the Z parameter is a two dimensional array of dimensions (n,m) , containing measurements over a regular grid. If any of XGRID, YGRID, XVALUES, or YVALUES are specified, REGULAR is implied. REGULAR is also implied if there is only one parameter, Z. If REGULAR is set, and no grid specifications are present, the grid is set to (0, 1, 2, ...).

XGRID

A two-element array, [*xstart*, *xspacing*], defining the input grid in the *x* direction. Do not specify both XGRID and XVALUES.

XVALUES

An *n*-element array defining the *x* locations of $Z[i,j]$. Do not specify both XGRID and XVALUES.

YGRID

A two-element array, [*ystart*, *yspacing*], defining the input grid in the *y* direction. Do not specify both YGRID and YVALUES.

YVALUES

An *n*-element array defining the *y* locations of $Z[i,j]$. Do not specify both YGRID and YVALUES.

Output Grid Keywords:

BOUNDS

If present, BOUNDS must be a four-element array containing the grid limits in x and y of the output grid: $[xmin, ymin, xmax, ymax]$. If not specified, the grid limits are set to the extent of x and y .

GS

The output grid spacing. If present, GS must be a two-element vector $[xs, ys]$, where xs is the horizontal spacing between grid points and ys is the vertical spacing. The default is based on the extents of x and y . If the grid starts at x value $xmin$ and ends at $xmax$, then the default horizontal spacing is $(xmax - xmin)/(NX-1)$. ys is computed in the same way. The default grid size, if neither NX or NY are specified, is 26 by 26.

NX

The output grid size in the x direction. NX need not be specified if the size can be inferred from GS and BOUNDS. The default value is 26.

NY

The output grid size in the y direction. NY need not be specified if the size can be inferred from GS and BOUNDS. The default value is 26.

Examples

```

; Make a random set of points that lie on a Gaussian:
N = 15
X = RANDOMU(seed, N)
Y = RANDOMU(seed, N)

; The Gaussian:
Z = EXP(-2 * ((X-.5)^2 + (Y-.5)^2))

; Get a 26 by 26 grid over the rectangle bounding x and y:
; Range is 0.25 and nugget is 0. These numbers are dependent on
; your data model:
E = [ 0.25, 0.0]

; Get the surface:
R = KRIG2D(Z, X, Y, EXPON = E)

```

Alternatively, get a surface over the unit square, with spacing of 0.05:

```
R = KRIG2D(Z, X, Y, EXPON=E, GS=[0.05, 0.05], BOUNDS=[0,0,1,1])
```

Version History

Introduced: Pre 4.0

See Also

[BILINEAR](#), [INTERPOLATE](#)

KURTOSIS

The KURTOSIS function computes the statistical kurtosis of an n -element vector. Kurtosis is defined as the degree to which a statistical frequency curve is peaked. KURTOSIS calls the IDL function MOMENT.

Note

KURTOSIS subtracts 3 from the raw kurtosis value since 3 is the kurtosis for a Gaussian (normal) distribution. For resulting values, positive values of the kurtosis (leptokurtic) indicate pointed or peaked distributions. Negative values (platykurtic) indicate flattened or non-peaked distributions.

Syntax

Result = KURTOSIS(*X* [, /DOUBLE] [, /NAN])

Return Value

Returns the floating point or double precision statistical kurtosis. If the variance of the vector is zero, the kurtosis is not defined, and KURTOSIS returns !VALUES.F_NAN as the result.

Arguments

X

An n -element, floating-point or double-precision vector.

Keywords

DOUBLE

If this keyword is set, computations are performed in double precision arithmetic.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Examples

```
; Define the n-element vector of sample data:  
x = [65, 63, 67, 64, 68, 62, 70, 66, 68, 67, 69, 71, 66, 65, 70]  
; Compute the kurtosis:  
result = KURTOSIS(x)  
; Print the result:  
PRINT, result
```

IDL prints

```
-1.18258
```

Version History

Introduced: 5.1

See Also

[MEAN](#), [MEANABSDEV](#), [MOMENT](#), [STDDEV](#), [SKEWNESS](#), [VARIANCE](#)

KW_TEST

The KW_TEST function tests the hypothesis that three or more sample populations have the same mean of distribution against the hypothesis that they differ. The populations may be of equal or unequal lengths. The output is a vector containing the test statistic, H, and value indicating the probability of obtaining a value equal to or greater than H from a Chi-square distribution.

This test is an extension of the Rank Sum Test implemented in the RS_TEST function. When each sample population contains at least five observations, the H test statistic is approximated very well by a Chi-square distribution with DF degrees of freedom. The hypothesis that three or more sample populations have the same mean of distribution is rejected if two or more populations differ with statistical significance. This type of test is often referred to as the Kruskal-Wallis H-Test.

The test statistic H is defined as follows:

$$H = \frac{12}{N_T(N_T + 1)} \sum_{i=0}^{M-1} \frac{R_i^2}{N_i} - 3(N_T + 1)$$

where N_i is the number of observations in the i^{th} sample population, N_T is the total number of observations in all sample populations, and R_i is the overall rank sum of the i^{th} sample population.

This routine is written in the IDL language. Its source code can be found in the file `kw_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = KW_TEST(X [, DF=variable] [, MISSING=nonzero_value])

Return Value

The result is a two-element vector containing the test statistic H and the one-tailed probability of obtaining a value of H or greater from a Chi-square distribution.

Arguments

X

An integer, single-, or double-precision floating-point array of m -columns (with $m \geq 3$) and n -rows. The columns of this two-dimensional array correspond to the sample populations.

If the sample populations are of unequal length, any columns of X that are shorter than the longest column must be “filled in” by appending a user-specified missing data value. This method requires the use of the `MISSING` keyword. See the *Example* section below for an example of this case.

Keywords

DF

Use this keyword to specify a named variable that will contain the number of degrees of freedom used to compute the probability of obtaining a value of H or greater from the corresponding Chi-square distribution

MISSING

Set this keyword equal to a non-zero numeric value that has been appended to some columns of X to make them all a common length of n .

Examples

Test the hypothesis that three sample populations have the same mean of distribution against the hypothesis that they differ at the 0.05 significance level. Assume we have the following sample populations:

```
sp0 = [24.0, 16.7, 22.8, 19.8, 18.9]
```

```
sp1 = [23.2, 19.8, 18.1, 17.6, 20.2, 17.8]
```

```
sp2 = [18.2, 19.1, 17.3, 17.3, 19.7, 18.9, 18.8, 19.3]
```

Since the sample populations are of unequal lengths, a missing value must be appended to `sp0` and `sp1`. In this example the missing value is `-1.0` and the 3-column, 8-row input array `x` is defined as:

```
X = [[24.0, 23.2, 18.2], $  
     [16.7, 19.8, 19.1], $  
     [22.8, 18.1, 17.3], $  
     [19.8, 17.6, 17.3], $  
     [18.9, 20.2, 19.7], $  
     [-1.0, 17.8, 18.9], $  
     [-1.0, -1.0, 18.8], $  
     [-1.0, -1.0, 19.3]]  
PRINT, KW_TEST(X, MISSING = -1)
```

IDL prints:

```
[1.65862, 0.436351]
```

The computed probability (0.436351) is greater than the 0.05 significance level and therefore we do not reject the hypothesis that the three sample populations sp0, sp1, and sp2 have the same mean of distribution.

Version History

Introduced: 4.0

See Also

[FV_TEST](#), [RS_TEST](#), [S_TEST](#), [TM_TEST](#)

L64INDGEN

The L64INDGEN function creates a 64-bit integer array with the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = L64INDGEN(D_1 [, \dots, D_8])$$

Return Value

The return value is the specified 64-bit integer array.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create L, a 10-element by 10-element 64-bit array where each element is set to the value of its one-dimensional subscript, enter:

```
L = L64INDGEN(10, 10)
```

Version History

Introduced: 5.2

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#),
[LINDGEN](#), [SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

LA_CHOLDC

The LA_CHOLDC procedure computes the Cholesky factorization of an n -by- n symmetric (or Hermitian) positive-definite array as:

$$\text{If } A \text{ is real: } A = U^T U \text{ or } A = L L^T$$

$$\text{If } A \text{ is complex: } A = U^H U \text{ or } A = L L^H$$

where U and L are upper and lower triangular arrays. The T represents the transpose while H represents the Hermitian, or transpose complex conjugate.

LA_CHOLDC is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	spotrf
Double	dpotrf
Complex	cpotrf
Double complex	zpotrf

Table 3-49: LAPACK Routine Basis for LA_CHOLDC

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_CHOLDC, *Array* [, /DOUBLE] [, STATUS=*variable*] [, /UPPER]

Arguments

Array

A named variable containing the real or complex array to be factorized. Only the lower triangular portion of *Array* is used (or upper if the UPPER keyword is set). This procedure returns *Array* as a lower triangular array from the Cholesky decomposition (upper triangular if the UPPER keyword is set).

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if *Array* is double precision, otherwise the default is `DOUBLE = 0`.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: The array is not positive definite and the factorization could not be completed. The `STATUS` value specifies the order of the leading minor which is not positive definite.

Note

If `STATUS` is not specified, any error messages will output to the screen.

UPPER

If this keyword is set, then only the upper triangular portion of *Array* is used, and the upper triangular array is returned. The default is to use the lower triangular portion and to return the lower triangular array.

Examples

The following example program computes the Cholesky decomposition of a given symmetric positive-definite array:

```
PRO ExLA_CHOLDC
; Create a symmetric positive-definite array.
n = 10
seed = 12321
array = RANDOMU(seed, n, n)
array = array ## TRANSPOSE(Array)

; Compute the Cholesky decomposition.
lower = array      ; make a copy
LA_CHOLDC, lower
```

```
    ; Zero out the upper triangular portion.  
    for i = 0,n - 2 Do lower[i+1:*,i] = 0  
  
    ; Reconstruct the array and check the difference  
    arecon = lower ## TRANSPOSE(lower)  
    PRINT, 'LA_CHOLDC Error:', MAX(ABS(arecon - array))  
    END
```

When this program is compiled and run, IDL prints:

```
LA_CHOLDC Error:  
4.76837e-007
```

Version History

Introduced 5.6

See Also

[CHOLDC](#), [LA_CHOLMPROVE](#), [LA_CHOLSOL](#)

LA_CHOLMPROVE

The LA_CHOLMPROVE function uses Cholesky factorization to improve the solution to a system of linear equations, $AX = B$ (where A is symmetric or Hermitian), and provides optional error bounds and backward error estimates.

The LA_CHOLMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the improved solution vector for that set of equations.

LA_CHOLMPROVE is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sporfs
Double	dporfs
Complex	cporf
Double complex	zporfs

Table 3-50: LAPACK Routine Basis for LA_CHOLMPROVE

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_CHOLMPROVE( Array, Achol, B, X
  [, BACKWARD_ERROR=variable] [, /DOUBLE]
  [, FORWARD_ERROR=variable] [, /UPPER] )
```

Return Value

The result is an n -element vector or k -by- n array.

Arguments

Array

The original n -by- n array of the linear system $AX = B$.

Achol

The n -by- n Cholesky factorization of *Array*, created by the LA_CHOLDC procedure.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

X

An n -element input vector, or a k -by- n array, containing the approximate solutions to the linear system, created by the LA_CHOLSOL function.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If B is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If B is an array containing k linear systems, then BACKWARD_ERROR will be a k -element vector. The backward error is the smallest relative change in any element of A or B that makes X an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If B is a vector containing a single linear system, then FORWARD_ERROR will be a scalar. If B is an array containing k linear systems, then FORWARD_ERROR will be a k -element vector. For each linear system, if X_{true} is the true solution corresponding to X , then the forward error is an estimated upper bound for the magnitude of the largest element in $(X - X_{true})$ divided by the magnitude of the largest element in X .

UPPER

Set this keyword if *A* contains the upper triangular array, rather than the lower triangular array.

Note

If the UPPER keyword is set in LA_CHOLDC and LA_CHOLSOL then the UPPER keyword must also be set in LA_CHOLMPROVE.

Examples

The following example program computes an improved solution to a set of 10 equations:

```

PRO ExLA_CHOLMPROVE
; Create a symmetric positive-definite array.
n = 10
seed = 12321
a = RANDOMU(seed, n, n, /DOUBLE)
a = a ## TRANSPOSE(a)

; Create the right-hand side vector b:
b = RANDOMU(seed, n, /DOUBLE)

; Compute the Cholesky decomposition.
achol = a ; make a copy
LA_CHOLDC, achol

; Compute the first approximation to the solution:
x = LA_CHOLSOL(achol, b)

; Improve the solution and print the error estimate:
xmprove = LA_CHOLMPROVE(a, achol, b, x, $
    FORWARD_ERROR = fError)
PRINT, 'LA_CHOLMPROVE error:', $
    MAX(ABS(a ## xmprove - b))
PRINT, 'LA_CHOLMPROVE Error Estimate:', fError
END

```

When this program is compiled and run, IDL prints:

```

LA_CHOLMPROVE error:    3.9412917e-15
LA_CHOLMPROVE error estimate:    5.1265892e-12

```

Version History

Introduced 5.6

See Also

[LA_CHOLD](#), [LA_CHOLSOL](#)

LA_CHOLSOL

The LA_CHOLSOL function is used in conjunction with the LA_CHOLDC to solve a set of n linear equations in n unknowns, $AX = B$, where A must be a symmetric (or Hermitian) positive-definite array. The parameter A is input not as the original array, but as its Cholesky decomposition, created by the routine LA_CHOLDC.

The LA_CHOLSOL function may also be used to solve for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the solution vector for that set of equations.

LA_CHOLSOL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	spotrs
Double	dpotrs
Complex	cpotrs
Double complex	zpotrs

Table 3-51: LAPACK Routine Basis for LA_CHOLSOL

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_CHOLSOL(*A*, *B* [, /DOUBLE] [, /UPPER])

Return Value

The result is an n -element vector or k -by- n array.

Arguments

A

The n -by- n Cholesky factorization of an array, created by the LA_CHOLDC procedure.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

Keywords**DOUBLE**

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

UPPER

Set this keyword if A contains the upper triangular array, rather than the lower triangular array.

Note

If the `UPPER` keyword is set in the `LA_CHOLDC` then the `UPPER` keyword must also be set in `LA_CHOLSOL`.

Examples

Given the following system of equations:

$$6u + 15v + 55w = 9.5$$

$$15u + 55v + 225w = 50$$

$$55u + 225v + 979w = 237$$

The solution can be derived by using the following program:

```
PRO ExLA_CHOLSOL
; Define the coefficient array:
a = [[6.0, 15.0, 55.0], $
     [15.0, 55.0, 225.0], $
     [55.0, 225.0, 979.0]]

; Define the right-hand side vector b:
b = [9.5, 50.0, 237.0]

; Compute the Cholesky decomposition of a:
achol = a      ; make a copy
```

```
LA_CHOLDC, achol

; Compute and print the solution:
x = LA_CHOLSOL(achol, b)
PRINT, 'LA_CHOLSOL solution:', x
END
```

When this program is compiled and run, IDL prints:

```
LA_CHOLSOL Solution:
-0.499999      -1.00000      0.500000
```

The exact solution vector is [-0.5, -1.0, 0.5].

Version History

Introduced 5.6

See Also

[CHOLSOL](#), [LA_CHOLDC](#), [LA_CHOLMPROVE](#)

LA_DETERM

The LA_DETERM function uses LU decomposition to compute the determinant of a square array.

This routine is written in the IDL language. Its source code can be found in the file `la_determ.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LA_DETERM( A [, /CHECK] [, /DOUBLE] [, ZERO=value] )
```

Return Value

The result is a scalar of the same type as the input array.

Arguments

A

An n -by- n real or complex array.

Keywords

CHECK

Set this keyword to check A for any singularities. The determinant of a singular array is returned as zero if this keyword is set. Run-time errors may result if A is singular and this keyword is not set.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

ZERO

Use this keyword to set the absolute value of the floating-point zero. A floating-point zero on the main diagonal of a triangular array results in a zero determinant. For single-precision inputs, the default value is 1.0×10^{-6} . For double-precision inputs,

the default value is 1.0×10^{-12} . Setting this keyword to a value less than the default may improve the precision of the result.

Examples

The following program computes the determinant of a square array:

```
PRO ExLA_DETERM
; Create a square array.
array =[[1d, 2, 1], $
        [4, 10, 15], $
        [3, 7, 1]]
; Compute the determinant.
adeterm = LA_DETERM(array)
PRINT, 'LA_DETERM:', adeterm
END
```

When this program is compiled and run, IDL prints:

```
A_DETERM:
-15.000000
```

Version History

Introduced 5.6

See Also

[DETERM](#), [LA_LUDC](#)

LA_EIGENPROBLEM

The LA_EIGENPROBLEM function uses the QR algorithm to compute all eigenvalues λ and eigenvectors $v \neq 0$ of an n -by- n real nonsymmetric or complex non-Hermitian array A , for the eigenproblem $Av = \lambda v$. The routine can also compute the left eigenvectors $u \neq 0$, which satisfy $u^H A = \lambda u^H$.

LA_EIGENPROBLEM may also be used for the generalized eigenproblem:

$$Av = \lambda Bv \quad \text{and} \quad u^H A = \lambda u^H B$$

where A and B are square arrays, v are the right eigenvectors, and u are the left eigenvectors.

LA_EIGENPROBLEM is based on the following LAPACK routines:

Output Type	Standard LAPACK Routine	Generalized LAPACK Routine
Float	sggevx	sggevx
Double	dgeevx	dggevx
Complex	cgeevx	cggevx
Double complex	zgeevx	zggevx

Table 3-52: LAPACK Routine Basis for LA_EIGENPROBLEM

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_EIGENPROBLEM( A [, B] [, ALPHA=variable] [, BALANCE=value]
  [, BETA=variable] [, /DOUBLE] [, EIGENVECTORS=variable]
  [, LEFT_EIGENVECTORS=variable] [, NORM_BALANCE = variable]
  [, PERMUTE_RESULT=variable] [, SCALE_RESULT=variable]
  [, RCOND_VALUE=variable] [, RCOND_VECTOR=variable]
  [, STATUS=variable] )
```

Return Value

The result is a complex n -element vector containing the eigenvalues.

Arguments

A

The real or complex array for which to compute eigenvalues and eigenvectors.

B

An optional real or complex n -by- n array used for the generalized eigenproblem. The elements of B are converted to the same type as A before computation.

Keywords

ALPHA

For the generalized eigenproblem with the B argument, set this keyword to a named variable in which the numerator of the eigenvalues will be returned as a complex n -element vector. For the standard eigenproblem this keyword is ignored.

Tip

The ALPHA and BETA values are useful for eigenvalues which underflow or overflow. In this case the eigenvalue problem may be rewritten as $\alpha Av = \beta Bv$.

BALANCE

Set this keyword to one of the following values:

- BALANCE = 0: No balancing is applied to A .
- BALANCE = 1: Both permutation and scale balancing are performed.
- BALANCE = 2: Permutations are performed to make the array more nearly upper triangular.
- BALANCE = 3: Diagonally scale the array to make the columns and rows more equal in norm.

The default is BALANCE = 1, which performs both permutation and scaling balances. Balancing a nonsymmetric (or non-Hermitian) array is recommended to reduce the sensitivity of eigenvalues to rounding errors.

BETA

For the generalized eigenproblem with the B argument, set this keyword to a named variable in which the denominator of the eigenvalues will be returned as a real or complex n -element vector. For the standard eigenproblem this keyword is ignored.

Tip

The ALPHA and BETA values are useful for eigenvalues which underflow or overflow. In this case, the eigenvalue problem may be rewritten as $\alpha Av = \beta Bv$.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

EIGENVECTORS

Set this keyword to a named variable in which the eigenvectors will be returned as a set of row vectors. If this variable is omitted then eigenvectors will not be computed unless the `RCOND_VALUE` or `RCOND_VECTOR` keywords are present.

Note

For the standard eigenproblem the eigenvectors are normalized and rotated to have norm 1 and largest component real. For the generalized eigenproblem the eigenvectors are normalized so that the largest component has $\text{abs}(\text{real}) + \text{abs}(\text{imaginary}) = 1$.

LEFT_EIGENVECTORS

Set this keyword to a named variable in which the left eigenvectors will be returned as a set of row vectors. If this variable is omitted then left eigenvectors will not be computed unless the `RCOND_VALUE` or `RCOND_VECTOR` keywords are present.

Note

Note - For the standard eigenproblem the eigenvectors are normalized and rotated to have norm 1 and largest component real. For the generalized eigenproblem the eigenvectors are normalized so that the largest component has $\text{abs}(\text{real}) + \text{abs}(\text{imaginary}) = 1$.

NORM_BALANCE

Set this keyword to a named variable in which the one-norm of the balanced matrix will be returned. The one-norm is defined as the maximum value of the sum of absolute values of the columns. For the standard eigenproblem, this will be returned as a scalar value; for the generalized eigenproblem this will be returned as a two-element vector containing the A and B norms.

PERMUTE_RESULT

Set this keyword to a named variable in which the result for permutation balancing will be returned as a two-element vector $[ilo, ihi]$. If permute balancing is not done then the values will be $ilo = 1$ and $ihi = n$.

RCOND_VALUE

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvalues will be returned as an n -element vector. If RCOND_VALUE is present then left and right eigenvectors must be computed.

RCOND_VECTOR

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvectors will be returned as an n -element vector. If RCOND_VECTOR is present then left and right eigenvectors must be computed.

SCALE_RESULT

Set this keyword to a named variable in which the results for permute and scale balancing will be returned. For the standard eigenproblem, this will be returned as an n -element vector. For the generalized eigenproblem, this will be returned as a n -by-2 array with the first row containing the permute and scale factors for the left side of A and B and the second row containing the factors for the right side of A and B .

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: The QR algorithm failed to compute all eigenvalues; no eigenvectors or condition numbers were computed. The STATUS value indicates that eigenvalues $ilo:STATUS$ (starting at index 1) did not converge; all other eigenvalues converged.

Note

If STATUS is not specified, any error messages will be output to the screen.

Examples

Find the eigenvalues and eigenvectors for an array using the following program:

```

PRO ExLA_EIGENPROBLEM
; Create a random array:
n = 4
seed = 12321
array = RANDOMN(seed, n, n)

; Compute all eigenvalues and eigenvectors:
eigenvalues = LA_EIGENPROBLEM(array, $
    EIGENVECTORS = eigenvectors)
PRINT, 'LA_EIGENPROBLEM Eigenvalues:'
PRINT, eigenvalues

; Check the results using the eigenvalue equation:
maxErr = 0d
FOR i = 0, n - 1 DO BEGIN
    ; A*z = lambda*z
    alhs = array ## eigenvectors[* , i]
    arhs = eigenvalues[i]*eigenvectors[* , i]
    maxErr = maxErr > MAX(ABS(alhs - arhs))
ENDFOR
PRINT, 'LA_EIGENPROBLEM Error:', maxErr

; Now try the generalized eigenproblem:
b = IDENTITY(n) + 0.01*RANDOMN(seed, n, n)
eigenvalues = LA_EIGENPROBLEM(Array, B)
PRINT, 'LA_EIGENPROBLEM Generalized Eigenvalues:'
PRINT, EIGENVALUES
END

```

When this program is compiled and run, IDL prints:

```

LA_EIGENPROBLEM eigenvalues:
(   -0.593459,    0.566318)(   -0.593459,    -0.566318)
(    1.06216,    0.00000)(    1.61286,    0.00000)
LA_EIGENPROBLEM error:  4.0978193e-07
LA_EIGENPROBLEM generalized eigenvalues:
(   -0.574766,    0.567452)(   -0.574766,    -0.567452)
(    1.57980,    0.00000)(    1.08711,    0.00000)

```

Version History

Introduced 5.6

See Also

[LA_EIGENVEC](#), [LA_ELMHES](#), [LA_HQR](#)

LA_EIGENQL

The LA_EIGENQL function computes selected eigenvalues λ and eigenvectors $z \neq 0$ of an n -by- n real symmetric or complex Hermitian array A , for the eigenproblem $Az = \lambda z$.

LA_EIGENQL may also be used for the generalized symmetric eigenproblems:

$$Az = \lambda Bz \text{ or } ABz = \lambda z \text{ or } BAz = \lambda z$$

where A and B are symmetric (or Hermitian) and B is positive definite.

LA_EIGENQL is based on the following LAPACK routines:

Output Type	Standard Eigenproblem	Generalized
Float	ssyevx, ssyevr, ssyevd	ssygvx, ssygvd
Double	dsyevx, dsyevr, dsyevd	dsygvx, dsygvd
Complex	cheevx, cheevr, cheevd	chegvx, chegvd
Double complex	zheevx, zheevr, zheevd	zhegvx, zhegvd

Table 3-53: LAPACK Routine Basis for LA_EIGENQL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_EIGENQL( A [, B] [, /DOUBLE] [, EIGENVECTORS=variable]
  [, FAILED=variable] [, GENERALIZED=value] [, METHOD=value]
  [, RANGE=vector] [, SEARCH_RANGE=vector] [, STATUS=variable]
  [, TOLERANCE=value] )
```

Return Value

The result is a real vector containing the eigenvalues in ascending order.

Arguments

A

The real or complex n -by- n array for which to compute eigenvalues and eigenvectors. A must be symmetric (or Hermitian).

B

An optional real or complex n -by- n array used for the generalized eigenproblem. B must be symmetric (or Hermitian) and positive definite. The elements of B are converted to the same type as A before computation.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

EIGENVECTORS

Set this keyword to a named variable in which the eigenvectors will be returned as a set of row vectors. If this variable is omitted then eigenvectors will not be computed. All eigenvectors will be returned unless the `RANGE` or `SEARCH_RANGE` keywords are used to restrict the eigenvalue range.

FAILED

Set this keyword to a named variable in which to return the indices of eigenvectors that did not converge. This keyword is only available for `METHOD = 0`, and will be ignored for other methods.

Note

Index numbers within `FAILED` start at 1.

GENERALIZED

For the generalized eigenproblem with the optional B argument, set this keyword to indicate which problem to solve. Possible values are:

- `GENERALIZED = 0` (the default): Solve $Az = \lambda Bz$.

- GENERALIZED = 1: Solve $ABz = \lambda z$.
- GENERALIZED = 2: Solve $BAz = \lambda z$.

This keyword is ignored if argument B is not present.

METHOD

Set this keyword to indicate which computation method to use. Possible values are:

- METHOD = 0 (the default): Use tridiagonal decomposition to compute some or all of the eigenvalues and (optionally) eigenvectors.
- METHOD = 1: Use the Relatively Robust Representation (RRR) algorithm to compute some or all of the eigenvalues and (optionally) eigenvectors. This method is unavailable for the generalized eigenproblem with the optional B argument, and will default to METHOD = 0.

Note

The RRR method may produce *NaN* and *Infinity* floating-point exception messages during normal execution.

- METHOD = 2: Use a divide-and-conquer algorithm to compute all of the eigenvalues and (optionally) all eigenvectors. This method is available for either the standard or generalized eigenproblems. For METHOD = 2 the RANGE, SEARCH_RANGE, and TOLERANCE keywords are ignored, and all eigenvalues are returned.

RANGE

Set this keyword to a two-element vector containing the indices of the smallest and largest eigenvalues to be returned. The default is $[0, n-1]$, which returns all eigenvalues and eigenvectors. This keyword is ignored for METHOD = 2.

SEARCH_RANGE

Set this keyword to a two-element floating-point vector containing the lower and upper bounds of the interval to be searched for eigenvalues. The default is to return all eigenvalues and eigenvectors. This keyword is ignored for METHOD = 2. If both RANGE and SEARCH_RANGE are specified, only the SEARCH_RANGE values are used.

Note

If the search range does not contain any eigenvalues, then Result, EIGENVECTORS, and FAILED will each be set to a scalar zero.

STATUS

Set this keyword to a named variable that will contain the status of the computation. In all cases `STATUS = 0` indicates successful computation. For the standard eigenproblem, possible nonzero values are:

- `METHOD = 0, STATUS > 0`: `STATUS` eigenvectors failed to converge. The `FAILED` keyword contains the indices of the eigenvectors that did not converge.
- `METHOD = 1, STATUS < 0` or `STATUS > 0`: An internal error occurred during the computation.
- `METHOD = 2, STATUS > 0`: `STATUS` off-diagonal elements of an intermediate tridiagonal matrix did not converge to zero.

For the generalized eigenproblem, possible nonzero values are:

- `METHOD = 0, 0 < STATUS ≤ n`: `STATUS` eigenvectors failed to converge. The `FAILED` keyword contains the indices of the eigenvectors that did not converge.
- `METHOD = 0, STATUS > n`: The factorization of B could not be completed and the computation failed. The value of $(STATUS - n)$ specifies the order of the leading minor of B which is not positive definite.
- `METHOD = 2, 0 < STATUS ≤ n`: `STATUS` off-diagonal elements of an intermediate tridiagonal matrix did not converge to zero.
- `METHOD = 2, STATUS > n`: The factorization of B could not be completed and the computation failed. The value of $(STATUS - n)$ specifies the order of the leading minor of B which is not positive definite.

Note

If `STATUS` is not specified, any error messages will be output to the screen.

TOLERANCE

Set this keyword to a scalar giving the absolute error tolerance for the eigenvalues and eigenvectors. For the most accurate eigenvalues, `TOLERANCE` should be set to $2 * XMIN$, where $XMIN$ is the magnitude of the smallest usable floating-point value. For `METHOD = 0`, if `TOLERANCE` is less than or equal to zero, or is unspecified, then a tolerance value of $EPS * \|T\|_1$ will be used, where T is the tridiagonal matrix obtained from A . For `METHOD = 1`, if `TOLERANCE` is less than or equal to $N * EPS * \|T\|_1$, or is unspecified, then a tolerance value of $N * EPS * \|T\|_1$ will be used.

For values of *EPS* and *XMIN*, see the [MACHAR](#). This keyword is ignored for *METHOD* = 2.

Tip

If the *LA_EIGENQL* routine fails to converge, try setting the *TOLERANCE* to a larger value.

Examples

Find the eigenvalues and eigenvectors for a symmetric array using the following program:

```

PRO ExLA_EIGENQL
; Create a random symmetric array:
n = 10
seed = 12321
array = RANDOMN(seed, n, n)
array = array + TRANSPOSE(array)

; Compute all eigenvalues and eigenvectors:
eigenvalues = LA_EIGENQL(array, $
    EIGENVECTORS=eigenvectors)

; Check the results using the eigenvalue equation:
maxErr = 0d
FOR i=0,n-1 DO BEGIN
    ; a*z = lambda*z
    alhs = array ## eigenvectors[* ,i]
    arhs = eigenvalues[i]*eigenvectors[* ,i]
    maxErr = maxErr > MAX(ABS(alhs - arhs))
ENDFOR
PRINT, 'LA_EIGENQL error:', maxErr

; Compute the three largest eigenvalues:
eigenvalues = LA_EIGENQL(array, $
    EIGENVECTORS = eigenvectors, $
    RANGE = [n-3,n-1])
PRINT, 'LA_EIGENQL eigenvalues:', eigenvalues

; Now try the generalized eigenproblem:
b = IDENTITY(n) + 0.01*RANDOMN(seed,n,n)
; Make B symmetric and positive definite:
b = b ## TRANSPOSE(b)

; Compute the three largest generalized eigenvalues:
eigenvalues = LA_EIGENQL(array, b, RANGE=[n-3,n-1])
PRINT, 'LA_EIGENQL Generalized Eigenvalues:'

```

```
PRINT, Eigenvalues  
END
```

When this program is compiled and run, IDL prints:

```
LA_EIGENQL error: 1.3560057e-06  
LA_EIGENQL eigenvalues: 3.82993 4.69785 5.61567  
LA_EIGENQL generalized eigenvalues:  
3.83750 4.74803 5.57692
```

Version History

Introduced 5.6

See Also

[EIGENQL](#), [LA_TRIQL](#), [LA_TRIRED](#)

LA_EIGENVEC

The LA_EIGENVEC function uses the QR algorithm to compute all or some of the eigenvectors $v \neq 0$ of an n -by- n real nonsymmetric or complex non-Hermitian array A , for the eigenproblem $Av = \lambda v$. The routine can also compute the left eigenvectors $u \neq 0$, which satisfy $u^H A = \lambda u^H$.

Note

The left and right eigenvectors returned by LA_EIGENVEC are normalized to norm 1. Unlike the LA_EIGENPROBLEM, they are not rotated to have largest component real. Therefore, you may notice slight differences in results between LA_EIGENVEC and LA_EIGENPROBLEM.

LA_EIGENVEC is based on the following LAPACK routines:

Output Type	Eigenvectors	Condition Numbers	Undo Balancing
Float	strevc	strsna	sgebak
Double	dtrevc	dtrsna	dgebak
Complex	ctrevc	ctrsna	cgebak
Double complex	ztrevc	ztrsna	zgebak

Table 3-54: LAPACK Routine Basis for LA_EIGENVEC

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_EIGENVEC( T, QZ [, BALANCE=value] [, /DOUBLE]
  [, EIGENINDEX=variable] [, LEFT_EIGENVECTORS=variable]
  [, PERMUTE_RESULT=[ilo, ihi]] [, SCALE_RESULT=vector]
  [, RCOND_VALUE=variable] [, RCOND_VECTOR=variable]
  [, SELECT=vector])
```

Return Value

The result is a complex array containing the eigenvectors as a set of row vectors.

Arguments

T

The upper quasi-triangular array containing the Schur form, created by LA_HQR.

QZ

The array of Schur vectors, created by LA_HQR.

Keywords

BALANCE

If balancing was applied in the call to LA_ELMHES, then set this keyword to the same value that was used, in order to apply the backward balancing transform to the eigenvectors. If BALANCE is not specified, then the default is BALANCE = 1.

Note

If BALANCE is not zero, then both PERMUTE_RESULT and SCALE_RESULT must be supplied.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *T* is double precision, otherwise the default is DOUBLE = 0.

EIGENINDEX

If keyword SELECT is used, then set this keyword to a named variable in which the indices of the eigenvalues that correspond to the selected eigenvectors will be returned. If the SELECT keyword is not used then EIGENINDEX will be set to LINDGEN(*n*).

Tip

This keyword is most useful for real input arrays when the SELECT keyword is present. In this case, a value of SELECT[*j*] equal to 1 may produce two eigenvectors if the eigenvalue is part of a complex-conjugate pair.

LEFT_EIGENVECTORS

Set this keyword to a named variable in which the left eigenvectors will be returned as a set of row vectors. If this variable is omitted then left eigenvectors will not be computed unless the `RCOND_VALUE` keyword is present.

PERMUTE_RESULT

Set this keyword to a two-element vector containing the $[ilo, ihi]$ permutation results from the `LA_ELMHES` procedure. This keyword must be present if `BALANCE = 1` or `BALANCE = 2`.

RCOND_VALUE

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvalues will be returned as an n -element vector. If `RCOND_VALUE` is present then left and right eigenvectors must be computed.

RCOND_VECTOR

Set this keyword to a named variable in which the reciprocal condition numbers for the eigenvectors will be returned as an n -element vector.

SCALE_RESULT

Set this keyword to an n -element vector containing the permute and scale balancing results from the `LA_ELMHES` procedure. This keyword must be present if `BALANCE` is not zero.

SELECT

Set this keyword to an n -element vector of zeroes or ones that indicates which eigenvectors to compute. There are two cases:

- The original array was real: If the j -th eigenvalue (as created by `LA_HQR`) is real, then if `SELECT[j]` is set to 1, then the j -th eigenvector will be computed. If the j -th and $(j+1)$ eigenvalues form a complex-conjugate pair, then if either `SELECT[j]` or `SELECT[j+1]` is set to 1, then the complex-conjugate pair of j -th and $(j+1)$ eigenvectors will be computed.
- The original array was complex: If `SELECT[j]` is set to 1, then the j -th eigenvector will be computed.

If `SELECT` is omitted then all eigenvectors are returned.

Examples

Compute the eigenvalues and selected eigenvectors of a random array using the following program:

```

PRO ExLA_EIGENVEC
; Create a random array:
n = 10
seed = 12321
array = RANDOMN(seed, n, n)

; Reduce to upper Hessenberg and compute Q:
H = LA_ELMHES(array, q, $
PERMUTE_RESULT = permute, SCALE_RESULT = scale)

; Compute eigenvalues, T, and QZ arrays:
eigenvalues = LA_HQR(h, q, PERMUTE_RESULT = permute)

; Compute eigenvectors corresponding to
; the first 3 eigenvalues.
select = [1, 1, 1, REPLICATE(0, n - 3)]
eigenvectors = LA_EIGENVEC(H, Q, $
EIGENINDEX = eigenindex, $
PERMUTE_RESULT = permute, SCALE_RESULT = scale, $
SELECT = select)

PRINT, 'LA_EIGENVEC eigenvalues:'
PRINT, eigenvalues[eigenindex]
END

```

When this program is compiled and run, IDL prints:

```

LA_EIGENVEC eigenvalues:
(-0.278633, 2.55055) (-0.278633, -2.55055)
(2.31208, 0.000000)

```

Version History

Introduced 5.6

See Also

[EIGENVEC](#), [LA_ELMHES](#), [LA_HQR](#)

LA_ELMHES

The LA_ELMHES function reduces a real nonsymmetric or complex non-Hermitian array to upper Hessenberg form H . If the array is real then the decomposition is $A = Q H Q^T$, where Q is orthogonal. If the array is complex Hermitian then the decomposition is $A = Q H Q^H$, where Q is unitary. The superscript T represents the transpose while superscript H represents the Hermitian, or transpose complex conjugate.

LA_ELMHES is based on the following LAPACK routines:

Output Type	Balance & Reduce	Norm	Optional Q
Float	sgebal, sgehrd	slange	sorghr
Double	dgebal, dgehrd	dlange	dorghr
Complex	cgebal, cgehrd	clange	cunghr
Double complex	zgebal, zgehrd	zlange	zunghr

Table 3-55: LAPACK Routine Basis for LA_ELMHES

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_ELMHES( Array [, Q] [, BALANCE=value] [, /DOUBLE]
  [, NORM_BALANCE=variable] [, PERMUTE_RESULT=variable]
  [, SCALE_RESULT=variable] )
```

Return Value

The result is an array of the same type as A containing the upper Hessenberg form. The Hessenberg array is stored in the upper triangle and the first subdiagonal. Elements below the subdiagonal should be ignored but are not automatically set to zero.

Arguments

Array

The n -by- n real or complex array to reduce to upper Hessenberg form.

Q

Set this optional argument to a named variable in which the array Q will be returned. The Q argument may then be input into LA_HQR to compute the Schur vectors.

Keywords

BALANCE

Set this keyword to one of the following values:

- BALANCE = 0: No balancing is applied to *Array*.
- BALANCE = 1: Both permutation and scale balancing are performed.
- BALANCE = 2: Permutations are performed to make the array more nearly upper triangular.
- BALANCE = 3: Diagonally scale the array to make the columns and rows more equal in norm.

The default is BALANCE = 1, which performs both permutation and scaling balances. Balancing a nonsymmetric array is recommended to reduce the sensitivity of eigenvalues to rounding errors.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

NORM_BALANCE

Set this keyword to a named variable in which the one-norm of the balanced matrix will be returned. The one-norm is defined as the maximum value of the sum of absolute values of the columns.

PERMUTE_RESULT

Set this keyword to a named variable in which the result for permutation balancing will be returned as a two-element vector $[ilo, ihi]$. If permute balancing is not done then the values will be $ilo = 1$ and $ihi = n$.

SCALE_RESULT

Set this keyword to a named variable in which the result for permute and scale balancing will be returned as an n -element vector.

Examples

See `LA_EIGENVEC` for an example of using this procedure.

Version History

Introduced 5.6

See Also

[ELMHES](#), [LA_HQR](#)

LA_GM_LINEAR_MODEL

The `LA_GM_LINEAR_MODEL` function is used to solve a general Gauss-Markov linear model problem:

$$\text{minimize}_x \|y\|_2 \text{ with constraint } d = Ax + By$$

where A is an m -column by n -row array, B is a p -column by n -row array, and d is an n -element input vector with $m \leq n \leq m+p$.

The following items should be noted:

- If A has full column rank m and the array $(A \ B)$ has full row rank n , then there is a unique solution x and a minimal 2-norm solution y .
- If B is square and nonsingular then the problem is equivalent to a weighted linear least-squares problem, $\text{minimize}_x \|B^{-1}(Ax - d)\|_2$.
- If B is the identity matrix then the problem reduces to the ordinary linear least-squares problem, $\text{minimize}_x \|Ax - d\|_2$.

`LA_GM_LINEAR_MODEL` is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sggglm
Double	dggglm
Complex	cggglm
Double complex	zggglm

Table 3-56: LAPACK Routine Basis for `LA_GM_LINEAR_MODEL`

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

`Result = LA_GM_LINEAR_MODEL(A, B, D, Y [, /DOUBLE])`

Return Value

The result (x) is an m -element vector whose type is identical to A .

Arguments

A

The m -by- n array used in the constraint equation.

B

The p -by- n array used in the constraint equation.

D

An n -element input vector used in the constraint equation.

Y

Set this argument to a named variable, which will contain the p -element output vector.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

Examples

Given the constraint equation $d = Ax + By$, (where A , B , and d are defined in the program below) the following example program solves the general Gauss-Markov problem:

```

PRO ExLA_GM_LINEAR_MODEL
; Define some example coefficient arrays:
a = [[2, 7, 4], $
      [5, 1, 3], $
      [3, 3, 6], $
      [4, 5, 2]]
b = [[-3, 2], $
      [1, 5], $
      [2, 9], $
      [4, 1]]

```

```
; Define a sample left-hand side vector D:
d = [-1, 2, -3, 4]

; Find and print the solution x:
x = LA_GM_LINEAR_MODEL(a, b, d, y)
PRINT, 'LA_GM_LINEAR_MODEL solution:'
PRINT, X
PRINT, 'LA_GM_LINEAR_MODEL 2-norm solution:'
PRINT, Y
END
```

When this program is compiled and run, IDL prints:

```
LA_GM_LINEAR_MODEL solution:
      1.04668      0.350346      -1.28445
LA_GM_LINEAR_MODEL 2-norm solution:
      0.151716      0.0235733
```

Version History

Introduced 5.6

See Also

[LA_LEAST_SQUARE_EQUALITY](#), [LA_LEAST_SQUARES](#)

LA_HQR

The LA_HQR function uses the multishift QR algorithm to compute all eigenvalues of an n -by- n upper Hessenberg array. The LA_ELMHES routine can be used to reduce a real or complex array to upper Hessenberg form suitable for input to this procedure. LA_HQR may also be used to compute the matrices T and QZ from the Schur decomposition $A = (QZ) T (QZ)^H$.

LA_HQR is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	shseqr
Double	dhseqr
Complex	chseqr
Double complex	zhseqr

Table 3-57: LAPACK Routine Basis for LA_HQR

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_HQR(H [, Q] [, /DOUBLE] [, PERMUTE_RESULT=[*ilo*, *ihi*]] [, STATUS=*variable*])

Return Value

The result is an n -element complex vector.

Arguments

H

An n -by- n upper Hessenberg array, created by the LA_ELMHES procedure. If argument Q is present, then on return H is replaced by the Schur form T . If argument Q is not present then H is unchanged.

Q

Set this optional argument to the array Q created by the LA_ELMHES procedure. If argument Q is present, then on return Q is replaced by the Schur vectors QZ .

Keywords**DOUBLE**

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if H is double precision, otherwise the default is `DOUBLE = 0`.

PERMUTE_RESULT

Set this keyword to a two-element vector containing the $[ilo, ihi]$ permutation results from the LA_ELMHES procedure. The default is $[1, n]$, indicating that permute balancing was not done on H .

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: The algorithm failed to find all eigenvalues in $30*(ihi - ilo + 1)$ iterations. The `STATUS` value indicates that eigenvalues $ilo:STATUS$ (starting at index l) did not converge; all other eigenvalues converged.

Note

If `STATUS` is not specified, any error messages will output to the screen.

Examples

See LA_EIGENVEC for an example of using this procedure.

Version History

Introduced 5.6

See Also

[HQR](#), [LA_EIGENVEC](#), [LA_ELMHES](#)

LA_INVERT

The LA_INVERT function uses LU decomposition to compute the inverse of a square array.

LA_INVERT is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgetrf, sgetri
Double	dgetrf, dgetri
Complex	cgetrf, cgetri
Double complex	zgetrf, zgetri

Table 3-58: LAPACK Routine Basis for LA_INVERT

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_INVERT(*A* [, /DOUBLE] [, STATUS=*variable*])

Return Value

The result is an array of the same dimensions as the input array.

Arguments

A

The *n*-by-*n* array to be inverted.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *A* is double precision, otherwise the default is DOUBLE = 0.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: The array is singular and the inverse could not be computed. The `STATUS` value specifies which value along the diagonal (starting at one) is zero.

Note

If `STATUS` is not specified, any error messages will be output to the screen.

Examples

The following program computes the inverse of a square array:

```
PRO ExLA_INVERT
; Create a square array.
array =[[1d, 2, 1], $
        [4, 10, 15], $
        [3, 7, 1]]
; Compute the inverse and check the error.
ainv = LA_INVERT(array)
PRINT, 'LA_INVERT Identity Matrix:'
PRINT, ainv ## array
END
```

When this program is compiled and run, IDL prints:

```
A_INVERT Identity Matrix:
1.0000000    1.7763568e-015    6.6613381e-016
0.00000000    1.0000000    1.2212453e-015
0.00000000    0.00000000    1.0000000
```

Version History

Introduced 5.6

See Also

[INVERT](#), [LA_LUDC](#)

LA_LEAST_SQUARE_EQUALITY

The LA_LEAST_SQUARE_EQUALITY function is used to solve the linear least-squares problem:

$$\text{Minimize}_x \|Ax - c\|_2 \text{ with constraint } Bx = d$$

where A is an n -column by m -row array, B is an n -column by p -row array, c is an m -element input vector, and d is a p -element input vector with $p \leq n \leq m+p$. If B has full row rank p and the array $\begin{pmatrix} A \\ B \end{pmatrix}$ has full column rank n , then a unique solution exists.

LA_LEAST_SQUARE_EQUALITY is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sggls
Double	dggls
Complex	cggls
Double complex	zggls

Table 3-59: LAPACK Routine Basis for LA_LEAST_SQUARE_EQUALITY

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_LEAST_SQUARE_EQUALITY(*A*, *B*, *C*, *D* [, /DOUBLE]
[, RESIDUAL=*variable*])

Return Value

The result (x) is an n -element vector.

Arguments

A

The n -by- m array used in the least-squares minimization.

B

The n -by- p array used in the equality constraint.

C

An m -element input vector containing the right-hand side of the least-squares system.

D

A p -element input vector containing the right-hand side of the equality constraint.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

RESIDUAL

Set this keyword to a named variable in which to return a scalar giving the residual sum-of-squares for `Result`. If $n = m + p$ then `RESIDUAL` will be zero.

Examples

Given the following system of equations:

$$\begin{aligned} 2t + 5u + 3v + 4w &= 9 \\ 7t + u + 3v + 5w &= 1 \\ 4t + 3u + 6v + 2w &= 2 \end{aligned}$$

with constraints,

$$\begin{aligned} -3t + u + 2v + 4w &= -4 \\ 2t + 5u + 9v + 1w &= 4 \end{aligned}$$

find the solution using the following program:

```
PRO ExLA_LEAST_SQUARE_EQUALITY
; Define the coefficient array:
a = [[2, 5, 3, 4], $
      [7, 1, 3, 5], $
      [4, 3, 6, 2]]
```

```
; Define the constraint array:
b = [[-3, 1, 2, 4], $
      [2, 5, 9, 1]]

; Define the right-hand side vector c:
c = [9, 1, 2]

; Define the constraint right-hand side d:
d = [-4, 4]

; Find and print the minimum norm solution of a:
x = LA_LEAST_SQUARE_EQUALITY(a, b, c, d)
PRINT, 'LA_LEAST_SQUARE_EQUALITY solution:'
PRINT, x
END
```

When this program is compiled and run, IDL prints:

```
LA_LEAST_SQUARE_EQUALITY solution:
      0.651349      2.72695      -1.14638      -0.620036
```

Version History

Introduced 5.6

See Also

[LA_GM_LINEAR_MODEL](#), [LA_LEAST_SQUARES](#)

LA_LEAST_SQUARES

The LA_LEAST_SQUARES function is used to solve the linear least-squares problem:

$$\text{Minimize}_x \|Ax - b\|_2$$

where A is a (possibly rank-deficient) n -column by m -row array, b is an m -element input vector, and x is the n -element solution vector. There are three possible cases:

- If $m \geq n$ and the rank of A is n , then the system is overdetermined and a unique solution may be found, known as the least-squares solution.
- If $m < n$ and the rank of A is m , then the system is under determined and an infinite number of solutions satisfy $Ax - b = 0$. In this case, the solution is found which minimizes $\|x\|_2$, known as the minimum norm solution.
- If A is rank deficient, such that the rank of A is less than $\text{MIN}(m, n)$, then the solution is found which minimizes both $\|Ax - b\|_2$ and $\|x\|_2$, known as the minimum-norm least-squares solution.

The LA_LEAST_SQUARES function may also be used to solve for multiple systems of least squares, with each column of b representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the solution vector for that set of equations.

LA_LEAST_SQUARES is based on the following LAPACK routines:

Output Type	LAPACK Routines
Float	sgels, sgelsy, sgelss, sgelsd
Double	dgels, dgelsy, dgelss, dgelsd
Complex	cgels, cgelsy, cgelss, cgelsd
Double complex	zgels, zgelsy, zgelss, zgelsd

Table 3-60: LAPACK Routine Basis for LA_LEAST_SQUARES

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_LEAST_SQUARES( A, B [, /DOUBLE] [, METHOD=value]
  [, RANK=variable] [, RCONDITION=value] [, RESIDUAL=variable]
  [, STATUS=variable] )
```

Return Value

The result is an n -element vector or k -by- n array.

Arguments

A

The n -by- m array used in the least-squares system.

B

An m -element input vector containing the right-hand side of the linear least-squares system, or a k -by- m array, where each of the k columns represents a different least-squares system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

METHOD

Set this keyword to indicate which computation method to use. Possible values are:

- `METHOD = 0` (the default): Assume that array A has full rank equal to $\min(m, n)$. If $m \geq n$, find the least-squares solution to the overdetermined system. If $m < n$, find the minimum norm solution to the under determined system. Both cases use QR or LQ factorization of A .
- `METHOD = 1`: Assume that array A may be rank deficient; use a complete orthogonal factorization of A to find the minimum norm least-squares solution.

- **METHOD = 2:** Assume that array A may be rank deficient; use singular value decomposition (SVD) to find the minimum norm least-squares solution.
- **METHOD = 3:** Assume that array A may be rank deficient; use SVD with a divide-and-conquer algorithm to find the minimum norm least-squares solution. The divide-and-conquer method is faster than regular SVD, but may require more memory.

RANK

Set this keyword to a named variable in which to return the effective rank of A . If **METHOD = 0** or the array is full rank, then **RANK** will have the value $\text{MIN}(m, n)$.

RCONDITION

Set this keyword to the reciprocal condition number used as a cutoff value in determining the effective rank of A . Arrays with condition numbers larger than $1/\text{RCONDITION}$ are assumed to be rank deficient. If **RCONDITION** is set to zero or omitted, then array A is assumed to be of full rank. This keyword is ignored for **METHOD = 0**.

RESIDUAL

If $m > n$ and the rank of A is n (the system is overdetermined), then set this keyword to a named variable in which to return the residual sum-of-squares for *Result*. If B is an m -element vector then **RESIDUAL** will be a scalar; if B is a k -by- m array then **RESIDUAL** will be a k -element vector containing the residual sum-of-squares for each system of equations. If $m \leq n$ or A is rank deficient ($\text{rank} < n$) then the values in **RESIDUAL** will be zero.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- **STATUS = 0:** The computation was successful.
- **STATUS > 0:** For **METHOD=2** or **METHOD=3**, this indicates that the SVD algorithm failed to converge, and **STATUS** off-diagonal elements of an intermediate bidiagonal form did not converge to zero. For **METHOD=0** or **METHOD=1** the **STATUS** will always be zero.

Examples

Given the following under determined system of equations:

$$2t + 5u + 3v + 4w = 3$$

$$7t + u + 3v + 5w = 1$$

$$4t + 3u + 6v + 2w = 6$$

The following program can be used to find the solution:

```

PRO ExLA_LEAST_SQUARES
; Define the coefficient array:
a = [[2, 5, 3, 4], $
      [7, 1, 3, 5], $
      [4, 3, 6, 2]]

; Define the right-hand side vector b:
b = [3, 1, 6]

; Find and print the minimum norm solution of a:
x = LA_LEAST_SQUARES(a, b)
PRINT, 'LA_LEAST_SQUARES solution:', x
END

```

When this program is compiled and run, IDL prints:

```

LA_LEAST_SQUARES solution:
-0.0376844      0.350628      0.986164      -0.409066

```

Version History

Introduced 5.6

See Also

[LA_GM_LINEAR_MODEL](#), [LA_LEAST_SQUARE_EQUALITY](#)

LA_LINEAR_EQUATION

The LA_LINEAR_EQUATION function uses LU decomposition to solve a system of linear equations, $AX = B$, and provides optional error bounds and backward error estimates.

The LA_LINEAR_EQUATION function may also be used to solve for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the solution vector for that set of equations.

This routine is written in the IDL language. Its source code can be found in the file `la_linear_equation.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LA_LINEAR_EQUATION( Array, B [, BACKWARD_ERROR=variable]  
[, /DOUBLE] [, FORWARD_ERROR=variable] [, STATUS=variable])
```

Return Value

The result is an n -element vector or k -by- n array.

Arguments

Array

The n -by- n array of the linear system $AX = B$.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If B is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If B is an array containing k linear systems,

then `BACKWARD_ERROR` will be a k -element vector. The backward error is the smallest relative change in any element of A or B that makes X an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if `Array` is double precision, otherwise the default is `DOUBLE = 0`.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If B is a vector containing a single linear system, then `FORWARD_ERROR` will be a scalar. If B is an array containing k linear systems, then `FORWARD_ERROR` will be a k -element vector. For each linear system, if X_{true} is the true solution corresponding to X , then the forward error is an estimated upper bound for the magnitude of the largest element in $(X - X_{true})$ divided by the magnitude of the largest element in X .

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: The computation failed because one of the diagonal elements of the LU decomposition is zero. The `STATUS` value specifies which value along the diagonal (starting at one) is zero.

Note

If `STATUS` is not specified, any error messages will be output to the screen.

Examples

Given the system of equations:

$$\begin{array}{rcl} 4u + 16000v + 17000w & = & 100.1 \\ 2u + 5v + 8w & = & 0.1 \\ 3u + 6v + 10w & = & 0.01 \end{array}$$

The following program can be used to find the solution:

```
PRO ExLA_LINEAR_EQUATION
; Define the coefficient array:
a = [[4, 16000, 17000], $
      [2, 5, 8], $
      [3, 6, 10]]

; Define the right-hand side vector b:
b = [100.1, 0.1, 0.01]

; Compute and print the solution to ax=b:
x = LA_LINEAR_EQUATION(a, b)
PRINT, 'LA_LINEAR_EQUATION solution:', x
end
```

When this program is compiled and run, IDL prints:

```
LA_LINEAR_EQUATION solution:
-0.397432    -0.334865     0.321148
```

The exact solution to 6 decimal places is [-0.397432, -0.334865, 0.321149].

Version History

Introduced 5.6

See Also

[LA_LUDC](#), [LA_LUMPROVE](#), [LA_LUSOL](#)

LA_LUDC

The LA_LUDC procedure computes the LU decomposition of an n -column by m -row array as:

$$A = P L U$$

where P is a permutation matrix, L is lower trapezoidal with unit diagonal elements (lower triangular if $n = m$), and U is upper trapezoidal (upper triangular if $n = m$).

LA_LUDC is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgetrf
Double	dgetrf
Complex	cgetrf
Double complex	zgetrf

Table 3-61: LAPACK Routine Basis for LA_LUDC

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_LUDC, *Array*, *Index* [, /DOUBLE] [, STATUS=*variable*]

Arguments

Array

A named variable containing the real or complex array to decompose. This procedure returns *Array* as its LU decomposition.

Index

An output vector with $\text{MIN}(m, n)$ elements that records the row permutations which occurred as a result of partial pivoting. For $1 < j < \text{MIN}(m, n)$, row j of the matrix was interchanged with row *Index*[j].

Note

Row numbers within *Index* start at one rather than zero.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if *Array* is double precision, otherwise the default is `DOUBLE = 0`.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: One of the diagonal elements of *U* is zero. The `STATUS` value specifies which value along the diagonal (starting at one) is zero.

Note

If `STATUS` is not specified, any error messages will output to the screen.

Examples

The following example uses the LU decomposition on a given array, then determines the residual error of using the resulting lower and upper arrays to recompute the original array:

```

PRO ExLA_LUDC
; Create a random array:
n = 20
seed = 12321
array = RANDOMN(seed, n, n)

; Compute LU decomposition.
aludc = array           ; make a copy
LA_LUDC, aludc, index

; Extract the lower and upper triangular arrays.
l = IDENTITY(n)
u = FLTARR(n, n)

```

```

FOR j = 1,n - 1 DO l[0:j-1,j] = aludc[0:j-1,j]
FOR j=0,n - 1 DO u[j:*,j] = aludc[j:*,j]

; Reconstruct array, but with rows permuted.
arecon = l ## u
; Adjust from LAPACK back to IDL indexing.
Index = Index - 1
; Permute the array rows back into correct order.
; Note that we need to loop in reverse order.
FOR i = n - 1,0,-1 DO BEGIN & $
    temp = arecon[*,i]
    arecon[*, i] = arecon[*,index[i]]
    arecon[*, index[i]] = temp
ENDFOR
PRINT, 'LA_LUDC Error:', MAX(ABS(arecon - array))
END

```

When this program is compiled and run, IDL prints:

```
LA_LUDC error: 4.76837e-007
```

Version History

Introduced 5.6

See Also

[LA_LUMPROVE](#), [LA_LUSOL](#), [LUDC](#)

LA_LUMPROVE

The LA_LUMPROVE function uses LU decomposition to improve the solution to a system of linear equations, $AX = B$, and provides optional error bounds and backward error estimates.

The LA_LUMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the improved solution vector for that set of equations.

LA_LUMPROVE is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgerfs
Double	dgetrfs
Complex	cgetrfs
Double complex	zgetrfs

Table 3-62: LAPACK Routine Basis for LA_LUMPROVE

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
Result = LA_LUMPROVE(Array, Aludc, Index, B, X
  [, BACKWARD_ERROR=variable] [, /DOUBLE]
  [, FORWARD_ERROR=variable])
```

Return Value

The result is an n -element vector or k -by- n array.

Arguments

Array

The original n -by- n array of the linear system.

Aludc

The n -by- n LU decomposition of *Array*, created by the LA_LUDC procedure.

Index

An n -element input vector, created by the LA_LUDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

X

An n -element input vector, or a k -by- n array, containing the approximate solutions to the linear system, created by the LA_LUSOL function.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If B is a vector containing a single linear system, then BACKWARD_ERROR will be a scalar. If B is an array containing k linear systems, then BACKWARD_ERROR will be a k -element vector. The backward error is the smallest relative change in any element of A or B that makes X an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set DOUBLE = 0 to use single-precision for computations and to return a single-precision (real or complex) result. The default is /DOUBLE if *Array* is double precision, otherwise the default is DOUBLE = 0.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If B is a vector containing a single linear system, then FORWARD_ERROR will be a scalar. If B is an array containing k linear systems, then FORWARD_ERROR will be a k -element vector. For each linear system, if X_{true} is the true solution corresponding to X , then the forward error is an estimated upper

bound for the magnitude of the largest element in $(X - X_{true})$ divided by the magnitude of the largest element in X .

Examples

The solution to a given system of equations can be derived and improved by using the following program:

```

PRO ExLA_LUMPROVE
; Define the coefficient array:
  a = [[4, 16000, 17000], $
       [2, 5, 8], $
       [3, 6, 10]]
; Compute the LU decomposition:
aludc = a
; make a copy
LA_LUDC, aludc, index

; Define the right-hand side vector B:
b = [100.1, 0.1, 0.01]
; Find the solution to Ax=b:
x = LA_LUSOL(aludc, index, b)
PRINT, 'LA_LUSOL Solution:', x

; Improve the solution:
xnew = LA_LUMPROVE(a, aludc, index, b, x)
PRINT, 'LA_LUMPROVE Solution:', xnew
END

```

When this program is compiled and run, IDL prints:

```

LA_LUSOL Solution:
-0.397355    -0.334742    0.321033
LA_LUMPROVE Solution:
-0.397432    -0.334865    0.321148

```

The exact solution to 6 decimal places is [-0.397432, -0.334865, 0.321149].

Version History

Introduced 5.6

See Also

[LA_LUDC](#), [LA_LUSOL](#), [LUMPROVE](#)

LA_LUSOL

The LA_LUSOL function is used in conjunction with the LA_LUDC procedure to solve a set of n linear equations in n unknowns, $AX = B$. The parameter A is not the original array, but its LU decomposition, created by the routine LA_LUDC.

The LA_LUSOL function may also be used to solve for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the solution vector for that set of equations.

LA_LUSOL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgetrs
Double	dgetrs
Complex	cgetrs
Double complex	zgetrs

Table 3-63: LAPACK Routine Basis for LA_LUSOL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

$Result = LA_LUSOL(A, Index, B [, /DOUBLE])$

Return Value

The result is an n -element vector or k -by- n array.

Arguments

A

The n -by- n LU decomposition of an array, created by the LA_LUDC procedure.

Note

LA_LUSOL cannot accept any non-square output generated by LA_LUDC.

Index

An n -element input vector, created by the LA_LUDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if A is double precision, otherwise the default is `DOUBLE = 0`.

Examples

Given the system of equations:

$$\begin{aligned} 4u + 16000v + 17000w &= 100.1 \\ 2u + \quad 5v + \quad 8w &= 0.1 \\ 3u + \quad 6v + \quad 10w &= 0.01 \end{aligned}$$

find the solution can be derived by using the following program:

```

PRO ExLA_LUSOL
; Define the coefficient array:
a = [[4, 16000, 17000], $
      [2, 5, 8], $
      [3, 6, 10]]
; Compute the LU decomposition:
aludc = a
; make a copy
LA_LUDC, aludc, index

; Define the right-hand side vector B:
b = [100.1, 0.1, 0.01]

; Compute and print the solution to Ax=b:
x = LA_LUSOL(aludc, index, b)
PRINT, 'LA_LUSOL Solution:', x
END

```

When this program is compiled and run, IDL prints:

```
LA_LUSOL solution:   -0.397355   -0.334742   0.321033
```

The exact solution to 6 decimal places is [-0.397432, -0.334865, 0.321149].

Note

UNIX users may see slightly different output results.

Version History

Introduced 5.6

See Also

[LA_LINEAR_EQUATION](#), [LA_LUDC](#), [LA_LUMPROVE](#), [LUSOL](#)

LA_SVD

The LA_SVD procedure computes the singular value decomposition (SVD) of an n -columns by m -row array as the product of orthogonal and diagonal arrays:

$$A \text{ is real: } A = U S V^T$$

$$A \text{ is complex: } A = U S V^H$$

where U is an orthogonal array containing the left singular vectors, S is a diagonal array containing the singular values, and V is an orthogonal array containing the right singular vectors. The superscript T represents the transpose while the superscript H represents the Hermitian, or transpose complex conjugate.

If $n < m$ then U has dimensions $(n \times m)$, S has dimensions $(n \times n)$, and V^H has dimensions $(n \times n)$. If $n \geq m$ then U has dimensions $(m \times m)$, S has dimensions $(m \times m)$, and V^H has dimensions $(n \times m)$. The following diagram shows the array dimensions:

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} U \end{bmatrix} \cdot \begin{bmatrix} S \end{bmatrix} \cdot \begin{bmatrix} V^T \end{bmatrix} \quad n < m$$

$$\begin{bmatrix} A \end{bmatrix} = \begin{bmatrix} U \end{bmatrix} \cdot \begin{bmatrix} S \end{bmatrix} \cdot \begin{bmatrix} V^T \end{bmatrix} \quad n \geq m$$

LA_SVD is based on the following LAPACK routines:

Output Type	LAPACK Routine	
	QR Iteration	Divide-and-conquer
Float	sgesvd	sgesdd
Double	dgesvd	dgesdd
Complex	cgesvd	cgesdd
Double complex	zgesvd	zgesdd

Table 3-64: LAPACK Routine Basis for LA_SVD

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

```
LA_SVD, Array, W, U, V [, /DOUBLE] [, /DIVIDE_CONQUER]  
[, STATUS=variable]
```

Arguments

Array

The real or complex array to decompose.

W

On output, W is a vector with $\text{MIN}(m, n)$ elements containing the singular values.

U

On output, U is an orthogonal array with $\text{MIN}(m, n)$ columns and m rows used in the decomposition of $Array$. If $Array$ is complex then U will be complex, otherwise U will be real.

V

On output, V is an orthogonal array with $\text{MIN}(m, n)$ columns and n rows used in the decomposition of $Array$. If $Array$ is complex then V will be complex, otherwise V will be real.

Note

To reconstruct $Array$, you will need to take the transpose or Hermitian of V .

Keywords

DIVIDE_CONQUER

If this keyword is set, then the divide-and-conquer method is used to compute the singular vectors, otherwise, QR iteration is used. The divide-and-conquer method is faster at computing singular vectors of large matrices, but uses more memory and may produce less accurate singular values.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if *Array* is double precision, otherwise the default is `DOUBLE = 0`.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: The computation did not converge. The `STATUS` value specifies how many superdiagonals did not converge to zero.

Note

If `STATUS` is not specified, any error messages will output to the screen.

Examples

Construct a sample input array *A*, consisting of smoothed random values:

```

PRO ExLA_SVD
; Create a smoothed random array:
n = 100
m = 200
seed = 12321
a = SMOOTH(RANDOMN(seed, n, m, /DOUBLE), 5)

; Compute the SVD and check reconstruction error:
LA_SVD, a, w, u, v
arecon = u ## DIAG_MATRIX(w) ## TRANSPOSE(v)
PRINT, 'LA_SVD error:', MAX(ABS(arecon - a))

; Keep only the 15 largest singular values
wfiltered = w
wfiltered[15:*] = 0.0
; Reconstruct the array:
afiltered = u ## DIAG_MATRIX(wfiltered) ## TRANSPOSE(v)
percentVar = 100*(w^2)/TOTAL(w^2)
PRINT, 'LA_SVD Variance:', TOTAL(percentVar[0:14])
END

```

When this program is compiled and run, IDL prints:

```
LA_SVD error: 1.0103030e-014  
LA_SVD variance: 82.802816
```

Note

More than 80% of the variance is contained in the 15 largest singular values.

Version History

Introduced 5.6

See Also

[LA_CHOLDC](#), [LA_LUDC](#), [SVDC](#)

LA_TRIDC

The LA_TRIDC procedure computes the LU decomposition of a tridiagonal ($n \times n$) array as $Array = L U$, where L is a product of permutation and unit lower bidiagonal arrays, and U is upper triangular with nonzero elements only in the main diagonal and the first two superdiagonals.

LA_TRIDC is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgttrf
Double	dgttrf
Complex	cgttrf
Double complex	zgttrf

Table 3-65: LAPACK Routine Basis for LA_TRIDC

For more details, see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_TRIDC, AL, A, AU, U2, Index [, /DOUBLE] [, STATUS=variable]

Arguments

AL

A named vector of length $(n - 1)$ containing the subdiagonal elements of an array. This procedure returns *AL* as the $(n - 1)$ elements of the lower bidiagonal array from the LU decomposition.

A

A named vector of length n containing the main diagonal elements of an array. This procedure returns *A* as the n diagonal elements of the upper array from the LU decomposition.

AU

A named vector of length $(n - 1)$ containing the superdiagonal elements of an array. This procedure returns *AU* as the $(n - 1)$ superdiagonal elements of the upper array.

U2

An output vector that contains the $(n - 2)$ elements of the second superdiagonal of the upper array.

Index

An output vector that records the row permutations which occurred as a result of partial pivoting. For $1 < j < n$, row j of the matrix was interchanged with row *Index*[j].

Note

Row numbers within *Index* start at one rather than zero.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if *AL* is double precision, otherwise the default is `DOUBLE = 0`.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: One of the diagonal elements of *U* is zero. The `STATUS` value specifies which value along the diagonal (starting at one) is zero.

Note

If `STATUS` is not specified, any error messages will output to the screen.

Examples

Create a test program to compute the LU decomposition of a tridiagonal array:

```

PRO EX_LA_TRIDC
  ; Create a random tridiagonal array.
  n = 9
  seed = 12321
  AL = RANDOMN(seed, n-1)
  A = RANDOMN(seed, n)
  AU = RANDOMN(seed, n-1)

  ; Construct tridiagonal array.
  Array = DIAG_MATRIX(AL, -1) + DIAG_MATRIX(A) + $
          DIAG_MATRIX(AU, 1)

  ; Compute the LU decomposition.
  LA_TRIDC, AL, A, AU, U2, Index

  ; Adjust from LAPACK back to IDL indexing.
  Index = Index - 1

  ; Create upper and lower arrays.
  Upper = DIAG_MATRIX(A) + $
          DIAG_MATRIX(AU, 1) + DIAG_MATRIX(U2, 2)
  Lower = DIAG_MATRIX(AL, -1) + IDENTITY(n)

  ; To conserve storage, LA_TRIDC keeps all lower diagonal
  ; elements in AL, regardless of row. The Index array
  ; tells which subdiagonals need to be shifted down.
  ; Loop starts at 1 since there aren't any subdiagonals
  ; to the left of the first diagonal element.
  FOR i = 1, n-2 DO BEGIN
    IF (Index[i] NE i) THEN $
      Lower[0:i-1, [i, i+1]] = Lower[0:i-1, [i+1, i]]
  ENDFOR

  ; Permute the row order.
  FOR i = n-2, 0, -1 DO BEGIN
    IF (Index[i] NE i) THEN $
      Lower[*, [i, i+1]] = Lower[*, [i+1, i]]
  ENDFOR

  ; Reconstruct the array and check the difference:
  Arecon = Lower ## Upper
  PRINT, 'LA_TRIDC error:', MAX(ABS(Arecon - Array))
END

```

When this program is compiled and run, IDL prints:

```
LA_TRIDC error: 1.50427e-008
```

Version History

Introduced 5.6

See Also

[LA_TRIMPROVE](#), [LA_TRISOL](#)

LA_TRIMPROVE

The LA_TRIMPROVE function improves the solution to a system of linear equations with a tridiagonal array, $AX = B$, and provides optional error bounds and backward error estimates.

The LA_TRIMPROVE function may also be used to improve the solutions for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the improved solution vector for that set of equations.

LA_TRIMPROVE is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgtrfs
Double	dgtrfs
Complex	cgtrfs
Double complex	zgtrfs

Table 3-66: LAPACK Routine Basis for LA_TRIMPROVE

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_TRIMPROVE(*AL*, *A*, *AU*, *DAL*, *DA*, *DAU*, *DU2*, *Index*, *B*, *X*
 [, [BACKWARD_ERROR=variable](#)] [, /[DOUBLE](#)]
 [, [FORWARD_ERROR=variable](#)])

Return Value

The result is an n -element vector or k -by- n array.

Arguments

AL

A vector of length $(n - 1)$ containing the subdiagonal elements of the original array.

A

A vector of length n containing the main diagonal elements of the original array.

AU

A vector of length $(n - 1)$ containing the superdiagonal elements of the original array.

DAL

The $(n - 1)$ elements of the lower bidiagonal array, created by the LA_TRIDC procedure.

DA

The n diagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

DAU

The $(n - 1)$ superdiagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

DU2

The $(n - 2)$ elements of the second superdiagonal of the upper triangular array, created by the LA_TRIDC procedure.

Index

An input vector, created by the LA_TRIDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

X

An n -element input vector, or a k -by- n array, containing the approximate solutions to the linear system, created by the LA_TRISOL function.

Keywords

BACKWARD_ERROR

Set this keyword to a named variable that will contain the relative backward error estimate for each linear system. If B is a vector containing a single linear system, then `BACKWARD_ERROR` will be a scalar. If B is an array containing k linear systems, then `BACKWARD_ERROR` will be a k -element vector. The backward error is the smallest relative change in any element of A or B that makes X an exact solution.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if `AL` is double precision, otherwise the default is `DOUBLE = 0`.

FORWARD_ERROR

Set this keyword to a named variable that will contain the estimated forward error bound for each linear system. If B is a vector containing a single linear system, then `FORWARD_ERROR` will be a scalar. If B is an array containing k linear systems, then `FORWARD_ERROR` will be a k -element vector. For each linear system, if X_{true} is the true solution corresponding to X , then the forward error is an estimated upper bound for the magnitude of the largest element in $(X - X_{true})$ divided by the magnitude of the largest element in X .

Examples

Given the tridiagonal system of equations:

$$\begin{array}{rcl} -4t + u & & = 6 \\ 2t - 4u + v & & = -8 \\ & 2u - 4v + w & = -5 \\ & & 2v - 4w = 8 \end{array}$$

the solution can be found and improved by using the following program:

```
PRO ExLA_TRIMPROVE
; Define array a:
upper = [1, 1, 1]
adiag = [-4, -4, -4, -4]
alower = [2, 2, 2]
; Define right-hand side vector b:
b = [6, -8, -5, 8]
```

```
; Decompose a:
dlower = alower
darray = adiaq
dupper = aupper
LA_TRIDC, dlower, darray, dupper, u2, index

; Compute and improve the solution:
x = LA_TRISOL(dlower, darray, dupper, u2, index, b)
xnew = LA_TRIMPROVE(Alower, Adiaq, Aupper, $
    dlower, darray, dupper, u2, index, b, x)
PRINT, 'LA_TRISOL improved solution:'
PRINT, xnew
END
```

When this program is compiled and run, IDL prints:

```
LA_TRISOL improved solution:
-1.00000      2.00000      2.00000      -1.00000
```

Version History

Introduced 5.6

See Also

[LA_TRIDC](#), [LA_TRISOL](#)

LA_TRIQL

The LA_TRIQL procedure uses the QL and QR variants of the implicitly-shifted QR algorithm to compute the eigenvalues and eigenvectors of a symmetric tridiagonal array. The LA_TRIRED routine can be used to reduce a real symmetric (or complex Hermitian) array to tridiagonal form suitable for input to this procedure.

LA_TRIQL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	ssteqr
Double	dsteqr
Complex	csteqr
Double complex	zsteqr

Table 3-67: LAPACK Routine Basis for LA_TRIQL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_TRIQL, D, E [, A] [, /DOUBLE] [, STATUS=*variable*]

Arguments

D

A named vector of length n containing the real diagonal elements, optionally created by the LA_TRIRED procedure. Upon output, D is replaced by a real vector of length n containing the eigenvalues.

E

The $(n - 1)$ real subdiagonal elements, optionally created by the LA_TRIRED procedure. On output, the values within E are destroyed.

A

An optional named variable that returns the eigenvectors as a set of n row vectors. If the eigenvectors of a tridiagonal array are desired, A should be input as an identity

array. If the eigenvectors of an array that has been reduced by LA_TRIRED are desired, *A* should be input as the Array output from LA_TRIRED. If *A* is not input, then eigenvectors are not computed. *A* may be either real or complex.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `DOUBLE = 0` if none of the inputs are double precision. If *A* is not input, then the default is `/DOUBLE` if *D* is double precision. If *A* is input, then the default is `/DOUBLE` if *A* is double precision (real or complex).

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- `STATUS = 0`: The computation was successful.
- `STATUS > 0`: The algorithm failed to find all eigenvalues in $30n$ iterations. The `STATUS` value specifies how many elements of *E* have not converged to zero.

Note

If `STATUS` is not specified, any error messages will be output to the screen.

Examples

The following example program computes the eigenvalues and eigenvectors of a given symmetric array:

```

PRO ExLA_TRIQL
; Create a symmetric random array:
n = 4
seed = 12321
Array = RANDOMN(seed, n, n)
array = array + TRANSPOSE(array)

; Reduce to tridiagonal form
q = array ; make a copy
LA_TRIRED, q, d, e

; Compute eigenvalues and eigenvectors

```

```
eigenvalues = d
eigenvectors = q
LA_TRIQL, eigenvalues, e, eigenvectors
PRINT, 'LA_TRIQL eigenvalues:'
PRINT, eigenvalues
END
```

When this program is compiled and run, IDL prints:

```
LA_TRIQL eigenvalues:
-2.87710    -0.663354    2.92018    3.59648
```

Version History

Introduced 5.6

See Also

[LA_TRIED](#), [TRIQLDOUBLE](#)

LA_TRIRED

The LA_TRIRED procedure reduces a real symmetric or complex Hermitian array to real tridiagonal form T . If the array is real symmetric then the decomposition is $A = Q T Q^T$, where Q is orthogonal. If the array is complex Hermitian then the decomposition is $A = Q T Q^H$, where Q is unitary. The superscript T represents the transpose while superscript H represents the Hermitian, or transpose complex conjugate.

LA_TRIRED is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	ssytrd, sorgtr
Double	dsytrd, dorgtr
Complex	chetrd, cungr
Double complex	zhetrd, zungr

Table 3-68: LAPACK Routine Basis for LA_TRIRED

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

LA_TRIRED, *Array*, *D*, *E* [, /DOUBLE] [, /UPPER]

Arguments

Array

A named variable containing the real or complex array to decompose. Only the lower triangular portion of *Array* is used (or upper if the /UPPER keyword is set). This procedure returns *Array* as the real orthogonal (or complex unitary) Q array used to reduce the original array to tridiagonal form.

D

An n -element output vector containing the real diagonal elements of the tridiagonal array. Note that D is always real.

E

An $(n - 1)$ element output vector containing the real subdiagonal elements of the tridiagonal array. Note that E is always real.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real) result. The default is `/DOUBLE` if *Array* is double precision, otherwise the default is `DOUBLE = 0`.

UPPER

If this keyword is set, then only the upper triangular portion of *Array* is used, and the upper triangular array is returned. The default is to use the lower triangular portion and return the lower triangular array.

Examples

See `LA_TRIQL` for an example of using this procedure.

Version History

Introduced 5.6

See Also

[LA_TRIQL](#), [TRIRED](#)

LA_TRISOL

The LA_TRISOL function is used in conjunction with the LA_TRIDC procedure to solve a set of n linear equations in n unknowns, $AX = B$, where A is a tridiagonal array. The parameter A is input not as the original array, but as its LU decomposition, created by the routine LA_TRIDC.

The LA_TRISOL function may also be used to solve for multiple systems of linear equations, with each column of B representing a different set of equations. In this case, the result is a k -by- n array where each of the k columns represents the solution vector for that set of equations.

LA_TRISOL is based on the following LAPACK routines:

Output Type	LAPACK Routine
Float	sgttrs
Double	dgttrs
Complex	cgttrs
Double complex	zgttrs

Table 3-69: LAPACK Routine Basis for LA_TRISOL

For details see Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, 1999.

Syntax

Result = LA_TRISOL(*AL*, *A*, *AU*, *U2*, *Index*, *B* [, /DOUBLE])

Return Value

The result is an n -element vector or k -by- n array.

Arguments

AL

The $(n - 1)$ elements of the lower bidiagonal array, created by the LA_TRIDC procedure.

A

The n diagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

AU

The $(n - 1)$ superdiagonal elements of the upper triangular array, created by the LA_TRIDC procedure.

U2

The $(n - 2)$ elements of the second superdiagonal of the upper triangular array, created by the LA_TRIDC procedure.

Index

An input vector, created by the LA_TRIDC procedure, containing the row permutations which occurred as a result of partial pivoting.

B

An n -element input vector containing the right-hand side of the linear system, or a k -by- n array, where each of the k columns represents a different linear system.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision (real or complex) result. Set `DOUBLE = 0` to use single-precision for computations and to return a single-precision (real or complex) result. The default is `/DOUBLE` if `AL` is double precision, otherwise the default is `DOUBLE = 0`.

Examples

For an example of using this routine see [LA_TRIMPROVE](#).

Version History

Introduced 5.6

See Also

[LA_TRIDC](#), [LA_TRIMPROVE](#), [TRISOL](#)

LABEL_DATE

The LABEL_DATE function can be used, in conjunction with the [XYZ]TICKFORMAT keyword to IDL plotting routines, to easily label axes with dates and times.

This routine is written in the IDL language. Its source code can be found in the file `label_date.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LABEL_DATE( [DATE_FORMAT=string/string array]
    [, AM_PM=2-element vector of strings]
    [, DAYS_OF_WEEK=7-element vector of strings]
    [, MONTHS=12-element vector of strings] [, OFFSET=value] [, /ROUND_UP] )
```

and then,

```
PLOT, x, y, XTICKFORMAT = 'LABEL_DATE'
```

Return Value

Returns a date formatting definition.

Arguments

If LABEL_DATE is being called to initialize string formats, it should be called with no arguments and the DATE_FORMAT keyword should be set.

Keywords

Note

The settings for LABEL_DATE remain in effect for all subsequent calls to LABEL_DATE. To restore any default settings, call LABEL_DATE again with the appropriate keyword set to either a null string (") or to 0, depending upon the data type of that keyword.

AM_PM

Set this keyword to a two-element string array that contains the names to be used with '%A'. The default is ['am','pm'].

DATE_FORMAT

Set this keyword to a format string or array of format strings. Each string corresponds to an axis level as provided by the [XYZ]TICKUNITS keyword to the plotting routine. If there are fewer strings than axis levels, then the strings are cyclically repeated. A string can contain any of the following codes:

Code	Description
%M	Month name.
%N	Month number (2 digits).
%D	Day of month (2 digits).
%Y	Year (4 digits, or 5 digits for negative years).
%Z	Last 2 digits of the year.
%W	Day of the week.
%A	AM or PM (%H is then 12-hour instead of 24-hour).
%H	Hours (2 digits).
%I	Minutes (2 digits).
%S	Seconds (2 digits), followed optionally by %n where n is an integer 0-9 representing the number of digits after the decimal point for seconds; the default is no decimal places.
%%	Represents the % character.

Table 3-70: DATE_FORMAT Codes

Other items you can include can consist of:

- Any other text characters in the format string.
- Any vector font positioning and font change commands. For more information, see [“Embedded Formatting Commands”](#) in Appendix H.

If DATE_FORMAT is not specified then the default is the standard 24-character system format, '%W %M %D %H:%I:%S %Y'.

The following table contains some examples of `DATE_FORMAT` strings and the resulting output:

DATE_FORMAT String	Example Result
'%D/%N/%Y'	11/12/1993
'%M!C%Y' Note - !C is the code for a newline character.	Dec 1993
'%H:%I:%S'	21:33:58
'%H:%I:%S%3'	21:33:58.125
'%W, %M %D, %H %A'	Sat, Jan 01, 9 pm
'%S seconds'	60 seconds

Table 3-71: DATE_FORMAT Examples

DAYS_OF_WEEK

Set this keyword to a seven-element string array that contains the names to be used with '%W'. The default is the three-letter English abbreviations, ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'].

MONTHS

Set this keyword to a twelve-element string array that contains the names to be used with '%M'. The default is the three-letter English abbreviations, ['Jan', 'Feb', ..., 'Dec'].

OFFSET

Set this keyword to a value representing the offset to be added to each tick value before conversion to a label. This keyword is usually used when your axis values are measured relative to a certain starting time. In this case, `OFFSET` should be set to the Julian date of the starting time.

ROUND_UP

Set this keyword to force times to be rounded up to the nearest minute after the 30 second mark in the `DATE_FORMAT` string. The default is for times to be truncated to the smallest time unit.

Note

If ROUND_UP is set, it will remain set unless set back to 0.

Examples

This example creates a sample plot that has a date axis from Jan 1 to June 30, 2000:

```

; Create format strings for a two-level axis:
dummy = LABEL_DATE(DATE_FORMAT=['%D-%M', '%Y'])

;Generate the Date/Time data
time = TIMEGEN(START=JULDAY(1,1,2000), FINAL=JULDAY(6,30,2000))

;Generate the Y-axis data
data = RANDOMN(seed, N_ELEMENTS(time))

;Plot the data
PLOT, time, data, XTICKUNITS = ['Time', 'Time'], $
    XTICKFORMAT='LABEL_DATE', XSTYLE=1, XTICKS=6, YMARGIN=[6,2]

```

For more examples, see “[XYZ]TICKFORMAT” on page 4345.

Version History

Introduced: Pre 4.0

See Also

“[XYZ]TICKFORMAT” on page 4345, CALDAT, JULDAY, SYSTIME, TIMEGEN, “Format Codes” in Chapter 11 of the *Building IDL Applications* manual.

LABEL_REGION

The LABEL_REGION function consecutively labels all of the regions, or blobs, of a bi-level image with a unique region index. This process is sometimes called “blob coloring”. A region is a set of non-zero pixels within a neighborhood around the pixel under examination.

The argument for LABEL_REGION is an n -dimensional bi-level integer type array—only zero and non-zero values are considered.

Statistics on each of the regions may be easily calculated using the HISTOGRAM function as shown in the examples below.

Syntax

Result = LABEL_REGION(*Data* [, /ALL_NEIGHBORS] [, /ULONG])

Return Value

The result of the function is an integer array of the same dimensions with each pixel containing its region index. A region index of zero indicates that the original pixel was zero and belongs to no region. Output values range from 0 to the number of regions.

Arguments

Data

A n -dimensional image to be labeled. *Data* is converted to integer type if necessary. Pixels at the edges of *Data* are considered to be zero.

Keywords

ALL_NEIGHBORS

Set this keyword to indicate that all adjacent neighbors to a given pixel should be searched. (This is sometimes called 8-neighbor searching when the image is 2-dimensional). The default is to search only the neighbors that are exactly one unit in distance from the current pixel (sometimes called 4-neighbor searching when the image is 2-dimensional).

EIGHT

This keyword is now obsolete. It has been replaced by the ALL_NEIGHBORS keyword (because this routine now handles N-dimensional data).

ULONG

Set this keyword to specify that the output array should be an unsigned long integer.

Examples

Example 1

This example counts the number of distinct regions within an image, and their population. Note that region 0 is the set of zero pixels that are not within a region:

```
image = DIST(40)

; Get blob indices:
b = LABEL_REGION(image)

; Get population of each blob:
h = HISTOGRAM(b)
FOR i=0, N_ELEMENTS(h)-1 DO PRINT, 'Region ',i, $
    ', Population = ', h[i]
```

Example 2

This example also prints the average value and standard deviation of each region:

```
image = DIST(40)

; Get blob indices:
b = LABEL_REGION(image)

; Get population and members of each blob:
h = HISTOGRAM(b, REVERSE_INDICES=r)

; Each region
FOR i=0, N_ELEMENTS(h)-1 DO BEGIN
    ;Find subscripts of members of region i.
    p = r[r[i]:r[i+1]-1]

    ; Pixels of region i
    q = image[p]
```

```
        PRINT, 'Region ', i, $  
            ', Population = ', h[i], $  
            ', Standard Deviation = ', STDEV(q, mean), $  
            ', Mean = ', mean  
    ENDFOR
```

Version History

Introduced: Pre 4.0

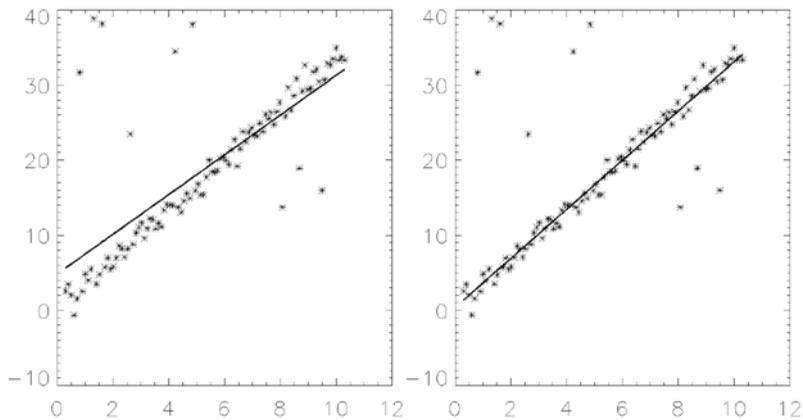
See Also

[ANNOTATE](#), [DEFROI](#), [HISTOGRAM](#), [SEARCH2D](#)

LADFIT

The LADFIT function fits the paired data $\{x_i, y_i\}$ to the linear model, $y = A + Bx$, using a “robust” least absolute deviation method.

The figure below displays a two-dimensional distribution that is fitted to the model $y = A + Bx$, using a minimized Chi-square error criterion (left) and a “robust” least absolute deviation technique (right). The use of the Chi-square error statistic can result in a poor fit due to an undesired sensitivity to outlying data.



This routine is written in the IDL language. Its source code can be found in the file `ladfit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LADFIT( X, Y [, ABSDEV=variable] [, /DOUBLE] )
```

Return Value

The result is a two-element vector containing the model parameters, A and B.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector. Note that the X vector must be sorted into ascending order.

Y

An n -element integer, single-, or double-precision floating-point vector. Note that the elements of the Y vector must be paired with the appropriate elements of X .

Keywords

ABSDEV

Set this keyword to a named variable that will contain the mean of the absolute deviation of the *Result* and Y .

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```

; Define two n-element vectors of paired data:
X = [-3.20, 4.49, -1.66, 0.64, -2.43, -0.89, -0.12, 1.41, $
      2.95, 2.18, 3.72, 5.26]
Y = [-7.14, -1.30, -4.26, -1.90, -6.19, -3.98, -2.87, -1.66, $
      -0.78, -2.61, 0.31, 1.74]

; Sort the X values into ascending order, and sort the Y values to
; match the new order of the elements in X:
XX = X[SORT(X)]
YY = Y[SORT(X)]

; Compute the model parameters, A and B:
PRINT, LADFIT(XX, YY)

```

IDL prints:

```
-3.15301      0.930440
```

Version History

Introduced: 4.0

See Also

[COMFIT](#), [CURVEFIT](#), [LINFIT](#), [SORT](#)

LAGUERRE

The LAGUERRE function returns the value of the associated Laguerre polynomial $L_n^k(x)$. The associated Laguerre polynomials are solutions to the differential equation:

$$xy'' + (k + 1 - x)y' + ny = 0$$

with orthogonality constraint:

$$\int_0^\infty e^{-x} x^{k+1} L_m^k(x) L_n^k(x) dx = \frac{(n+k)!}{n!} \delta_{mn}$$

Laguerre polynomials are used in quantum mechanics, for example, where the wave function for the hydrogen atom is given by the Laguerre differential equation.

This routine is written in the IDL language. Its source code can be found in the file `laguerre.pro` in the `lib` subdirectory of the IDL distribution.

This routine is written in the IDL language. Its source code can be found in the file `laguerre.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = LAGUERRE(*X*, *N* [, *K*] [, **COEFFICIENTS**=*variable*] [, **/DOUBLE**])

Return Value

This function returns a scalar or array with the same dimensions as *X*. If *X* is double-precision or if the **DOUBLE** keyword is set, the result is double-precision complex, otherwise the result is single-precision complex.

Arguments

X

The value(s) at which $L_n^k(x)$ is evaluated. *X* can be either a scalar or an array.

N

A scalar integer, $N \geq 0$, specifying the order *n* of $L_n^k(x)$. If *N* is of type float, it will be truncated.

K

A scalar, $K \geq 0$, specifying the order k of $L_n^k(x)$. If K is not specified, the default $K = 0$ is used and the Laguerre polynomial, $L_n(x)$, is returned.

Keywords

COEFFICIENTS

Set this keyword to a named variable that will contain the polynomial coefficients in the expansion $C[0] + C[1]x + C[2]x^2 + \dots$.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To compute the value of the Laguerre polynomial at the following X values:

```
;Define the parametric X values:
X = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

;Compute the Laguerre polynomial of order N=2, K=1:
result = LAGUERRE(X, 2, 1)

;Print the result:
PRINT, result
```

IDL prints:

```
3.00000  2.42000  1.88000  1.38000  0.920000  0.500000
```

This is the exact solution vector to six-decimal accuracy.

Version History

Introduced: 5.4

See Also

[LEGENDRE](#), [SPHER_HARM](#)

LEEFILT

The LEEFILT function performs the Lee filter algorithm on an image array using a box of size $2N+1$. This function can also be used on vectors. The Lee technique smooths additive image noise by generating statistics in a local neighborhood and comparing them to the expected values.

This routine is written in the IDL language. It is based upon the algorithm published by Lee (*Optical Engineering* 25(5), 636-646, May 1986). Its source code can be found in the file `leefilt.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LEEFILT( A [, N [, Sig]] [, /DOUBLE] [, /EXACT] )
```

Return Value

This function returns an array with the same dimensions as *A*. If any of the inputs are double-precision or if the `DOUBLE` keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

A

The input image array or one-dimensional vector.

N

The size of the filter box is $2N+1$. The default value is 5.

Sig

Estimate of the standard deviation. The default is 5. If *Sig* is negative, IDL interactively prompts for a value of `sigma`, and displays the resulting image using `TVSCL` (for arrays) or `PLOT` (for vectors). To end this cycle, enter a value of 0 (zero) for `sigma`.

Keywords

DOUBLE

Set this keyword to force the computations to be done in double-precision arithmetic.

EXACT

Set this keyword to apply a more accurate (but slower) implementation of the Lee filter.

Version History

Introduced: Original

See Also

[DIGITAL_FILTER](#), [MEDIAN](#), [SMOOTH](#), [VOIGT](#)

LEGENDRE

The LEGENDRE function returns the value of the associated Legendre polynomial $P_l^m(x)$. The associated Legendre functions are solutions to the differential equation:

$$(1-x^2)y'' - 2xy' + \left[l(l+1) - \frac{m^2}{(1-x^2)} \right] y = 0$$

with orthogonality constraints:

$$\int_{-1}^{+1} P_l^m(x) P_k^n(x) dx = \frac{2}{2l+1} \frac{(l+m)!}{(l-m)!} \delta_{lk} \delta_{mn}$$

The Legendre polynomials are the solutions to the Legendre equation with $m = 0$. For positive m , the associated Legendre functions can be written in terms of the Legendre polynomials as:

$$P_l^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x)$$

Associated polynomials for negative m are related to positive m by:

$$P_l^{-m}(x) = (-1)^m \frac{(l-m)!}{(l+m)!} P_l^m(x)$$

LEGENDRE is based on the routine *plgndr* described in section 6.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

Result = LEGENDRE(*X*, *L* [, *M*] [, /DOUBLE])

Return Value

If all arguments are scalar, the function returns a scalar. If all arguments are arrays, the function matches up the corresponding elements of *X*, *L*, and *M*, returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other arguments are arrays, the function uses the scalar value with each element of the arrays, and returns an array with the same dimensions as the smallest input array.

If any of the arguments are double-precision or if the `DOUBLE` keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

X

The expression for which $P_l^m(x)$ is evaluated. Values for X must be in the range $-1 \leq X \leq 1$.

L

An integer scalar or array, $L \geq 0$, specifying the order l of $P_l^m(x)$. If L is of type float, it will be truncated.

M

An integer scalar or array, $-L \leq M \leq L$, specifying the order m of $P_l^m(x)$. If M is not specified, then the default $M = 0$ is used and the Legendre polynomial, $P_l(x)$, is returned. If M is of type float, it will be truncated.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

Example 1

Compute the value of the Legendre polynomial at the following X values:

```

; Define the parametric X values:
X = [-0.75, -0.5, -0.25, 0.25, 0.5, 0.75]

; Compute the Legendre polynomial of order L=2:
result = LEGENDRE(X, 2)

; Print the result:
PRINT, result

```

The result of this is:

```
0.343750 -0.125000 -0.406250 -0.406250 -0.125000 0.343750
```

Example 2

Compute the value of the associated Legendre polynomial at the same X values:

```
; Compute the associated Legendre polynomial of order L=2, M=1:  
result = LEGENDRE(X, 2, 1)  
; Print the result:  
PRINT, result
```

IDL prints:

```
1.48824 1.29904 0.726184 -0.726184 -1.29904 -1.48824
```

This is the exact solution vector to six-decimal accuracy.

Version History

Introduced: 5.4

See Also

[SPHER_HARM](#), [LAGUERRE](#)

LINBCG

The LINBCG function is used in conjunction with SPRSIN to solve a set of n sparse linear equations with n unknowns using the iterative biconjugate gradient method.

LINBCG is based on the routine `linbcg` described in section 2.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

Numerical Recipes recommends using double-precision arithmetic to perform this computation.

Syntax

```
Result = LINBCG( A, B, X [, /DOUBLE] [, ITOL={4 | 5 | 6 | 7}] [, TOL=value]
               [, ITER=variable] [, ITMAX=value] )
```

Return Value

The result is an n -element vector.

Arguments

A

A row-indexed sparse array created by the SPRSIN function.

B

An n -element vector containing the right-hand side of the linear system $\mathbf{Ax}=\mathbf{b}$.

X

An n -element vector containing the initial solution of the linear system.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ITOL

Use this keyword to specify which convergence test should be used. Set ITOL to one of the following:

1. Iteration stops when $|A \cdot x - b| / |b|$ is less than the value specified by TOL.
2. Iteration stops when $|\tilde{A}^{-1} \cdot (A \cdot x - b)| / |\tilde{A}^{-1} \cdot b|$ (where \tilde{A} is a “preconditioning” matrix close to A) is less than the value specified by TOL.
3. The routine uses its own estimate of error in x . Iteration stops when the magnitude of the error divided by the magnitude of x is less than the value specified by TOL. This is the default setting.
4. The same as 3, except that the routine uses the largest (in absolute value) component of the error and the largest component of x rather than the vector magnitudes.

TOL

Use this keyword to specify the desired convergence tolerance. For single-precision calculations, the default value is 1.0×10^{-7} . For double-precision values, the default is 1.0×10^{-14} .

ITER

Use this keyword to specify an output variable that will be set to the number of iterations performed.

ITMAX

The maximum allowed number of iterations. The default is n^2 .

Examples

```

; Begin with an array A:
A = [[ 5.0,  0.0, 0.0,  1.0, -2.0], $
      [ 3.0, -2.0, 0.0,  1.0,  0.0], $
      [ 4.0, -1.0, 0.0,  2.0,  0.0], $
      [ 0.0,  3.0, 3.0,  1.0,  0.0], $
      [-2.0,  0.0, 0.0, -1.0,  2.0]]

; Define a right-hand side vector B:
B = [7.0, 1.0, 3.0, 3.0, -4.0]

; Start with an initial guess at the solution:
X = REPLICATE(1.0, N_ELEMENTS(B))

```

```
; Solve the linear system Ax=b:  
result = LINBCG(SPRSIN(A), B, X)
```

```
; Print the result:  
PRINT, result
```

IDL prints:

```
1.00000  1.00000  8.94134e-008  -2.37107e-007  -1.00000
```

The exact solution is [1, 1, 0, 0, -1].

Version History

Introduced: 4.0

See Also

[FULSTR](#), [READ_SPR](#), [SPRSAB](#), [SPRSAX](#), [SPRSIN](#), [SPRSTP](#), [WRITE_SPR](#)

LINDGEN

The LINDGEN function creates a longword integer array with the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = LINDGEN(D_1 [, \dots, D_8])$$

Return Value

Returns a longword integer array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create L, a 10-element by 10-element longword array where each element is set to the value of its one-dimensional subscript, enter:

```
L = LINDGEN(10, 10)
```

Version History

Introduced: Original

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [L64INDGEN](#),
[SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

LINFIT

The LINFIT function fits the paired data $\{x_i, y_i\}$ to the linear model, $y = A + Bx$, by minimizing the chi-square error statistic.

This routine is written in the IDL language. Its source code can be found in the file `linfit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LINFIT( X, Y [, CHISQ=variable] [, COVAR=variable] [, /DOUBLE]  
[, MEASURE_ERRORS=vector] [, PROB=variable] [, SIGMA=variable]  
[, YFIT=variable] )
```

Return Value

The result is a two-element vector containing the linear model parameters [A, B].

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An n -element integer, single-, or double-precision floating-point vector.

Keywords

CHISQ

Set this keyword to a named variable that will contain the value of the unreduced chi-square goodness-of-fit statistic.

COVAR

Set this keyword to a named variable that will contain the covariance matrix of the coefficients.

Note

The COVAR matrix depends only upon the independent variable X and (optionally) the MEASURE_ERRORS. The values do not depend upon Y . See section 15.4 of *Numerical Recipes in C* (Second Edition) for details.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

MEASURE_ERRORS

Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y .

Note

For Gaussian errors (e.g., instrumental uncertainties), MEASURE_ERRORS should be set to the standard deviations of each point in Y . For Poisson or statistical weighting, MEASURE_ERRORS should be set to $\text{SQRT}(\text{ABS}(Y))$.

PROB

Set this keyword to a named variable that will contain the probability that the computed fit would have a value of CHISQ or greater. If PROB is greater than 0.1, the model parameters are “believable”. If PROB is less than 0.1, the accuracy of the model parameters is questionable.

SDEV

*The SDEV keyword is obsolete and has been replaced by the **MEASURE_ERRORS** keyword. Code that uses the SDEV keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. The definition of the MEASURE_ERRORS keyword is identical to that of the SDEV keyword.*

SIGMA

Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters

Note

If MEASURE_ERRORS is omitted, then you are assuming that a straight line is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number

of coefficients. See section 15.2 of *Numerical Recipes in C* (Second Edition) for details.

YFIT

Set this keyword equal to a named variable that will contain the vector of calculated *Y* values.

Examples

```
; Define two n-element vectors of paired data:
X = [-3.20, 4.49, -1.66, 0.64, -2.43, -0.89, -0.12, 1.41, $
      2.95, 2.18, 3.72, 5.26]
Y = [-7.14, -1.30, -4.26, -1.90, -6.19, -3.98, -2.87, -1.66, $
      -0.78, -2.61, 0.31, 1.74]

; Define an n-element vector of Poisson measurement errors:
measure_errors = SQRT(ABS(Y))

; Compute the model parameters, A and B, and print the result:
result = LINFIT(X, Y, MEASURE_ERRORS=measure_errors)
PRINT, result
```

IDL prints:

```
-3.16574    0.829856
```

Version History

Introduced: 4.0

See Also

[COMFIT](#), [CURVEFIT](#), [GAUSSFIT](#), [LADFIT](#), [LMFIT](#), [POLY_FIT](#), [REGRESS](#), [SFIT](#), [SVDFIT](#)

LINKIMAGE

The LINKIMAGE procedure merges routines written in other languages with IDL at run-time. Each call to LINKIMAGE defines a new system procedure or function by specifying the routine's name, the name of the file containing the code, and the entry point name. The name of your routine is added to IDL's internal system routine table, making it available in the same manner as any other IDL built-in routine.

LINKIMAGE can also be used to add graphics device drivers.

Warning

Using LINKIMAGE requires intimate knowledge of the internals of IDL, and is not for use by the novice user. We recommend use of CALL_EXTERNAL, which has a simpler interface, instead of LINKIMAGE unless your application specifically requires it. To use LINKIMAGE, you should be familiar with the material in the *IDL External Development Guide*.

LINKIMAGE uses the dynamic linking interface supported by the operating system to do its work. Programmers should be familiar with the services supported by their system in order to better understand LINKIMAGE:

- Under UNIX, LINKIMAGE uses the `dlopen()` interface to the dynamic linker in all cases except for HP-UX (which uses `shl_load()`) and AIX (which uses `load()`).
- Under Windows, LINKIMAGE uses `LoadLibrary()` to load a 32-bit, Win32 DLL.

Note

Modules must be merged via LINKIMAGE before other procedures and functions that call them are compiled, or the compilation of those routines will fail. Note that because routines merged via LINKIMAGE are considered built-in routines by IDL, declaring the routine with the FORWARD_FUNCTION statement will not eliminate this restriction.

Syntax

```
LINKIMAGE, Name, Image [, Type [, Entry]] [, /DEVICE] [, /FUNCT]  
[, /KEYWORDS] [, MAX_ARGS=value] [, MIN_ARGS=value]
```

Arguments

Name

A string containing the IDL name of the function, procedure or device routine which is to be merged. When loading a device driver, *Name* contains the name of the global `DEVICE_DEF` structure in the driver. Upon successful loading of the routine, a new procedure or function with the given name will exist, or the new device driver will be loaded.

Image

A string containing the full path specification of the dynamically loaded object file. See your system documentation on sharable libraries or DLLs for details.

Type

An optional scalar integer parameter that contains 0 (zero) for a procedure, 1 (one) for a function, and 2 for a device driver. The keyword parameters `DEVICE` and `FUNCT` can also be used to indicate the type of routine being merged. The default value is 0, for procedure.

Entry

An optional string that contains the name of the symbol which is the entry point of the procedure or function. With some compilers or operating systems, this name may require the addition of leading or trailing characters. For example, some UNIX C compilers add a leading underscore to the beginning of a function name, and some UNIX FORTRAN compilers add a trailing underscore.

If *Entry* is not supplied, `LINKIMAGE` will provide a default name by converting the value supplied for *Name* to lower case and adding any special characters (leading or trailing underscores) typical of the system.

Warning

Under Microsoft Windows operating systems, only `cdecl` functions can be used with `LINKIMAGE`. Attempting to use routines with other calling conventions will yield undefined results, including memory corruption or even IDL crashing.

The Windows operating system has two distinct system defined standards that govern how routines pass arguments: `stdcall`, which is used by much of the operating system as well as languages such as Visual Basic, and `cdecl`, which is used widely for programming in the C language. These standards differ in how and when arguments are pushed onto the system stack. The standard used by a given

function is determined when the function is compiled, and can be controlled by the programmer. LINKIMAGE can only be used with `cdecl` functions. Unfortunately, there is no way for IDL to know which convention a given function uses, meaning that LINKIMAGE will quietly accept an entry point of the wrong type. The LINKIMAGE user is responsible for ensuring that Entry is a `cdecl` function.

Keywords

DEVICE

Set this keyword to indicate that the module being loaded contains a device driver.

FUNCT

Set this keyword to indicate that the module being loaded contains a function.

KEYWORDS

Set this keyword to indicate that the procedure or function being loaded accepts keyword parameters.

MAX_ARGS

Set this keyword equal to the maximum number of non-keyword arguments the procedure or function accepts. If this keyword is not present, the maximum number of parameters is not checked when the routine is called.

Note

It is a very good idea to specify a value for MAX_ARGS. Passing the wrong number of arguments to an external routine may cause unexpected results, including causing IDL to crash. By forcing IDL to check the number of arguments before passing them to the linked routine, you will avoid parameter mismatch problems.

MIN_ARGS

Set this keyword equal to the minimum number of non-keyword arguments accepted by the procedure or function.

Obsolete Keywords

The following keywords are obsolete:

- DEFAULT

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

To add a procedure called `MY_PROC`, whose entry symbol is named `my_proc`, that is contained in the file `/home/smith/my_proc.so`:

```
LINKIMAGE, 'MY_PROC', '/home/smith/my_proc.so'
```

Version History

Introduced: Pre 4.0

See Also

[CALL_EXTERNAL](#), [SPAWN](#), and the IDL *External Development Guide*.

LL_ARC_DISTANCE

The LL_ARC_DISTANCE function returns a two-element vector containing the longitude and latitude [lon, lat] of a point given arc distance ($-\pi \leq \text{Arc_Dist} \leq \pi$), and azimuth (*Az*), from a specified location *Lon_lat0*. Values are in radians unless the keyword DEGREES is set.

This routine is written in the IDL language. Its source code can be found in the file `ll_arc_distance.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LL_ARC_DISTANCE( Lon_lat0, Arc_Dist, Az [, /DEGREES] )
```

Return Value

Returns a two-element vector containing the point longitude and latitude.

Arguments

Lon_lat0

A 2-element vector containing the longitude and latitude of the starting point. Values are assumed to be in radians unless the keyword DEGREES is set.

Arc_Dist

The arc distance from *Lon_lat0*. The value must be between $-\pi$ and $+\pi$. To express distances in arc units, divide by the radius of the globe expressed in the original units. For example, if the radius of the earth is 6371 km, divide the distance in km by 6371 to obtain the arc distance.

Az

The azimuth from *Lon_lat0*. The value is assumed to be in radians unless the keyword DEGREES is set.

Keywords

DEGREES

Set this keyword to express all measurements and results in degrees.

Examples

```
; Initial point specified in radians:  
Lon_lat0 = [1.0, 2.0]  
  
; Arc distance in radians:  
Arc_Dist = 2.0  
  
; Azimuth in radians:  
Az = 1.0  
  
Result = LL_ARC_DISTANCE(Lon_lat0, Arc_Dist, Az)  
PRINT, Result
```

IDL prints:

```
2.91415   -0.622234
```

Version History

Introduced: Pre 4.0

See Also

[MAP_SET](#)

LMFIT

The LMFIT function does a non-linear least squares fit to a function with an arbitrary number of parameters. LMFIT uses the Levenberg-Marquardt algorithm, which combines the steepest descent and inverse-Hessian function fitting methods. The function may be any non-linear function.

Iterations are performed until three consecutive iterations fail to change the chi-square value by more than the specified tolerance amount, or until a maximum number of iterations have been performed. The LMFIT function returns a vector of values for the dependent variables, as fitted by the function fit.

The initial guess of the parameter values should be as close to the actual values as possible or the solution may not converge. Test the value of the variable specified by the CONVERGENCE keyword to determine whether the algorithm converged, failed to converge, or encountered a singular matrix.

This routine is written in the IDL language. Its source code can be found in the file `lmfit.pro` in the `lib` subdirectory of the IDL distribution. LMFIT is based on the routine `mrqmin` described in section 15.5 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = LMFIT( X, Y, A [, ALPHA=variable] [, CHISQ=variable]
  [, CONVERGENCE=variable] [, COVAR=variable] [, /DOUBLE]
  [, FITA=vector] [, FUNCTION_NAME=string] [, ITER=variable]
  [, ITMAX=value] [, ITMIN=value] [, MEASURE_ERRORS=vector]
  [, SIGMA=variable] [, TOL=value] )
```

Return Value

Returns a vector of values for the dependent variable, resulting from the function fit.

Arguments

X

A row vector of independent variables. LMFIT does not manipulate or use values in *X*, it simply passes *X* to the user-written function.

Y

A row vector containing the dependent variables.

A

A vector that contains the initial estimate for each coefficient. Upon return, *A* will contain the final estimates for the coefficients.

Keywords

ALPHA

Set this keyword equal to a named variable that will contain the value of the curvature matrix.

CHISQ

Set this keyword equal to a named variable that will contain the value of the unreduced chi-square goodness-of-fit statistic.

CONVERGENCE

Set this keyword equal to a named variable that will indicate whether the LMFIT algorithm converged. The possible returned values are:

- 1 = the algorithm converged.
- 0 = the algorithm did not converge.
- -1 = the algorithm encountered a singular matrix and did not converge.

Tip

If LMFIT fails to converge, try setting the DOUBLE keyword.

COVAR

Set this keyword equal to a named variable that will contain the value of the covariance matrix.

Note

The COVAR matrix depends only upon the independent variable *X* and (optionally) the MEASURE_ERRORS. The values do not depend upon *Y*. See section 15.4 of *Numerical Recipes in C* (Second Edition) for details.

DOUBLE

Set this keyword to force the computations to be performed in double precision.

FITA

Set this keyword equal to a vector, with as many elements as A , which contains a zero for each fixed parameter, and a non-zero value for elements of A to fit. If FITA is not specified, all parameters are taken to be non-fixed.

FUNCTION_NAME

Use this keyword to specify the name of the function to fit. If this keyword is omitted, LMFIT assumes that the IDL routine LMFUNCT is to be used. If LMFUNCT is not already compiled, IDL compiles the function from the file `lmfunct.pro`, located in the `lib` subdirectory of the IDL distribution. LMFUNCT is designed to fit a quadratic equation.

The function to be fit must be written as an IDL function and compiled prior to calling LMFIT. The function must accept a vector X (the independent variables) and a vector A containing the fitted function's parameter values. It must return an $N_ELEMENTS(A)+1$ -element vector in which the first (zeroth) element is the evaluated function value and the remaining elements are the partial derivatives with respect to each parameter in A .

Note

The returned value must be of the same data type as the input X value.

ITER

Set this keyword equal to a named variable that will contain the actual number of iterations which were performed

ITMAX

Set this keyword equal to the maximum number of iterations. The default is 50.

ITMIN

Set this keyword equal to the minimum number of iterations. The default is 5.

MEASURE_ERRORS

Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y .

Note

For Gaussian errors (e.g., instrumental uncertainties), `MEASURE_ERRORS` should be set to the standard deviations of each point in Y . For Poisson or statistical weighting, `MEASURE_ERRORS` should be set to $\text{SQRT}(\text{ABS}(Y))$.

SIGMA

Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters

Note

If `MEASURE_ERRORS` is omitted, then you are assuming that your user-supplied model (or the default quadratic) is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in `SIGMA` are multiplied by $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number of coefficients. See section 15.2 of *Numerical Recipes in C* (Second Edition) for details.

TOL

Set this keyword to the convergence tolerance. The routine returns when the relative decrease in chi-squared is less than `TOL` in an iteration. The default is 1.0×10^{-6} for single-precision, and 1.0×10^{-12} for double-precision.

WEIGHTS

The `WEIGHTS` keyword is obsolete and has been replaced by the `MEASURE_ERRORS` keyword. Code that uses the `WEIGHTS` keyword will continue to work as before, but new code should use the `MEASURE_ERRORS` keyword. Note that the definition of the `MEASURE_ERRORS` keyword is not the same as the `WEIGHTS` keyword. Using the `WEIGHTS` keyword, $\text{SQRT}(1/\text{WEIGHTS}[i])$ represents the measurement error for each point $Y[i]$. Using the `MEASURE_ERRORS` keyword, the measurement error for each point is represented as simply `MEASURE_ERRORS[i]`.

Examples

In this example, we fit a function of the form:

$$f(x) = a[0] * \exp(a[1]*x) + a[2] + a[3] * \sin(x)$$

```

; First, define a return function for LMFIT:
FUNCTION myfunct, X, A
    bx = A[0]*EXP(A[1]*X)
    RETURN, [ [bx+A[2]+A[3]*SIN(X)], [EXP(A[1]*X)], [bx*X], $
            [1.0] ,[SIN(X)] ]
END

PRO lmfit_example

; Compute the fit to the function we have just defined. First,
; define the independent and dependent variables:
X = FINDGEN(40)/20.0
Y = 8.8 * EXP(-9.9 * X) + 11.11 + 4.9 * SIN(X)
measure_errors = 0.05 * Y

; Provide an initial guess for the function's parameters:
A = [10.0, -0.1, 2.0, 4.0]
fita = [1,1,1,1]

; Plot the initial data, with error bars:
PLOTERR, X, Y, measure_errors
coefs = LMFIT(X, Y, A, MEASURE_ERRORS=measure_errors, /DOUBLE, $
            FITA = fita, FUNCTION_NAME = 'myfunct')

; Overplot the fitted data:
OPLLOT, X, coefs

END

```

Version History

Introduced: 5.0

See Also

[CURVEFIT](#), [GAUSSFIT](#), [LINFIT](#), [POLY_FIT](#), [REGRESS](#), [SFIT](#), [SVDFIT](#)

LMGR

The LMGR function tests whether a particular licensing mode is in effect. Different licensing modes are specified by keyword; see the “Keywords” section below for a description of each licensing mode.

The LMGR function can also force IDL into time demo mode or report the LMHostid number for the machine in use.

For more information on IDL’s licensing methods, consult the *IDL License Management Guide*, which is included in Adobe Acrobat Portable Document Format on your IDL CD-ROM.

Syntax

```
Result = LMGR( [, /CLIENTSERVER | , /DEMO | , /EMBEDDED | , /RUNTIME | ,  
/STUDENT | , /TRIAL | , /VM] [, EXPIRE_DATE=variable] [, /FORCE_DEMO]  
[, INSTALL_NUM=variable] [, LMHOSTID=variable]  
[, SITE_NOTICE=variable])
```

Return Value

The function returns True (1) if the mode specified is in effect, or False (0) otherwise.

Arguments

None

Keywords

CLIENTSERVER

Set this keyword to test whether the current IDL session is using Client/Server licensing (as opposed to Desktop licensing).

DEMO

Set this keyword to test whether the current IDL session is running in timed demo mode. Unlicensed copies of IDL and copies running directly from a CD-ROM run in timed demo mode.

EMBEDDED

Set this keyword to test whether the current IDL session is running in embedded mode. Embedded-mode applications contain a built-in version of the IDL license. Examples of applications running in embedded mode are the IDL demo and the IDL registration program.

EXPIRE_DATE

Set this keyword to a named variable that will receive a string containing the expiration date of the current IDL session if the session is a trial session. This named variable will be undefined if the IDL session has a permanent license.

FORCE_DEMO

Set this keyword to force the current session into timed demo mode. Forcing an IDL session into demo mode can be useful if you are testing an application that will be run with an unlicensed copy of IDL. Note that you must exit IDL and restart to return to normal licensed mode after forcing IDL into demo mode.

INSTALL_NUM

Set this keyword to a named variable that will receive a string containing the installation number of the current IDL session. This named variable will be undefined if the IDL session is unlicensed.

LMHOSTID

Set this keyword equal to a named variable that will contain a string value representing the LMHostid for the machine in use. The LMHostid is used when creating client/server IDL licenses. This keyword returns the string “0” on machines which do not have a unique LMHostid (some Windows machines that use Desktop licensing.)

RUNTIME

Set this keyword to test whether the current IDL session is running in runtime mode. Runtime-mode applications do not provide access to the IDL Command Line. See [Chapter 22, “Distributing IDL Applications”](#) in the *Building IDL Applications* manual for additional details on runtime applications.

SITE_NOTICE

Set this keyword to a named variable that will receive a string containing the site notice of the current IDL session. This named variable will be undefined if the IDL session is unlicensed.

STUDENT

Set this keyword to test whether the current IDL session is running in student mode. The IDL Student version, which provides a subset of IDL's full functionality, is currently the only product that runs in student mode.

TRIAL

Set this keyword to test whether the current IDL session is running in trial mode. Trial mode licenses allow IDL to operate for a limited time period (generally 30 days) but do not otherwise restrict functionality.

VM

Set this keyword to test whether the current IDL session is running in IDL Virtual Machine mode. IDL Virtual Machine applications do not provide access to the IDL Command Line. See [“The IDL Virtual Machine”](#) in Chapter 22 of the *Building IDL Applications* manual for additional details on IDL Virtual Machine applications.

Examples

Use the following commands to test whether the current IDL session is running in timed demo mode:

```
Result = LMGR(/DEMO)
IF (Result GT 0) THEN PRINT, "IDL is in Demo Mode"
```

Use the following commands to generate the LMHostid number for the machine in use:

```
Result = LMGR(LMHOSTID = myId)
PRINT, "LMHostid for this machine is: ", myId
```

Version History

Introduced: Pre 4.0

VM keyword added: 6.0

LNGAMMA

The LNGAMMA function returns the logarithm of the gamma function of Z .

Syntax

Result = LNGAMMA(Z)

Return Value

For negative integers, LNGAMMA returns the correct value of Infinity. If Z is double-precision, the result is double-precision (either double or double complex), otherwise the result is single-precision (either float or complex).

Note

For negative nonintegers, LNGAMMA will also return Infinity. To compute the actual LNGAMMA of a negative noninteger, you should convert your input to complex first.

Arguments

Z

The expression for which the logarithm of the gamma function will be evaluated. Z may be complex.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To find the logarithm of the gamma function of 0.5 and store the result in variable A, enter:

```
A = LNGAMMA(0.5)
```

Version History

Introduced: Pre 4.0

Z argument accepts complex input: 5.6

See Also

[BETA](#), [GAMMA](#), [IBETA](#), [IGAMMA](#)

LNP_TEST

The LNP_TEST function computes the Lomb Normalized Periodogram of two sample populations X and Y and tests the hypothesis that the populations represent a significant periodic signal against the hypothesis that they represent random noise.

LNP_TEST is based on the routine `fasper` described in section 13.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = LNP_TEST( X, Y [, /DOUBLE] [, HIFAC=scale_factor] [, JMAX=variable]  
[, OFAC=value] [, WK1=variable] [, WK2=variable] )
```

Return Value

The result is a two-element vector containing the maximum peak in the Lomb Normalized Periodogram and its significance. The significance is a value in the interval [0.0, 1.0]; a small value indicates that a significant periodic signal is present.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector containing equally or unequally spaced time samples.

Y

An n -element integer, single-, or double-precision floating-point vector containing amplitudes corresponding to X_i .

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

HIFAC

Use this keyword to specify the scale factor of the average Nyquist frequency. The default value is 1.

JMAX

Use this keyword to specify a named variable that will contain the index of the maximum peak in the Lomb Normalized Periodogram.

OFAC

Use this keyword to specify the oversampling factor. The default value is 4.

WK1

Use this keyword to specify a named variable that will contain a vector of increasing linear frequencies.

WK2

Use this keyword to specify a named variable that will contain a vector of values from the Lomb Normalized Periodogram corresponding to the frequencies in WK1.

Examples

This example tests the hypothesis that two sample, n -element populations X and Y represent a significant periodic signal against the hypothesis that they represent random noise:

```

; Define two n-element sample populations:
X = [ 1.0,  2.0,  5.0,  7.0,  8.0,  9.0, $
      10.0, 11.0, 12.0, 13.0, 14.0, 15.0, $
      16.0, 17.0, 18.0, 19.0, 20.0, 22.0, $
      23.0, 24.0, 25.0, 26.0, 27.0, 28.0]
Y = [ 0.69502, -0.70425,  0.20632,  0.77206, -2.08339,  0.97806, $
      1.77324,  2.34086,  0.91354,  2.04189,  0.53560, -2.05348, $
      -0.76308, -0.84501, -0.06507, -0.12260,  1.83075,  1.41403, $
      -0.26438, -0.48142, -0.50929,  0.01942, -1.29268,  0.29697]

; Test the hypothesis that X and Y represent a significant periodic
; signal against the hypothesis that they represent random noise:
result = LNP_TEST(X, Y, WK1 = wk1, WK2 = wk2, JMAX = jmax)
PRINT, result

```

IDL prints:

```
4.69296    0.198157
```

The small value of the significance represents the possibility of a significant periodic signal. A larger number of samples for X and Y would produce a more conclusive result. `WK1` and `WK2` are both 48-element vectors containing linear frequencies and corresponding Lomb values, respectively. `JMAX` is the indexed location of the maximum Lomb value in `WK2`.

Version History

Introduced: 4.0

See Also

[CTI_TEST](#), [FV_TEST](#), [KW_TEST](#), [MD_TEST](#), [R_TEST](#), [RS_TEST](#), [S_TEST](#),
[TM_TEST](#), [XSQ_TEST](#)

LOADCT

The LOADCT procedure loads one of 41 predefined IDL color tables. These color tables are defined in the file `colors1.tbl`, located in the `\resource\colors` subdirectory of the main IDL directory, unless the FILE keyword is specified. The selected colortable is loaded into the COLORS common block as both the “current” and “original” colortable. If the current device has fewer than 256 colors, the color table data is interpolated to cover the number of colors in the device.

This routine is written in the IDL language. Its source code can be found in the file `loadct.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
LOADCT [, Table] [, BOTTOM=value] [, FILE=string] [, GET_NAMES=variable]
      [, NCOLORS=value] [, /SILENT]
```

Arguments

Table

The number of the pre-defined color table to load, from 0 to 40. If this value is omitted, a menu of the available tables is printed and the user is prompted to enter a table number.

Keywords

BOTTOM

The first color index to use. LOADCT will use color indices from BOTTOM to BOTTOM+NCOLORS-1. The default is BOTTOM=0.

FILE

Set this keyword to the name of a colortable file to be used instead of the file `colors1.tbl`. See [MODIFYCT](#) to create and modify colortable files.

GET_NAMES

Set this keyword to a named variable in which the names of the color tables are returned as a string array. No changes are made to the color table.

NCOLORS

The number of colors to use. The default is all available colors (this number is stored in the system variable !D.TABLE_SIZE).

SILENT

If this keyword is set, the Color Table message is suppressed.

Version History

Introduced: Original

See Also

[MODIFYCT](#), [XLOADCT](#), [TVLCT](#)

LOCALE_GET

The LOCALE_GET function returns the current locale (string) of the operating platform.

Syntax

Result = LOCALE_GET()

Return Value

Returns a string containing the current operating platform locale.

Arguments

None

Keywords

None

Version History

Introduced: 5.2.1

LOGICAL_AND

The LOGICAL_AND function performs a logical AND operation on its arguments. It returns True (1) if both of its arguments are non-zero (non-NULL for strings and heap variables), or False (0) otherwise.

The LOGICAL_AND function is similar to the AND operator, except that it performs a logical “and” rather than a bitwise “and” on its arguments.

Note

LOGICAL_AND always returns either 0 or 1, unlike the AND operator, which performs a bitwise AND operation on integers, and returns one of the two arguments for other types.

Unlike the && operator, LOGICAL_AND accepts multi-element arrays as its arguments. In addition, where the && operator *short-circuits* if it can determine the result by evaluating only the first argument, all arguments to a function are always evaluated.

Syntax

Result = LOGICAL_AND(*Arg1*, *Arg2*)

Return Value

Integer zero (false) or one (true) if both arguments are scalars, or an array of zeroes and ones if either argument is an array.

Arguments

Arg1, Arg2

The expressions on which the logical AND operation is to be carried out. The arguments can be scalars or arrays of any type other than structure.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU

system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See Appendix D, “Thread Pool Keywords” for details.

Example

At the IDL Command line, enter:

```
PRINT, LOGICAL_AND(2,4), LOGICAL_AND(2,0), LOGICAL_AND(0,4), $  
      LOGICAL_AND(0,0)
```

IDL Prints:

```
1 0 0 0
```

Version History

Introduced: 6.0

See Also

“[Logical Operators](#)” in the *Building IDL Applications* manual, “[Bitwise Operators](#)” in the *Building IDL Applications* manual, [LOGICAL_OR](#), [LOGICAL_TRUE](#)

LOGICAL_OR

The LOGICAL_OR function performs a logical OR operation on its arguments. It returns True (1) if either of its arguments are non-zero (non-NULL for strings and heap variables), and False (0) otherwise.

The LOGICAL_OR function is similar to the OR operator, except that it performs a logical “or” rather than a bitwise “or” on its arguments.

Note

LOGICAL_OR always returns either 0 or 1, unlike the OR operator, which performs a bitwise OR operation on integers, and returns one of the two arguments for other types.

Unlike the || operator, LOGICAL_OR accepts multi-element arrays as its arguments. In addition, where the || operator *short-circuits* if it can determine the result by evaluating only the first argument, all arguments to a function are always evaluated.

Syntax

$$Result = LOGICAL_OR(Arg1, Arg2)$$

Return Value

Integer zero (false) or one (true) if both operands are scalars, or an array of zeroes and ones if either operand is an array.

Arguments

Arg1, Arg2

The expressions on which the logical OR operation is to be carried out. The arguments can be scalars or arrays of any type other than structure.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU

system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See Appendix D, “Thread Pool Keywords” for details.

Example

At the IDL Command Line, enter:

```
PRINT, LOGICAL_OR(2,4), LOGICAL_OR(2,0), LOGICAL_OR(0,4), $  
      LOGICAL_OR(0,0)
```

IDL Prints:

```
1 1 1 0
```

Version History

Introduced: 6.0

See Also

“[Logical Operators](#)” in the *Building IDL Applications* manual, “[Bitwise Operators](#)” in the *Building IDL Applications* manual, [LOGICAL_AND](#), [LOGICAL_TRUE](#)

LOGICAL_TRUE

The LOGICAL_TRUE function returns True (1) if its arguments are non-zero (non-NULL for strings and heap variables), and False (0) otherwise.

Note

For a given argument, the value returned by LOGICAL_TRUE is the opposite of the value returned by the ~ operator.

Syntax

Result = LOGICAL_TRUE(*Arg*)

Return Value

Integer zero (false) or one (true) if the argument is a scalar, or an array of zeroes and ones if the argument is an array.

Arguments

Arg

The expression on which the logical truth evaluation is to be carried out. The argument can be a scalar or an array of any type other than structure.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See Appendix D, "Thread Pool Keywords" for details.

Example

At the IDL Command Line, enter:

```
PRINT, LOGICAL_TRUE(2), LOGICAL_TRUE(0)
```

IDL Prints:

```
1 0
```

Version History

Introduced: 6.0

See Also

“[Logical Operators](#)” in the *Building IDL Applications* manual, “[Bitwise Operators](#)” in the *Building IDL Applications* manual, [KEYWORD_SET](#), [LOGICAL_AND](#), [LOGICAL_OR](#)

LON64ARR

The LON64ARR function returns a 64-bit integer vector or array.

Syntax

$$\text{Result} = \text{LON64ARR}(D_1 [, \dots, D_8] [, /\text{NOZERO}])$$

Return Value

Returns a 64-bit array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

NOZERO

Normally, LON64ARR sets every element of the result to zero. If NOZERO is set, this zeroing is not performed and LON64ARR executes faster.

Examples

To create L, a 100-element, 64-bit vector with each element set to 0, enter:

```
L = LON64ARR(100)
```

Version History

Introduced: 5.2

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#),
[LONARR](#), [MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

LONARR

The LONARR function returns a longword integer vector or array.

Syntax

```
Result = LONARR(  $D_1$  [, ...,  $D_8$ ] [, /NOZERO] )
```

Return Value

Returns a long array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

NOZERO

Normally, LONARR sets every element of the result to zero. If NOZERO is set, this zeroing is not performed and LONARR executes faster.

Examples

To create L, a 100-element, longword vector with each element set to 0, enter:

```
L = LONARR(100)
```

Version History

Introduced: Original

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#),
[LON64ARR](#), [MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

LONG

The LONG function returns a result equal to *Expression* converted to longword integer type.

Syntax

$$\text{Result} = \text{LONG}(\text{Expression}[, \text{Offset} [, D_1 [, \dots, D_8]]])$$

Return Value

Returns the conversion of the given scalar or array to a longword integer type.

Arguments

Expression

The expression to be converted to longword integer.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as longword integer data.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid floating-point value and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

If A contains the floating-point value 32000.0, it can be converted to a longword integer and stored in the variable B by entering:

```
B = LONG(A)
```

Version History

Introduced: Original

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

LONG64

The LONG64 function returns a result equal to *Expression* converted to 64-bit integer type.

Syntax

$$\text{Result} = \text{LONG64}(\text{Expression}[, \text{Offset} [, D_1 [, \dots, D_8]]])$$

Return Value

Returns the conversion of the given scalar or array to a 64-bit integer type.

Arguments

Expression

The expression to be converted to 64-bit integer.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as 64-bit integer data.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid floating-point value and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

If A contains the floating-point value 32000.0, it can be converted to a 64-bit integer and stored in the variable B by entering:

```
B = LONG64(A)
```

Version History

Introduced: 5.2

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#), [STRING](#), [UINT](#), [ULONG](#), [ULONG64](#)

LSODE

The LSODE function uses adaptive numerical methods to advance a solution to a system of ordinary differential equations one time-step H , given values for the variables Y and X .

Syntax

```
Result = LSODE( Y, X, H, Derivs[, Status] [, ATOL=value] [, /QUIET]  
[, RTOL=value] )
```

Return Value

Returns the solution in a vector with the same number of elements as Y .

Arguments

Y

A vector of values for Y at X

X

A scalar value for the initial condition.

H

A scalar value giving interval length or step size.

Derivs

A scalar string specifying the name of a user-supplied IDL function that calculates the values of the derivatives $Dydx$ at X . This function must accept two arguments: A scalar floating value X , and an n -element vector Y . It must return an n -element vector result.

For example, suppose the values of the derivatives are defined by the following relations:

$$dy_0/dx = -0.5y_0, \quad dy_1/dx = 4.0 - 0.3y_1 - 0.1y_0$$

We can write a function called `differential` to express these relationships in the IDL language:

```
FUNCTION differential, X, Y
  RETURN, [-0.5 * Y[0], 4.0 - 0.3 * Y[1] - 0.1 * Y[0]]
END
```

Status

An index used for input and output to specify the state of the calculation. This argument contains a positive value if the function was successfully completed. Negative values indicate different errors.

Input Value	Description
1	This is the first call for the problem; initializations will occur. This is the default value.
2	This is not the first call. The calculation is to continue normally.
3	This is not the first call. The calculation is to continue normally, but with a change in input parameters.

Table 3-72: Input Values for Status

Note

A preliminary call with `tout = t` is not counted as a first call here, as no initialization or checking of input is done. (Such a call is sometimes useful for the purpose of outputting the initial condition *s*.) Thus, the first call for which `tout ≠ t` requires `STATUS = 1` on input.

Output Value	Description
1	Nothing occurred. (However, an internal counter was set to detect and prevent repeated calls of this type.)
2	The integration was performed successfully, and no roots were found.
3	The integration was successful, and one or more roots were found.

Table 3-73: Output Values for Status

Output Value	Description
-1	An excessive amount of work was done on this call, but the integration was otherwise successful. To continue, reset STATUS to a value greater than 1 and begin again (the excess work step counter will be reset to 0).
-2	The precision of the machine being used is insufficient for the requested amount of accuracy. Integration was successful. To continue, the tolerance parameters must be reset, and STATUS must be set to 3. (If this condition is detected before taking any steps, then an illegal input return (STATUS = -3) occurs instead.)
-3	Illegal input was detected, before processing any integration steps. If the solver detects an infinite loop of calls to the solver with illegal input, it will cause the run to stop.
-4	There were repeated error test failures on one attempted step, before completing the requested task, but the integration was successful. The problem may have a singularity, or the input may be inappropriate.
-5	There were repeated convergence test failures on one attempted step, before completing the requested task, but the integration was successful. This may be caused by an inaccurate jacobian matrix, if one is being used.
-6	ewt(<i>i</i>) became zero for some <i>i</i> during the integration. Pure relative error control was requested on a variable which has now vanished. Integration was successful.

Table 3-73: Output Values for Status (Continued)

Note

Since the normal output value of STATUS is 2, it does not need to be reset for normal continuation. Also, since a negative input value of STATUS will be regarded as illegal, a negative output value requires the user to change it, and possibly other inputs, before calling the solver again.

Keywords

ATOL

A scalar or array value that specifies the absolute tolerance. The default value is 1.0e-7. Use $ATOL = 0.0$ (or $ATOL[i] = 0.0$) for pure relative error control, and use $RTOL = 0.0$ for pure absolute error control. For an explanation of how to use ATOL and RTOL together, see RTOL below.

QUIET

Set this keyword to suppress warning messages. By default, warning messages will be output to the screen.

RTOL

A scalar value that specifies the relative tolerance. The default value is 1.0e-7. Use $RTOL = 0.0$ for pure absolute error control, and use $ATOL = 0.0$ (or $ATOL[i] = 0.0$) for pure relative error control.

The estimated local error in the $Y[i]$ argument will be controlled to be less than

```
ewt[i] = RTOL*abs(Y[i]) + ATOL    ; If ATOL is a scalar.
ewt[i] = RTOL*abs(Y[i]) + ATOL[i] ; If ATOL is an array.
```

Thus, the local error test passes if, in each component, either the absolute error is less than ATOL (or $ATOL[i]$), or if the relative error is less than RTOL.

Warning

Actual, or global, errors might exceed these local tolerances, so choose values for ATOL and RTOL conservatively.

Examples

To integrate the example system of differential equations for one time step, H:

```
PRO LSODETEST

    ; Define the step size:
    H = 0.5

    ; Define an initial X value:
    X = 0.0

    ; Define initial Y values:
    Y = [4.0, 6.0]
```

```

; Integrate over the interval (0, 0.5):
result = LSODE(Y, X, H, 'differential')

; Print the result:
PRINT, result

```

```
END
```

```

FUNCTION differential, X, Y
  RETURN, [-0.5 * Y[0], 4.0 - 0.3 * Y[1] - 0.1 * Y[0]]
END

```

IDL prints:

```
3.11523      6.85767
```

This is the exact solution vector to 5-decimal precision.

See Also

[DERIV](#), [DERIVSIG](#), [RK4](#)

References

1. Alan C. Hindmarsh, ODEPACK, A Systematized Collection of ODE Solvers, in *Scientific Computing*, R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, 1983, pp. 55-64.
2. Linda R. Petzold, Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations, *SIAM J. SCI. STAT. COMPUT.* 4 (1983), pp. 136-148.
3. Kathie L. Hiebert and Lawrence F. Shampine, Implicitly Defined Output Points for Solutions of ODE's, Sandia Report SAND80-0180, February, 1980.

Version History

Introduced: 5.1

QUIET keyword added: 6.1

LU_COMPLEX

The LU_COMPLEX function solves an n by n complex linear system $\mathbf{Az} = \mathbf{b}$ using LU decomposition. The result is an n -element complex vector z . Alternatively, LU_COMPLEX computes the generalized inverse of an n by n complex array.

This routine is written in the IDL language. Its source code can be found in the file `lu_complex.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = LU_COMPLEX( A, B [, /DOUBLE] [, /INVERSE] [, /SPARSE] )
```

Return Value

The result is an n by n complex array.

Arguments

A

An n by n complex array.

B

An n -element right-hand side vector (real or complex).

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

INVERSE

Set this keyword to compute the generalized inverse of A . If INVERSE is specified, the input argument B is ignored.

SPARSE

Set this keyword to convert the input array to row-indexed sparse storage format. Computations are done using the iterative biconjugate gradient method. This

keyword is effective only when solving complex linear systems. This keyword has no effect when calculating the generalized inverse.

Examples

```

; Define a complex array A and right-side vector B:
A = [[COMPLEX(1, 0), COMPLEX(2,-2), COMPLEX(-3,1)], $
      [COMPLEX(1,-2), COMPLEX(2, 2), COMPLEX(1, 0)], $
      [COMPLEX(1, 1), COMPLEX(0, 1), COMPLEX(1, 5)]]
B = [COMPLEX(1, 1), COMPLEX(3,-2), COMPLEX(1,-2)]

; Solve the complex linear system Az = b:
Z = LU_COMPLEX(A, B)
PRINT, 'Z:'
PRINT, Z

; Compute the inverse of the complex array A by supplying a scalar
; for B (in this example -1):
inv = LU_COMPLEX(A, B, /INVERSE)
PRINT, 'Inverse:'
PRINT, inv

```

IDL prints:

```

Z:
(   0.552267,   1.22818)(  -0.290371,  -0.600974)
(  -0.629824,  -0.340952)

Inverse:
(   0.261521,  -0.0303485)(   0.0138629,   0.329337)
(  -0.102660,  -0.168602)
(   0.102660,   0.168602)(   0.0340952,  -0.162982)
(   0.125890,  -0.0633196)
(  -0.0689397,   0.0108655)(  -0.0666916,  -0.0438366)
(   0.0614462,  -0.161858)

```

Version History

Introduced: Pre 4.0

See Also

[CRAMER](#), [CHOLSOL](#), [GS_ITER](#), [LUSOL](#), [SVSOL](#), [TRISOL](#), and “Sparse Arrays” in Chapter 22 of the *Using IDL* manual.

LUDC

The LUDC procedure replaces an n by n array, A , with the LU decomposition of a row-wise permutation of itself.

LUDC is based on the routine `ludcmp` described in section 2.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, use the `LA_LUDC` procedure instead.

Syntax

LUDC, A , $Index$ [, /COLUMN] [, /DOUBLE] [, INTERCHANGES=*variable*]

Arguments

A

An n by n array of any type except string. Upon output, A is replaced with its LU decomposition.

Index

An output vector that records the row permutations which occurred as a result of partial pivoting.

Keywords

COLUMN

Set this keyword if the input array A is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

INTERCHANGES

An output variable that is set to positive 1 if the number of row interchanges was even, or to negative 1 if the number of interchanges was odd.

Examples

See the description of [LUSOL](#) for an example using this procedure.

Version History

Introduced: 4.0

See Also

[LA_LUDC](#), [LUSOL](#)

LUMPROVE

The LUMPROVE function uses LU decomposition to iteratively improve an approximate solution X of a set of n linear equations in n unknowns $Ax = b$.

LUMPROVE is based on the routine `improve` described in section 2.5 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, use the `LA_LUMPROVE` function instead.

Syntax

Result = LUMPROVE(*A*, *Alud*, *Index*, *B*, *X* [, /COLUMN] [, /DOUBLE])

Return Value

The result is a vector, whose type and length are identical to X , containing the improved solution.

Arguments

A

The n by n coefficient array of the linear system $Ax = b$.

Alud

The n by n LU decomposition of A created by the LUDC procedure.

Index

An input vector, created by the LUDC procedure, containing a record of the row permutations which occurred as a result of partial pivoting.

B

An n -element vector containing the right-hand side of the linear system $Ax = b$.

X

An n -element vector containing the approximate solution of the linear system $Ax = b$.

Keywords

COLUMN

Set this keyword if the input array A is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

This example uses LUMPROVE to improve an approximate solution X to the linear system $Ax = B$:

```

; Create coefficient array A:
A = [[ 2.0,  1.0,  1.0], $
      [ 4.0, -6.0,  0.0], $
      [-2.0,  7.0,  2.0]]

; Create a duplicate of A:
alud = A
; Define the right-hand side vector B:
B = [3.0, -8.0, 10.0]

; Begin with an estimated solution X:
X = [.89, 1.78, -0.88]

; Decompose the duplicate of A:
LUDC, alud, INDEX

; Compute an improved solution:
result = LUMPROVE(A, alud, INDEX, B, X)

; Print the result:
PRINT, result

```

IDL prints:

```
1.00000 2.00000 -1.00000
```

This is the exact solution vector.

Version History

Introduced: 4.0

See Also

[GS_ITER](#), [LA_LUMPROVE](#), [LUDC](#)

LUSOL

The LUSOL function is used in conjunction with the LUDC procedure to solve a set of n linear equations in n unknowns $Ax = b$. The parameter A is input not as the original array, but as its LU decomposition, created by the routine LUDC.

LUSOL is based on the routine `lubksb` described in section 2.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, use the LA_LUSOL function instead.

Syntax

Result = LUSOL(*A*, *Index*, *B* [, /COLUMN] [, /DOUBLE])

Return Value

The result is an n -element vector whose type is identical to A .

Arguments

A

The n by n LU decomposition of an array created by the LUDC procedure.

Index

An input vector, created by the LUDC procedure, containing a record of the row permutations which occurred as a result of partial pivoting.

B

An n -element vector containing the right-hand side of the linear system $Ax = b$.

Keywords

COLUMN

Set this keyword if the input array A is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

This example solves the linear system $Ax = b$ using LU decomposition and back substitution:

```

; Define array A:
A = [[ 2.0,  1.0,  1.0], $
      [ 4.0, -6.0,  0.0], $
      [-2.0,  7.0,  2.0]]

; Define right-hand side vector B:
B = [3.0, -8.0, 10.0]

; Decompose A:
LUDC, A, INDEX

; Compute the solution using back substitution:
result = LUSOL(A, INDEX, B)

; Print the result:
PRINT, result

```

IDL prints:

```
1.00000  2.00000  -1.00000
```

This is the exact solution vector.

Version History

Introduced: Pre 4.0

See Also

[CHOLSOL](#), [CRAMER](#), [GS_ITER](#), [LA_LUSOL](#), [LU_COMPLEX](#), [LUDC](#), [SVSOL](#),
[TRISOL](#)

M_CORRELATE

The M_CORRELATE function computes the multiple correlation coefficient of a dependent variable and two or more independent variables.

This routine is written in the IDL language. Its source code can be found in the file `m_correlate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = M_CORRELATE( X, Y [, /DOUBLE] )
```

Return Value

Returns the single or double-precision multiple correlation coefficient.

Arguments

X

An integer, single-, or double-precision floating-point array of m -columns and n -rows that specifies the independent variable data. The columns of this two dimensional array correspond to the n -element vectors of independent variable data.

Y

An n -element integer, single-, or double-precision floating-point vector that specifies the dependent variable data.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Example

```
PRO M_CORRELATE_TEST

; Define the independent (X) and dependent (Y) data:
X = [[0.477121, 2.0, 13.0], $
     [0.477121, 5.0, 6.0], $
     [0.301030, 5.0, 9.0], $
```

```

[0.000000, 7.0, 5.5], $
[0.602060, 3.0, 7.0], $
[0.698970, 2.0, 9.5], $
[0.301030, 2.0, 17.0], $
[0.477121, 5.0, 12.5], $
[0.698970, 2.0, 13.5], $
[0.000000, 3.0, 12.5], $
[0.602060, 4.0, 13.0], $
[0.301030, 6.0, 7.5], $
[0.301030, 2.0, 7.5], $
[0.698970, 3.0, 12.0], $
[0.000000, 4.0, 14.0], $
[0.698970, 6.0, 11.5], $
[0.301030, 2.0, 15.0], $
[0.602060, 6.0, 8.5], $
[0.477121, 7.0, 14.5], $
[0.000000, 5.0, 9.5]]
Y = [97.682, 98.424, 101.435, 102.266, 97.067, 97.397, $
99.481, 99.613, 96.901, 100.152, 98.797, 100.796, $
98.750, 97.991, 100.007, 98.615, 100.225, 98.388, $
98.937, 100.617]

; Compute the multiple correlation of Y on the first column of
; X. The result should be 0.798816.
PRINT, 'Multiple correlation of Y on 1st column of X:'
PRINT, M_CORRELATE(X[0,*], Y)

; Compute the multiple correlation of Y on the first two columns
; of X. The result should be 0.875872.
PRINT, 'Multiple correlation of Y on 1st two columns of X:'
PRINT, M_CORRELATE(X[0:1,*], Y)

; Compute the multiple correlation of Y on all columns of X. The
; result should be 0.877197.
PRINT, 'Multiple correlation of Y on all columns of X:'
PRINT, M_CORRELATE(X, Y)

END

```

IDL prints:

```

Multiple correlation of Y on 1st column of X:
0.798816
Multiple correlation of Y on 1st two columns of X:
0.875872
Multiple correlation of Y on all columns of X:
0.877196

```

Version History

Introduced: 4.0

See Also

[A_CORRELATE](#), [CORRELATE](#), [C_CORRELATE](#), [P_CORRELATE](#),
[R_CORRELATE](#)

MACHAR

The MACHAR function determines and returns machine-specific parameters affecting floating-point arithmetic. Information is returned in the form of a structure with the fields listed in the “Return Value” section.

MACHAR is based on the routine `machar` described in section 20.1 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission. See that section for more details on and sample values of the various parameters returned.

Syntax

Result = MACHAR([, /DOUBLE])

Return Value

The following table lists the fields in the structure returned from the MACHAR function:

Field Name	Description
IBETA	The radix in which numbers are represented. A longword integer.
IT	The number of base-IBETA digits in the floating-point mantissa M. A longword integer.
IRND	A code in the range 0 – 5 giving information on what type of rounding is done and how underflow is handled. A longword integer.
NGRD	The number of “guard digits” used when truncating the product of two mantissas. A longword integer.
MACHEP	The exponent of the smallest power of IBETA that, added to 1.0, gives something different from 1.0. A longword integer.
NEGEP	The exponent of the smallest power of IBETA that, subtracted from 1.0, gives something different from 1.0. A longword integer.

Table 3-74: MACHAR Fields

Field Name	Description
IEXP	The number of bits in the exponent. A longword integer.
MINEXP	The smallest value of IBETA consistent with there being no leading zeros in the mantissa. A longword integer.
MAXEXP	The smallest positive value of IBETA that causes overflow. A longword integer.
EPS	The floating-point number $IBETA^{MACHEP}$, loosely referred to as the “floating-point precision.”
EPSNEG	The floating-point number $IBETA^{NEGEP}$, which is another way of determining floating-point precision.
XMIN	The floating-point number $IBETA^{MINEXP}$, generally the magnitude of the smallest usable floating-point value.
XMAX	The largest usable floating-point value, defined as the number $(1-EPSNEG) \times IBETA^{MAXEXP}$.

Table 3-74: MACHAR Fields

Arguments

None

Keywords

DOUBLE

The information returned is normally for single-precision floating-point arithmetic. Specify DOUBLE to see double-precision information.

Version History

Introduced: 4.0

See Also

[CHECK_MATH](#), “!VALUES” on page 4357, and “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual.

MAKE_ARRAY

The MAKE_ARRAY function enables you to dynamically create an array whose characteristics are not known until run time.

Syntax

```
Result = MAKE_ARRAY ( [D1 [, ..., D8]] [, /BYTE | /COMPLEX | /DCOMPLEX
| /DOUBLE | /FLOAT | /INTEGER | /L64 | /LONG | /OBJ | /PTR |
/STRING | /UINT | /UL64 | /ULONG] [, DIMENSION=vector] [, /INDEX]
[, /NOZERO] [, SIZE=vector] [, TYPE=type_code] [, VALUE=value] )
```

Return Value

Returns an array of the specified type, dimensions, and initialization.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

The D_i arguments are optional if the dimensions of the result are specified using the DIMENSION keyword. Note that you should supply either the D_i arguments or a value for the DIMENSION keyword, but not both.

Keywords

BYTE

Set this keyword to create a byte array.

COMPLEX

Set this keyword to create a complex, single-precision, floating-point array.

DCOMPLEX

Set this keyword to create a complex, double-precision, floating-point array.

DIMENSION

An array of up to eight scalar elements, specifying the dimensions of the result. Note that you should supply either the D_i arguments or a value for the DIMENSION keyword, but not both.

DOUBLE

Set this keyword to create a double-precision, floating-point array.

FLOAT

Set this keyword to create a single-precision, floating-point array.

L64

Set this keyword to create a 64-bit integer array.

INDEX

Set this keyword to initialize the array with each element set to the value of its one-dimensional subscript.

INTEGER

Set this keyword to create an integer array.

LONG

Set this keyword to create a longword integer array.

NOZERO

Set this keyword to prevent the initialization of the array. Normally, each element of the resulting array is set to zero.

OBJ

Set this keyword to create an object reference array.

PTR

Set this keyword to create a pointer array.

SIZE

A size vector specifying the type and dimensions of the result. The format of a size vector is given in the description of the `SIZE` function.

STRING

Set this keyword to create a string array.

TYPE

The type code to set the type of the result. See the description of the `SIZE` function for a list of IDL type codes.

UINT

Set this keyword to create an unsigned integer array.

UL64

Set this keyword to create an unsigned 64-bit integer array.

ULONG

Set this keyword to create an unsigned longword integer array.

VALUE

The value to initialize each element of the resulting array. `VALUE` can be a scalar of any type including structure types. The result type is taken from `VALUE` unless one of the other keywords that specify a type is also set. In that case, `VALUE` is converted to the type specified by the other keyword prior to initializing the resulting array.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the `!CPU` system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create a 3-element by 4-element integer array with each element set to the value 5, enter:

```
M = MAKE_ARRAY(3, 4, /INTEGER, VALUE = 5)
```

Version History

Introduced: 4.0

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#), [LON64ARR](#), [LONARR](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

MAKE_DLL

The MAKE_DLL procedure builds a sharable library from C language code which is suitable for use by IDL's dynamic linking features such as CALL_EXTERNAL, LINKIMAGE, and dynamically loadable modules (DLMs). MAKE_DLL reduces the complexity of building sharable libraries by providing a stable cross-platform method for the user to describe the desired library, and issuing the necessary operating system commands to build the library.

Although MAKE_DLL is very convenient, it is not intended for use as a general purpose compiler. Instead, MAKE_DLL is specifically targeted to solving the most common IDL dynamic linking problem: building a sharable library from C language source files that are usable by IDL. Because of this, the following requirements apply:

- You must have a C compiler installed on your system. It is easiest to use the compiler used to build IDL, because MAKE_DLL already knows how to use that compiler without any additional configuring. To determine which compiler was used, query the !MAKE_DLL system variable with a print statement such as the following:


```
PRINT, !MAKE_DLL.COMPILER_NAME
```
- MAKE_DLL only compiles programs written in the C language; it does not understand Fortran, C++, or any other languages.
- MAKE_DLL provides only the functionality necessary to build C code intended to be linked with IDL. Not every possible option supported by the C compiler or system linker is addressed, only those commonly needed by IDL-related C code.

MAKE_DLL solves the most common IDL-centric problem of linking C code with IDL. To do more than this or to use a different language requires a system-specific building process (*e.g.* make files or projects).

Syntax

```
MAKE_DLL, InputFiles [, OutputFile], ExportedRoutineNames [, CC=string]
  [, COMPILE_DIRECTORY=path] [, DLL_PATH=variable]
  [, EXPORTED_DATA=string] [, EXTRA_CFLAGS=string]
  [, EXTRA_LFLAGS=string] [, INPUT_DIRECTORY=path] [, LD=string]
  [, /NOCLEANUP] [, OUTPUT_DIRECTORY=path] [, /REUSE_EXISTING]
  [, /SHOW_ALL_OUTPUT] [, /VERBOSE]
```

Arguments

InputFiles

A string (scalar or array) giving the names of the input C program files to be compiled by MAKE_DLL. These names should not include any directory path information or the `.c` suffix, they are simply the base file names.

The input directory is specified using the INPUT_DIRECTORY keyword, and the `.c` file suffix is assumed.

OutputFile

The base name of the resulting sharable library. This name should not include any directory path information or the sharable library suffix, which differs between platforms (for example: `.so`, `.a`, `.sl`, `.exe`, `.dll`).

The output directory can be specified using the OUTPUT_DIRECTORY keyword.

If the *OutputFile* argument is omitted, the first name given by *InputFile* is used as the base name of output file.

ExportedRoutineNames

A string (scalar or array) specifying the names of the routines to be exported (i.e., that are visible for linking) from the resulting sharable library.

Keywords

CC

If present, a template string to use in generating the C compiler commands to compile *InputFiles*. If CC is not specified, the value given by the !MAKE_DLL.CC system variable is used by default. See the discussion of !MAKE_DLL for a description of how to write the format string for CC.

COMPILE_DIRECTORY

To build a sharable library, MAKE_DLL requires a place to create the necessary intermediate files and possibly the final library itself. If COMPILE_DIRECTORY is specified, the directory specified is used. If COMPILE_DIRECTORY is not specified, the directory given by the !MAKE_DLL.COMPILE_DIRECTORY system variable is used.

DLL_PATH

If present, the name of a variable to receive the complete file path for the newly created sharable library. The location of the resulting sharable library depends on the setting of the `OUTPUT_DIRECTORY` or `COMPILE_DIRECTORY` keywords as well as the `!MAKE_DLL.COMPILE_DIRECTORY` system variable, and different platforms use different file suffixes to indicate sharable libraries. Use of the `DLL_PATH` keyword makes it possible to determine the resulting file path in a simple and portable manner.

EXPORTED_DATA

A string (scalar or array) containing the names of variables to be exported (i.e., are visible for linking) from the resulting sharable library.

EXTRA_CFLAGS

If present, a string supplying extra options for the command used to execute the C compiler to compile the files given by *InputFiles*. This keyword is frequently used to specify header file include directories. This text is inserted in place of the `%X` format code in the compile string. See the discussion of the `CC` keyword and `!MAKE_DLL.CC` system variable for more information.

EXTRA_LFLAGS

If present, a string supplying extra options for the command used to execute the linker when combining the object files to produce the sharable library. This keyword is frequently used to specify libraries to be included in the link, and is inserted in place of the `%X` format code in the linker string. See the discussion of the `LD` keyword and `!MAKE_DLL.LD` system variable for more information.

INPUT_DIRECTORY

If present, the path to the directory containing the source C files listed in *InputFiles*. If `INPUT_DIRECTORY` is not specified, the directory given by `COMPILE_DIRECTORY` is assumed to contain the files.

LD

If present, a template string to use when generating the linker command to generate the resulting sharable library. If `LD` is not specified, the value given by the `!MAKE_DLL.LD` system variable is used by default. See the discussion of `!MAKE_DLL` for a description of how to write the format string for `LD`.

NOCLEANUP

To produce a sharable library, MAKE_DLL produces several intermediate files:

1. A shell script (UNIX) or batch file (Windows) that is then executed via SPAWN to build the library.
2. A linker options file. This file is used to control the linker. MAKE_DLL uses it to cause the routines given by the *ExportedRoutineNames* argument (and EXPORTED_DATA keyword) to be exported from the resulting sharable library. The general platform terminology is shown below.

Platform	Linker Options File Terminology
UNIX	export file, or linker map file
Windows	a .DEF file

Table 3-75: Platform Terminology for Linker Options File

3. Object files, resulting from compiling the source C files given by the *InputFiles* argument.
4. A log file that captures the output from executing the script, and which can be used for debugging in case of error.

By default, MAKE_DLL deletes all of these intermediate files once the sharable library has been successfully built. Setting the NOCLEANUP keyword prevents MAKE_DLL from removing them.

Note

Set the NOCLEANUP keyword (possibly in conjunction with VERBOSE) for troubleshooting, or to read the files for additional information on how MAKE_DLL works.

OUTPUT_DIRECTORY

By default, MAKE_DLL creates the resulting sharable library in the compile directory specified by the COMPILE_DIRECTORY keyword or the !MAKE_DLL.COMPILE_DIRECTORY system variable. The OUTPUT_DIRECTORY keyword can be used to override this and explicitly specify where the library file should go.

REUSE_EXISTING

If this keyword is set, and the sharable library file specified by *OutputFile* already exists, MAKE_DLL returns without building the sharable library again. Use this keyword in situations where you wish to ensure that a library exists, but only want to build it if it does not. Combining the REUSE_EXISTING and DLL_PATH keywords allows you to get a path to the library in a platform independent manner, building the library only if necessary.

SHOW_ALL_OUTPUT

MAKE_DLL normally produces no output unless an error prevents successful building of the sharable library. Set SHOW_ALL_OUTPUT to see all output produced by the spawned process building the library.

VERBOSE

If set, VERBOSE causes MAKE_DLL to issue informational messages as it carries out the task of building the sharable library. These messages include information on the intermediate files created to build the library and how they are used.

Obsolete Keywords

The following keywords are obsolete:

- VAX_FLOAT

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

Example 1: Testmodule DLM

The IDL distribution contains an example of a simple DLM (dynamically loadable module) in the `external/dlm` subdirectory. This example consists of a single C source file, and the desired sharable library exports a single function called `IDL_Load`. The following MAKE_DLL statement builds this sharable library, leaving the resulting file in the directory given by `!MAKE_DLL.COMPILE_DIRECTORY`:

```
; Locate the source file:
INDIR = FILEPATH('', SUBDIRECTORY=['external', 'dlm'])
; Build the sharable library:
MAKE_DLL, 'testmodule', 'IDL_Load', INPUT_DIRECTORY=INDIR
```

Example 2: Using GCC

IDL is built with the standard vendor-supported C compiler in order to get maximum integration with the target system. MAKE_DLL assumes that you have the same compiler installed on your system and its defaults are targeted to use it. To use other compilers, you tell MAKE_DLL how to use them.

For example, many IDL users have the gcc compiler installed on their systems. This example (tested under 32-bit Solaris 7 using gcc 2.95.2) shows how to use gcc to build the testmodule sharable library from the previous example:

```
; We need the include directory for the IDL export.h header
; file. One way to get this is to extract it from the
; !MAKE_DLL system variable using the STREGEX function
INCLUDE=STREGEX(!MAKE_DLL.CC, '-I[^ ]+', /EXTRACT)
; Locate the source file
INDIR = FILEPATH('', SUBDIRECTORY=['external', 'dlm'])
; Build the sharable library, using the CC keyword to specify gcc:
MAKE_DLL, 'testmodule', 'IDL_Load', INPUT_DIRECTORY=INDIR, $
      CC='gcc -c -fPIC '+ INCLUDE + '%C -o %O'
```

Version History

Introduced: 5.4

REUSE_EXISTING keyword added: 5.6

See Also

[!MAKE_DLL](#)

MAP_2POINTS

The MAP_2POINTS function returns parameters such as distance, azimuth, and path relating to the great circle or rhumb line connecting two points on a sphere.

This routine is written in the IDL language. Its source code can be found in the file `map_2points.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MAP_2POINTS( lon0, lat0, lon1, lat1 [, DPATH=value | , /METERS |  
, /MILES | , NPATH=integer{2 or greater} | , /PARAMETERS | , RADIANS=value]  
[, /RADIUS] [, /RHUMB] )
```

Return Value

This function returns a two-element vector containing the distance and azimuth of the great circle or rhumb line connecting the two points, P0 to P1, in the specified angular units, unless one or more of the keywords NPATH, DPATH, METERS, MILES, PARAMETERS, or RADIUS is specified. See the keyword descriptions for the return value associated with each of these keywords.

If MILES, METERS, or RADIUS is not set, distances are angular distance, from 0 to 180 degrees (or 0 to !DPI if the RADIANS keyword is set). Azimuth is measured in degrees or radians, east of north.

Arguments

Lon0, Lat0

Longitude and latitude of the first point, P0.

Lon1, Lat1

Longitude and latitude of the second point, P1.

Keywords

DPATH

Set this keyword to a value specifying the maximum angular distance between the points on the path in the prevalent units, degrees or radians.

METERS

Set this keyword to return the distance between the two points in meters, calculated using the Clarke 1866 equatorial radius of the earth.

MILES

Set this keyword to return the distance between the two points in miles, calculated using the Clarke 1866 equatorial radius of the earth.

NPATH

Set this keyword to a value specifying the number of points to return. If this keyword is set, the function returns a (2, NPATH) array containing the longitude/latitude of the points on the great circle or rhumb line connecting P0 and P1. For a great circle, the points will be evenly spaced in distance, while for a rhumb line, the points will be evenly spaced in longitude.

Note

This keyword must be set to an integer of 2 or greater.

PARAMETERS

Set this keyword to return the parameters determining the great circle connecting the two points, $[\sin(c), \cos(c), \sin(az), \cos(az)]$, where c is the great circle angular distance, and az is the azimuth of the great circle at P0, in degrees east of north.

RADIANS

Set this keyword if inputs and angular outputs are to be specified in radians. The default is degrees.

RADIUS

Set this keyword to a value specifying the radius of the sphere to be used to calculate the distance between the two points. If this keyword is specified, the function returns the distance between the two points calculated using the given radius.

RHUMB

Set this keyword to return the distance and azimuth of the rhumb line connecting the two points, P0 to P1. The default is to return the distance and azimuth of the great circle connecting the two points. A rhumb line is the line of constant direction connecting two points.

Examples

The following examples use the geocoordinates of two points, Boulder and London:

```
B = [ -105.19, 40.02] ;Longitude, latitude in degrees.
L = [ -0.07, 51.30]
```

Example 1

Print the angular distance and azimuth, from B, of the great circle connecting the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1])
```

IDL prints 67.854333 40.667833

Example 2

Print the angular distance and course (azimuth), connecting the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1],/RHUMB)
```

IDL prints 73.966283 81.228057

Example 3

Print the distance in miles between the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1],/MILES)
```

IDL prints 4693.5845

Example 4

Print the distance in miles along the rhumb line connecting the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1], /MILES, /RHUMB)
```

IDL prints 5116.3571

Example 5

Display a map containing the two points, and annotate the map with both the great circle and the rhumb line path between the points, drawn at one degree increments:

```
MAP_SET, /MOLLWEIDE, 40,-50, /GRID, SCALE=75e6,/CONTINENTS
PLOTS, MAP_2POINTS(B[0], B[1], L[0], L[1],/RHUMB, DPATH=1)
PLOTS, MAP_2POINTS(B[0], B[1], L[0], L[1],DPATH=1)
```

This displays the following map:

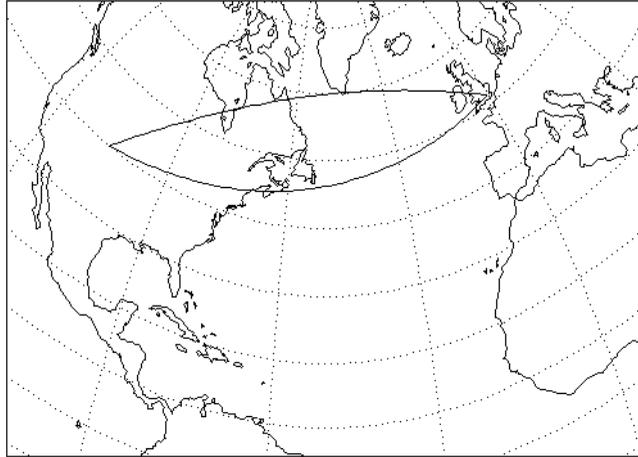


Figure 3-43: Map annotated with great circle and rhumb line path between Boulder and London, drawn at one degree increments.

Version History

Introduced: 5.4

See Also

[MAP_SET](#)

MAP_CONTINENTS

The MAP_CONTINENTS procedure draws continental boundaries, filled continents, political boundaries, coastlines, and/or rivers, over a map projection. Outlines can be drawn in low or high-resolution (if the optional high-resolution CIA World Map database is installed). If MAP_CONTINENTS is called without any keywords, it draws low-resolution, unfilled continent outlines.

The MAP_STRUCTURE keyword can be used to draw directly to UV (Cartesian) coordinates. Otherwise, MAP_SET must be called before MAP_CONTINENTS to establish the projection type, the center of the projection, polar rotation and geographic limits.

Syntax

```
MAP_CONTINENTS [ , /COASTS ] [ , COLOR=index ] [ , /CONTINENTS ]
  [ , /COUNTRIES ] [ , FILL_CONTINENTS={ 1 | 2 } [ , ORIENTATION=value ] ]
  [ , /HIRES ] [ , LIMIT=vector ] [ , MAP_STRUCTURE=structure ]
  [ , MLINESTYLE={ 0 | 1 | 2 | 3 | 4 | 5 } ] [ , MLINETHICK=value ] [ , /RIVERS ]
  [ , SPACING=centimeters ] [ , /USA ]
```

Graphics Keywords: [, /T3D] [, ZVALUE=*value*{ 0 to 1 }]

Keywords

COASTS

Set this keyword to draw coastlines, islands, and lakes instead of the default continent outlines. Note that if you are using the low-resolution map database (if the HIRES keyword is *not* set), many islands are drawn even when COASTS is not set. If you are using the high-resolution map database (if the HIRES keyword *is* set), no islands are drawn unless COASTS is set.

COLOR

Set this keyword to the color index of the lines being drawn.

CONTINENTS

Set this keyword to plot the continental boundaries. This is the default, unless COASTS, COUNTRIES, RIVERS and/or USA is set.

Note

If you are using the low-resolution map database (if the HIRES keyword is *not* set), outlines for continents, islands, and lakes are drawn when the CONTINENTS keyword is set. If you are using the high-resolution map database (if the HIRES keyword *is* set), only continental outlines are drawn when the CONTINENTS keyword is set. To draw islands and lakes when using the high-resolution map database, use the COASTS keyword.

Note

If you are using the USA and CONTINENTS keywords in conjunction to map the outline of each state in the United States onto an existing outline of the continent, you may see discrepancies in the coastline. This is due to the fact that the two outlines are derived from different databases (the USA keyword uses a geographical database, and the CONTINENTS keyword uses a geological database).

COUNTRIES

Set this keyword to draw political boundaries as of 1993.

FILL_CONTINENTS

Set this keyword to 1 to fill continent boundaries with a solid color. The color is set by the COLOR keyword. Set this keyword to 2 to fill continent boundaries with a line fill. For line filling, the COLOR, MLINESTYLE, MLINE THICK, ORIENTATION, and SPACING keywords can be used to control the type of line fill.

Note

When using this keyword in conjunction with the HIRES keyword, lakes on continents will be filled and islands will not be filled.

HIRES

Set this keyword to use high-resolution map data instead of the default low-resolution data. This option is only available if you have installed the optional high-resolution map datasets. If the high-resolution data is not available, a warning is printed and the low-resolution data is used instead.

This keyword can be used in conjunction with the COASTS, COUNTRIES, FILL_CONTINENTS, and RIVERS keywords.

LIMIT

Set this keyword to a four-element vector [Lat_{min} , Lon_{min} , Lat_{max} , Lon_{max}] to only plot continents that pass through the LIMIT rectangle. The points (Lat_{min} , Lon_{min}) and (Lat_{max} , Lon_{max}) are the latitudes and longitudes of two points diagonal from each other on the region's boundary. The default is to use the limits from the current map projection.

Note

Line segments for continents which extend outside of the LIMIT rectangle will still be plotted.

MAP_STRUCTURE

Set this keyword to a !MAP structure, as returned from [MAP_PROJ_INIT](#). If this keyword is set, the !MAP system variable is ignored, and the polygons and polylines are drawn using UV (Cartesian) coordinates.

MLINESTYLE

The line style of the boundaries being drawn. The default is solid lines. Valid linestyle are shown in the table below:

Index	Linestyle
0	Solid
1	Dotted
2	Dashed
3	Dash Dot
4	Dash Dot Dot
5	Long Dashes

Table 3-76: IDL Linestyles

MLINETHICK

The thickness of the boundary or fill lines. The default thickness is 1.

ORIENTATION

Set this keyword to the counterclockwise angle in degrees from horizontal that the line fill should be drawn. The default is 0. This keyword only has effect if the `FILL_CONTINENTS` keyword is set to 2.

RIVERS

Set this keyword to draw rivers.

SPACING

Set this keyword to the spacing, in centimeters, for a line fill. This keyword only has effect if the `FILL_CONTINENTS` keyword is set to 2. The default is 0.5 centimeters.

USA

Set this keyword to draw borders for each state in the United States in addition to continental boundaries.

Note

If you are using the `USA` and `CONTINENTS` keywords in conjunction to map the outline of each state in the United States onto an existing outline of the continent, you may see discrepancies in the coastline. This is due to the fact that the two outlines are derived from different databases (the `USA` keyword uses a geographical database, and the `CONTINENTS` keyword uses a geological database).

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for descriptions of graphics and plotting keywords not listed above. [T3D](#), [ZVALUE](#).

Examples

The following example demonstrates the use of map outlines to embellish a map projection:

```

; Handle TrueColor displays:
DEVICE, DECOMPOSED=0

; Load discrete color table:
tek_color

; Match color indices to colors we want to use:
black=0 & white=1 & red=2

```

```

green=3 & dk_blue=4 & lt_blue=5

; Set up an orthographic projection centered over the north
; Atlantic.Fill the hemisphere with dark blue. Specify black
; gridlines:
MAP_SET, /ORTHO, 40, -30, 23, /ISOTROPIC, $
      /HORIZON, E_HORIZON={FILL:1, COLOR:dk_blue}, $
      /GRID, COLOR=black

; Fill the continent boundaries with solid white:
MAP_CONTINENTS, /FILL_CONTINENTS, COLOR=white

; Overplot coastline data:
MAP_CONTINENTS, /COASTS, COLOR=black

; Add rivers, in light blue:
MAP_CONTINENTS, /RIVERS, COLOR=lt_blue

; Show national borders:
MAP_CONTINENTS, /COUNTRIES, COLOR=red, MLINE_THICK=2

```

The following example uses the `MAP_STRUCTURE` keyword to map the continents in UV (Cartesian) coordinates, without having to set up a `!MAP` transform using `MAP_SET`.

```

; GCTP Polar stereographic projection
mapStruct = MAP_PROJ_INIT(106, LIMIT=[0,-180,90,180], $
      CENTER_LATITUDE=90)

; Create a plot window using the UV Cartesian range.
PLOT, mapStruct.uv_box[[0,2]],mapStruct.uv_box[[1,3]], $
      /NODATA, /ISOTROPIC, XSTYLE=1, YSTYLE=1

MAP_CONTINENTS, MAP_STRUCTURE=mapStruct
MAP_GRID, MAP_STRUCTURE=mapStruct

```

Version History

Introduced: Pre 4.0

`MAP_STRUCTURE` keyword added: 6.1

See Also

[MAP_GRID](#), [MAP_IMAGE](#), [MAP_PATCH](#), [MAP_PROJ_INIT](#), [MAP_SET](#)

MAP_GRID

The MAP_GRID procedure draws the graticule of parallels and meridians over a map projection.

The MAP_STRUCTURE keyword may be used to draw directly to UV (Cartesian) coordinates. Otherwise, MAP_SET must be called before MAP_GRID to establish the projection type, the center of the projection, polar rotation and geographical limits.

Syntax

```
MAP_GRID [, /BOX_AXES | [, CLIP_TEXT=0] [, LATALIGN=value{0.0 to 1.0}]
  [, LONALIGN=value{0.0 to 1.0}] [, LATLAB=longitude] [, LONLAB=latitude]
  [, ORIENTATION=clockwise_degrees_from_horiz]]
  [, CHARSIZE=value] [, COLOR=index] [, /FILL_HORIZON]
  [, GLINESTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, GLINETHICK=value] [, /HORIZON]
  [, INCREMENT=value] [, LABEL=n{label_every_nth_gridline}]
  [, LATDEL=degrees] [, LATNAMES=array, LATS=vector]
  [, LONDEL=degrees] [, LONNAMES=array, LONS=vector]
  [, MAP_STRUCTURE=structure] [, /NO_GRID]
```

Graphics Keywords: [, /T3D] [, ZVALUE=value{0 to 1}]

Keywords

BOX_AXES

Set this keyword to create box-style axes for map plots where the parallels intersect the sides, and the meridians intersect the bottom and top edges of the box.

Warning

If a device (or hardware) font is in use, the labels generated by this keyword may appear incorrectly. Consider using TrueType or vector fonts instead.

CHARSIZE

Set this keyword to the size of the characters used for the labels. The default is 1.

Warning

If a device (or hardware) font is in use, this keyword will have no effect. Use the `FONTSIZE` keyword to the `DEVICE` procedure to specify character size when using hardware fonts.

CLIP_TEXT

Set this keyword to a zero value to turn off clipping of text labels. By default, text labels are clipped. This keyword is ignored if the `BOX_AXES` keyword is set.

COLOR

Set this keyword to the color index for the grid lines.

FILL_HORIZON

Set this keyword to fill the current `map_horizon`.

GLINESTYLE

If set, the line style used to draw the grid of parallels and meridians. See “`LINestyle`” on page 4337 for a list of available linestyles. The default index is 1, drawing a dotted line.

GLINETHICK

Set this keyword to the thickness of the grid lines. Default is 1.

HORIZON

Set this keyword to draw the current map horizon.

INCREMENT

Set this keyword to the spacing between graticule points.

LABEL

Set this keyword to label the parallels and meridians with their corresponding latitudes and longitudes. Setting this keyword to an integer will cause every `LABEL` gridline to be labeled (that is, if `LABEL=3` then every third gridline will be labeled).

The starting point for determining which gridlines are labeled is the minimum latitude or longitude (-180 to 180), unless the LATS or LONS keyword is set to a single value. In this case, the starting point is the value of LATS or LONS.

LATALIGN

This keyword controls the alignment of the text baseline for latitude labels. A value of 0.0 left justifies the label, 1.0 right justifies it, and 0.5 centers it. This keyword is ignored if the BOX_AXES keyword is set.

LATDEL

Set this keyword equal to the spacing (in degrees) between parallels of latitude in the grid. If this keyword is not set, a suitable value is determined from the current map projection.

LATLAB

The longitude at which to place latitude labels. The default is the center longitude on the map. This keyword is ignored if the BOX_AXES keyword is set.

LATNAMES

Set this keyword equal to an array specifying the names to be used for the latitude labels. By default, this array is automatically generated in units of degrees. The LATNAMES array can be either type string or any single numeric type, but should not be of mixed type.

When LATNAMES is specified, the LATS keyword must also be specified. The number of elements in the two arrays need not be equal. If there are more elements in the LATNAMES array than in the LATS array, the extra LATNAMES are ignored. If there are more elements in the LATS array than in the LATNAMES array, labels in degrees will be automatically provided for the missing latitude labels.

The LATNAMES keyword can be also used when the LATS keyword is set to a single value. In this case, the first label supplied will be used at the specified latitude; subsequent names will be placed at the next latitude line to the north, wrapping around the globe if appropriate. Caution should be used when using LATNAMES in conjunction with a single LATS value, since the number of visible latitude gridlines is dependent on many factors.

LATS

Set this keyword equal to a one or more element vector of latitudes for which lines will be drawn (and optionally labeled). If LATS is omitted, appropriate latitudes will

be generated based on the value of the (optional) LATDEL keyword. If LATS is set to a single value, that latitude and a series of automatically generated latitudes will be drawn (and optionally labeled). Automatically generated latitudes have the values:

```
[ . . . , LATS-LATDEL , LATS , LATS+LATDEL , . . . ]
```

over the extent of the map. If LATS is a single value, that value is taken to be the starting point for labelling (See the LABEL keyword).

LONALIGN

This keyword controls the alignment of the text baseline for longitude labels. A value of 0.0 left justifies the label, 1.0 right justifies it, and 0.5 centers it. This keyword is ignored if the BOX_AXES keyword is set.

LONDEL

Set this keyword equal to the spacing (in degrees) between meridians of longitude in the grid. If this keyword is not set, a suitable value is determined from the current map projection.

LONLAB

The latitude at which to place longitude labels. The default is the center latitude on the map. This keyword is ignored if the BOX_AXES keyword is set.

LONNAMES

Set this keyword equal to an array specifying the names to be used for the longitude labels. By default, this array is automatically generated in units of degrees. The LONNAMES array can be either type string or any single numeric type, but should not be of mixed type.

When LONNAMES is specified, the LONS keyword must also be specified. The number of elements in the two arrays need not be equal. If there are more elements in the LONNAMES array than in the LONS array, the extra LONNAMES are ignored. If there are more elements in the LONS array than in the LONNAMES array, labels in degrees will be automatically provided for the missing longitude labels.

The LONNAMES keyword can be also used when the LONS keyword is set to a single value. In this case, the first label supplied will be used at the specified longitude; subsequent names will be placed at the next longitude line to the east, wrapping around the globe if appropriate. Caution should be used when using LONNAMES in conjunction with a single LONS value, since the number of visible longitude gridlines is dependent on many factors.

LONS

Set this keyword equal to a one or more element vector of longitudes for which lines will be drawn (and optionally labeled). If LONS is omitted, appropriate longitudes will be generated based on the value of the (optional) LONDEL keyword. If LONS is set to a single value, that longitudes and a series of automatically generated longitudes will be drawn (and optionally labeled). Automatically generated longitudes have the values:

```
[ . . . , LONS-LONDEL , LONS , LONS+LONDEL , . . . ]
```

over the extent of the map. If LONS is a single value, that value is taken to be the starting point for labelling (See the LABEL keyword).

MAP_STRUCTURE

Set this keyword to a !MAP structure, as returned from [MAP_PROJ_INIT](#). If this keyword is set, the !MAP system variable is ignored, and the grid lines are drawn using UV (Cartesian) coordinates.

NO_GRID

Set this keyword if you only want labels but not gridlines.

ORIENTATION

Set this keyword equal to an angle in degrees from horizontal (in the clockwise direction) to rotate the labels. This keyword is ignored if the BOX_AXES keyword is set.

Warning

If a device (or hardware) font is in use, this keyword will have no effect.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for descriptions of graphics and plotting keywords not listed above. [T3D](#), [ZVALUE](#).

Examples

The following example creates an orthographic projection, defines which latitudes to label, and provides text labels. Note that the text labels are rotated to match the orientation of the map projection.

```
; Set up an orthographic projection:
```

```

MAP_SET, /ORTHO, 10, 20, 30, /ISOTROPIC, /CONTINENTS, /HORIZON
; Define latitudes of interest:
lats = [ -80, -45, -30, -20, 0, 15, 27, 35, 45, 55, 75]
; Create string equivalents of latitudes:
latnames = strtrim(lats, 2)
; Label the equator:
latnames(where(lats eq 0)) = 'Equator'
; Draw the grid:
MAP_GRID, LABEL=2, LATS=lats, LATNAMES=latnames, LATLAB=7, $
    LONLAB=-2.5, LONDEL=20, LONS=-15, ORIENTATION=-30

```

The following example uses the `MAP_STRUCTURE` keyword to map the grid lines in UV (Cartesian) coordinates, without having to set up a `!MAP` transform using `MAP_SET`.

```

; GCTP Polar stereographic projection
mapStruct = MAP_PROJ_INIT(106, LIMIT=[0,-180,90,180], $
    CENTER_LATITUDE=90)

; Create a plot window using the UV Cartesian range.
PLOT, mapStruct.uv_box[[0,2]],mapStruct.uv_box[[1,3]], $
    /NODATA, /ISOTROPIC, XSTYLE=1, YSTYLE=1

MAP_CONTINENTS, MAP_STRUCTURE=mapStruct
MAP_GRID, MAP_STRUCTURE=mapStruct

```

Version History

Introduced: Pre 4.0

`MAP_STRUCTURE` keyword added: 6.1

See Also

[MAP_CONTINENTS](#), [MAP_IMAGE](#), [MAP_PATCH](#), [MAP_PROJ_INIT](#),
[MAP_SET](#)

MAP_IMAGE

The MAP_IMAGE function warps an image (or other dataset) to a map projection. This function provides an easy method for displaying geographical data as an image on a map. The MAP_SET or MAP_PROJ_INIT procedures should be called prior to calling MAP_IMAGE.

MAP_IMAGE works in image (graphic) space. For each destination pixel (when COMPRESS is set to one) MAP_IMAGE calculates the latitude and longitude by applying the inverse map projection. This latitude and longitude are then used to index and interpolate the *Image* argument, obtaining an interpolated value for the destination pixel. The time required by MAP_IMAGE depends mainly on the number of pixels in the destination and the setting of the COMPRESS parameter.

Note

MAP_IMAGE is more efficient than MAP_PATCH when the input data set is large compared to the destination area. If the converse is true, MAP_PATCH is more efficient.

Syntax

```
Result = MAP_IMAGE( Image [, Startx, Starty [, Xsize, Ysize]]
  [, LATMIN=degrees{-90 to 90}] [, LATMAX=degrees{-90 to 90}]
  [, LONMIN=degrees{-180 to 180}] [, LONMAX=degrees{-180 to 180}]
  [, /BILINEAR] [, COMPRESS=value] [, SCALE=value]
  [, MAP_STRUCTURE=structure] [, MASK=variable]
  [, MAX_VALUE=value] [, MIN_VALUE=value] [, MISSING=value] )
```

Return Value

Returns the image or dataset warped to the current map projection.

Arguments

Image

A two-dimensional array containing the image to be overlaid on the map.

Startx

A named variable that, upon return, contains the X coordinate position where the left edge of the image should be placed on the screen.

Starty

A named variable that, upon return, contains the Y coordinate position where the left edge of the image should be placed on the screen.

Xsize

A named variable that, upon return, contains the width of the image expressed in graphic coordinate units. If the current graphics device uses scalable pixels, the values of *Xsize* and *Ysize* should be passed to the TV procedure.

Ysize

A named variable that, upon return, contains the height of the image expressed in graphic coordinate units. If the current graphics device uses scalable pixels, the values of *Xsize* and *Ysize* should be passed to the TV procedure.

Keywords

LATMIN

The latitude corresponding to the first row of *Image*. The default is -90 degrees. Note also that $-90^\circ \leq \text{LATMIN} < \text{LATMAX} \leq 90^\circ$.

LATMAX

The latitude corresponding to the last row of *Image*. The default value is 90 degrees. Note also that $-90^\circ \leq \text{LATMIN} < \text{LATMAX} \leq 90^\circ$.

LONMIN

The longitude corresponding to the first (leftmost) column of the *Image* argument. Select LONMIN so that $-180^\circ \leq \text{LONMIN} \leq 180^\circ$. The default value is -180.

LONMAX

The longitude corresponding to the last (rightmost) column of the *Image* argument. Select LONMAX so that it is larger than LONMIN. If the longitude of the last column is equal to $(\text{LONMIN} - (360. / N_x)) \text{ MODULO } 360$, it is assumed that the

image covers all longitudes (N_x being the total number of columns in the *Image* argument).

BILINEAR

Set this flag to use bilinear interpolation to soften edges in the returned image, otherwise, nearest neighbor sampling is used.

COMPRESS

This keyword, the interpolation compression flag, controls the accuracy of the results from `MAP_IMAGE`. The default is 4 for output devices with fixed pixel sizes. The inverse projection transformation is applied to each i th row and column. Setting this keyword to a higher number saves time while lower numbers produce more accurate results. Setting this keyword to 1 solves the inverse map transformation for every pixel of the output image.

SCALE

Set this keyword to the pixel/graphics scale factor for devices with scalable pixels (e.g., PostScript). The default is 0.02 pixels/graphic coordinate. This setting yields an approximate output image size of 350 x 250. Make this number larger for more resolution (and larger PostScript files and images), or smaller for faster, smaller, and less accurate images.

MAP_STRUCTURE

Set this keyword to a !MAP structure as returned from `MAP_PROJ_INIT`, to be used instead of the default !MAP projection. This keyword is useful if you want to display the image in a UV (Cartesian) coordinate system, instead of the !MAP coordinate system.

MASK

Set this keyword equal to a named variable that will contain a byte array of the same dimensions as the *Result*, containing a mask of the good values. Values in the *Result* that were set to `MISSING` (that is, values were either off the map or outside the range specified by the `MIN_VALUE` and `MAX_VALUE` keywords) will have a mask value of zero, while all other mask values will be one.

Tip

The MASK array allows you to distinguish values in the *Result* that were explicitly set to `MISSING` versus those that happen to have the same value. The array may

also be useful in defining an alpha (transparency) channel for an object graphics image.

MAX_VALUE

Data points with values equal to or greater than this value will be treated as missing data, and will be set to the value specified by the MISSING keyword.

MIN_VALUE

Data points with values equal to or less than this value will be treated as missing data, and will be set to the value specified by the MISSING keyword.

MISSING

The pixel value to set areas outside the valid map coordinates. If this keyword is omitted, areas outside the map are set to 255 (white) if the current graphics device is PostScript, otherwise they are set to 0.

Examples

The following lines of code set up an orthographic map projection and warp a simple image to it.

```

; Create a simple image to be warped:
image = BYTSC(SIN(DIST(400)/10))

; Display the image so we can see what it looks like before
; warping:
TV, image
latmin = -65
latmax = 65

; Left edge is 160 East:
lonmin = 160

; Right edge is 70 West = +360:
lonmax = -70 + 360
MAP_SET, 0, -140, /ORTHOGRAPHIC, /ISOTROPIC, $
LIMIT=[latmin, lonmin, latmax, lonmax]
result = MAP_IMAGE(image, Startx, Starty, COMPRESS=1, $
LATMIN=latmin, LONMIN=lonmin, $
LATMAX=latmax, LONMAX=lonmax)

; Display the warped image on the map at the proper position:
TV, result, Startx, Starty

```

```
; Draw gridlines over the map and image:  
MAP_GRID, latdel=10, lonel=10, /LABEL, /HORIZON  
  
; Draw continent outlines:  
MAP_CONTINENTS, /coasts
```

Version History

Introduced: Pre 4.0

MAP_STRUCTURE, MASK keywords added: 6.1

See Also

[MAP_CONTINENTS](#), [MAP_GRID](#), [MAP_PATCH](#), [MAP_PROJ_IMAGE](#),
[MAP_PROJ_INIT](#), [MAP_SET](#)

MAP_PATCH

The `MAP_PATCH` function warps an image (or other dataset) to the current map projection. Mapping coordinates should be setup via a call to `MAP_SET` before using `MAP_PATCH`.

`MAP_PATCH` works in object (data) space. It divides the input data set, *Image_Orig*, into triangular patches, either directly from the implicit rectangular grid, or by triangulating the data points on the surface of the sphere using the `TRIANGULATE` procedure. These triangular patches are then projected to the map plane in the image space of the destination array and then interpolated. The time required by `MAP_PATCH` depends mainly on the number of elements in the input array.

Note

`MAP_PATCH` is more efficient than `MAP_IMAGE` when the destination area is large compared to the input data set. If the converse is true, `MAP_IMAGE` is more efficient.

This routine is written in the IDL language. Its source code can be found in the file `map_patch.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MAP_PATCH( Image_Orig [, Lons, Lats] [, LAT0=value] [, LAT1=value]
  [, LON0=value] [, LON1=value] [, MAX_VALUE=value] [, MISSING=value]
  [, /TRIANGULATE] [, XSIZE=variable] [, XSTART=variable]
  [, YSIZE=variable] [, YSTART=variable] )
```

Return Value

Returns the image or dataset warped to the current map projection.

Arguments

Image_Orig

A one- or two-dimensional array that contains the data to be overlaid on the map. If the `TRIANGULATE` keyword is not set, *Image_Orig* must be a two-dimensional array. Rows and columns must be arranged in increasing longitude and latitude order. Also, the corner points of each cell must be contiguous. This means that the seam of a map must lie on a cell boundary, not in its interior, splitting the cell.

Lons

An optional vector that contains the longitude value for each column in *Image_Orig*. If *Lons* is a one-dimensional vector, longitude ($Image_Orig[i,j] = Lons[i]$); if *Lons* is a two-dimensional vector, longitude ($Image_Orig[i,j] = Lons[i,j]$).

This argument can be omitted if the longitudes are equally-spaced and the beginning and ending longitudes are specified with the LON0 and LON1 keywords.

Lats

An optional vector that contains the latitude value for each row in *Image_Orig*. If *Lats* is a one-dimensional vector, latitude ($Image_Orig[i,j] = Lats[i]$); if *Lats* is a two-dimensional vector, latitude ($Image_Orig[i,j] = Lats[i,j]$).

This argument can be omitted if the latitudes are equally-spaced and the beginning and ending latitudes are specified with the LAT0 and LAT1 keywords.

Keywords

LAT0

The latitude of the first row of data. The default is -90.

LAT1

The latitude of the last row of data. The default is +90.

LON0

The longitude of the first column of data. The default is -180.

LON1

The longitude of the last column of data. The default is $180 - (360/\text{number of rows})$

MAX_VALUE

The largest data value to be warped. Values in *Image_Orig* greater than this value are considered missing. Pixels in the output image that correspond to these missing values are set to the value specified by the MISSING keyword.

MISSING

Set this keyword to a value to be used for areas outside the valid map coordinates (i.e., the “background color”). If the current plotting device is PostScript, the default is 255 (white). Otherwise, the default is 0 (usually black).

TRIANGULATE

Set this keyword to convert the input data to device space and triangulate them. This keyword must be specified if the connectivity of the data points is not rectangular and monotonic in device space.

XSIZE

Set this keyword to a named variable in which the width of the output image is returned, in graphic coordinate units. If the current graphics device has scalable pixels (e.g., PostScript), the values returned by XSIZE and YSIZE should be passed to the TV procedure.

XSTART

Set this keyword to a named variable in which the X coordinate where the left edge of the image should be placed on the screen is returned.

YSIZE

Set this keyword to a named variable in which the height of the output image is returned, in graphic coordinate units. If the current graphics device has scalable pixels (e.g., PostScript), the values returned by XSIZE and YSIZE should be passed to the TV procedure.

YSTART

Set this keyword to a named variable in which the Y coordinate where the bottom edge of the image should be placed on the screen is returned.

Examples

```
; Form a 24 x 24 dataset on a sphere:
n = 24

; Specify equally gridded latitudes:
lat = replicate(180./(n-1),n) # findgen(n) - 90
```

```
; Specify equally gridded longitudes:
lon = findgen(n) # replicate(360./(n-1), n)

; Convert to Cartesian coordinates:
x = cos(lon * !dtor) * cos(lat * !dtor)
y = sin(lon * !dtor) * cos(lat * !dtor)
z = sin(lat * !dtor)

; Set interpolation function to scaled distance squared
; from (1,1,0):
f = BYTSCL((x-1)^2 + (y-1)^2 + z^2)

; Set up projection:
MAP_SET, 90, 0, /STEREO, /ISOTROPIC, /HORIZ

; Grid and display the data:
TV, MAP_PATCH(f, XSTART=x0, YSTART=y0), x0, y0

; Draw gridlines over the map and image:
MAP_GRID

; Draw continent outlines:
MAP_CONTINENTS

; Draw a horizon line:
MAP_HORIZON
```

Version History

Introduced: 4.0

See Also

[MAP_CONTINENTS](#), [MAP_GRID](#), [MAP_IMAGE](#), [MAP_SET](#)

MAP_PROJ_FORWARD

The MAP_PROJ_FORWARD function transforms map coordinates from longitude and latitude to Cartesian (x, y) coordinates, using either the !MAP system variable or a supplied map projection structure.

Syntax

```
Result = MAP_PROJ_FORWARD(Longitude [, Latitude]
  [, CONNECTIVITY=vector] [, /FILL] [, MAP_STRUCTURE=value]
  [, POLYGONS=variable] [, POLYLINES=variable] [, /RADIANS] )
```

Return Value

The result is a (2, n) array containing the Cartesian (x, y) coordinates.

Note

If the POLYGONS or POLYLINES keyword is present, the number of points in the result may be different than the number of input points, depending upon whether clipping and splitting occurs.

Arguments

Longitude

An *n*-element vector containing the longitude values. If the *Latitude* argument is omitted, *Longitude* must be a (2, *n*) array of longitude and latitude pairs.

Latitude

An *n*-element vector containing latitude values. If this argument is omitted, *Longitude* must be a (2, *n*) array of longitude and latitude pairs.

Keywords

CONNECTIVITY

Set this keyword to a vector containing an input connectivity list for polygons or polylines. The CONNECTIVITY keyword allows you to specify multiple polygons or polylines using a single array. The CONNECTIVITY list is a one-dimensional integer array of the form:

$$\left[m_1, i_0, i_1, \dots, i_{m_1-1}, m_2, i_0, i_1, \dots, i_{m_2-1}, \dots, m_n, i_0, i_1, \dots, i_{m_n-1} \right]$$

where each m_j is an integer specifying the number of vertices that define the polyline or polygon (the *vertex count*), and each associated set of $i_0 \dots i_{m-1}$ are indices into the arrays of vertices specified by the *Longitude* and *Latitude* arguments.

For example, to draw polylines between the first, third, and sixth longitude and latitude values and the fourth, sixth, ninth, and tenth longitude and latitude values, set the CONNECTIVITY array equal to [3, 0, 2, 5, 4, 3, 5, 8, 9].

To ignore a set of entries in the CONNECTIVITY array, set the vertex count, m_j , equal to zero. (Note that if you set an m equal to zero, you must remove the associated set of $i_0 \dots i_{m-1}$ values as well.) To ignore the remaining entries in the CONNECTIVITY array, set the vertex count, m_j , equal to -1.

This keyword is ignored if neither POLYGONS nor POLYLINES is present.

FILL

Set this keyword, along with the POLYGONS keyword, to perform a tessellation on the returned polygons after any clipping or splitting has been completed. This keyword is ignored if POLYGONS is not set.

Tip

The use of the FILL keyword avoids having to first pass your data through MAP_PROJ_FORWARD, and then again through the IDLgrTessellator object. Since IDLgrPolygon can only draw convex polygons correctly, you should use the FILL keyword when your input polygon is either complex (that is, if the polygon crosses itself) or concave. Tessellation is also useful if your data contains one polygon within another, such as an island within a lake within a country. In this case the CONNECTIVITY array could be used to pass all three polygons in to MAP_PROJ_FORWARD, and the FILL keyword would automatically perform the correct tessellation.

MAP_STRUCTURE

Set this keyword to a !MAP structure variable containing the projection parameters, as constructed by the MAP_PROJ_INIT. If this keyword is omitted, the !MAP system variable is used.

POLYGONS

Set this keyword to a named variable that will contain a connectivity array of the form described above in the `CONNECTIVITY` keyword.

If this keyword is present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be the vertices of a closed polygon. In this case, polygon clipping and splitting is performed in addition to the map transform, and the connectivity array is returned in the specified variable.

If this keyword is not present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be independent points and no clipping or splitting is performed.

POLYLINES

Set this keyword to a named variable that will contain a connectivity array of the form described above in the `CONNECTIVITY` keyword.

If this keyword is present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be the vertices of a polyline. In this case, polyline clipping and splitting is performed in addition to the map transform, and the connectivity array is returned in the specified variable.

If this keyword is not present, the arrays specified by the *Longitude* and *Latitude* arguments are assumed to be independent points and no clipping or splitting is performed.

RADIANS

Set this keyword to indicate that the input longitude and latitude coordinates are in radians. By default, coordinates are assumed to be in degrees.

Example

The following example creates a latitude and longitude grid with labels for the Goodes Homolosine map projection.

```

; Helper function. Constructs the polyline objects.
PRO Ex_Map_AddPolyline, label, $
  gridLon, gridLat, sMap, oModel, oContainer, oFont, $
  LONGITUDE = longitude

longitude = KEYWORD_SET(longitude)

; Transform from lat/lon to X/Y cartesian.
gridUV = MAP_PROJ_FORWARD(gridLon, gridLat, $

```

```

        MAP=sMap, POLYLINES = gridPoly)
IF (N_ELEMENTS(gridUV) LT 2) THEN $
    RETURN

    ; Construct label object if desired.
IF (label NE '') THEN BEGIN
    oLabel = OBJ_NEW('IDLgrText', label, $
        ALIGN = longitude ? 0.5 : 1, $
        FONT = oFont, VERTICAL_ALIGN=0.5)
    oContainer->Add, oLabel
ENDIF

; Create the polyline object.
oModel->Add, OBJ_NEW('IDLgrPolyline', gridUV, $
    LABEL_OBJ = oLabel, $
    LABEL_OFFSET = longitude ? 0.35 : 0, $
    /USE_LABEL_ORIENTATION, /USE_TEXT_ALIGN, $
    POLYLINE = gridPoly)

END

; Main function. Creates a grid over a map projection.
PRO Ex_Map_Proj_Forward

; Construct !MAP structure containing the projection.
sMap = MAP_PROJ_INIT('Goodes Homolosine')

; Create a graphics model to hold the visualizations.
oModel = OBJ_NEW('IDLgrModel')
oContainer = OBJ_NEW('IDL_Container')
oFont = OBJ_NEW('IDLgrFont', SIZE = 4)
oContainer->Add, oFont
deg = STRING(176b) ; degrees symbol in Truetype

; Latitude lines.
gridLon = DINDGEN(361) - 180
latitude = 15*(INDGEN(11) - 5)

FOR i = 0,(N_ELEMENTS(latitude) - 1) DO BEGIN
    lat = latitude[i]
    gridLat = REPLICATE(lat, 361)

    ; Create the latitude label.
    label = (lat EQ 0) ? 'Equ' : $
        STRTRIM(ABS(lat),2) + deg + (['N','S'])[lat LT 0]
    Ex_Map_Addpolyline, label, gridLon, gridLat, $
        sMap, oModel, oContainer, oFont
ENDFOR

```

```

; Longitude lines.
gridLat = DINDGEN(181) - 90

; Add in some extra lines for the Goode projections.
longitude = [20*(DINDGEN(18) - 9), $
  -179.999d, -20.001d, -100.001d, -40.001d, 80.001d]

FOR i = 0, N_ELEMENTS(longitude) - 1 DO BEGIN
  lon = longitude[i]
  gridLon = REPLICATE(lon, 181)

  ; Create the longitude label.
  label = STRTRIM(ROUND(ABS(lon)), 2) + deg
  IF ((lon mod 180) NE 0) THEN $
    label = label + (['E', 'W'])[lon LT 0]
  IF (lon NE FIX(lon)) THEN label = ''

  Ex_Map_Addpolyline, label, gridLon, gridLat, $
    sMap, oModel, oContainer, oFont, /LONGITUDE
ENDFOR

; Visualize our map projection.
XOBJVIEW, oModel, SCALE = 0.9, /BLOCK

; Clean up our objects.
OBJ_DESTROY, [oModel, oContainer]

END

```

Version History

Introduced: 5.6

FILL keyword added: 6.1

See Also

[MAP_PROJ_INIT](#), [MAP_PROJ_INVERSE](#)

MAP_PROJ_IMAGE

The MAP_PROJ_IMAGE function warps an image (or other 2-dimensional dataset) from geographic coordinates (longitude and latitude) to a specified map projection.

Note

Use the MAP_SET or MAP_PROJ_INIT procedure to set up the desired map projection.

Optionally, the MAP_PROJ_IMAGE function can be used to warp an image in Cartesian (UV) coordinates from one map projection to another.

Syntax

```
Result = MAP_PROJ_IMAGE(Image [, Range] [, /BILINEAR]
    [, DIMENSIONS=vector] [, IMAGE_STRUCTURE=structure]
    [, MAP_STRUCTURE=structure] [, MASK=variable] [, MAX_VALUE=value]
    [, MIN_VALUE=value] [, MISSING=value] [, UVRANGE=variable] )
```

Return Value

This routine returns a two-dimensional array containing the warped image. The returned array has the same type as the input image.

Arguments

Image

A two-dimensional array containing the image to warp.

Range

If you are warping *Image* from geographic coordinates to a map projection, *Range* is a four-element array specifying the longitude/latitude range (in degrees) of *Image*. *Range* has the format [*Lonmin*, *Latmin*, *Lonmax*, *Latmax*]. If *Range* is not supplied, the routine uses a default range of [-180, -90, 180, 90].

If you are warping *Image* from one map projection to another (that is, if you provide a value for the IMAGE_STRUCTURE keyword), then *Range* is a four-element array specifying the Cartesian (UV) range. In this case, *Range* must be provided and has the format [*Umin*, *Vmin*, *Umax*, *Vmax*].

Tip

Use the `UVRANGE` keyword to retrieve the range in Cartesian (UV) coordinates of *Result*.

Keywords

BILINEAR

Set this keyword to use bilinear interpolation when warping *Image*. By default, the routine will use nearest neighbor sampling. Bilinear interpolation will produce smoother results but may be slower.

DIMENSIONS

Set this keyword equal to a two-element vector specifying the width and height of the desired *Result*, in pixels. If this keyword is not specified, then the *Result* has the same dimensions as the input *Image*.

IMAGE_STRUCTURE

If the input *Image* is already in a certain projection, set this keyword equal to the !MAP structure associated with the input image. If `IMAGE_STRUCTURE` is supplied, the *Range* argument *must* be provided. In this case, *Image* will be warped from the projection specified by `IMAGE_STRUCTURE` to the projection specified by `MAP_STRUCTURE`.

MAP_STRUCTURE

Set this keyword equal to a !MAP structure (as returned from the `MAP_PROJ_INIT` function) to be used instead of the default !MAP projection. If this keyword is not specified, the value of the !MAP system variable must have been previously set by a call to the `MAP_SET` procedure.

MASK

Set this keyword equal to a named variable that will contain a byte array of the same dimensions as the *Result*, containing a mask of the “good” values. Values in *Result* that were set to `MISSING` (that is, the values were either off the map or outside the range specified by the `MIN_VALUE` and `MAX_VALUE` keywords) will have a mask value of 0 (zero), while all other mask values will be 1 (one).

Tip

The MASK array allows you to distinguish values in the *Result* that were explicitly set to MISSING versus those that happen to have a value equal to the MISSING value.

MAX_VALUE

Set this keyword equal to an integer that represents the maximum value an element of the *Result* array should be allowed to have. Data points with values greater than or equal to this value will be treated as missing data, and will be set to the value specified by the MISSING keyword.

MIN_VALUE

Set this keyword equal to an integer that represents the minimum value an element of the *Result* array should be allowed to have. Data points with values less than or equal to this value will be treated as missing data, and will be set to the value specified by the MISSING keyword.

MISSING

Set this keyword equal to an integer value to be used for pixels that fall outside of the valid map coordinates or whose data values are outside of the range specified by the MIN_VALUE and MAX_VALUE keywords. The default value is 0.

UVRANGE

Set this keyword equal to a named variable that will contain the range of the UV coordinates of the *Result* in a four-element array of the form [*U min*, *V min*, *U max*, *V max*].

Example

The following example warps a global map from longitude/latitude (geographic) coordinates into the Interrupted Goode projection:

```
; Load an image in longitude/latitude (geographic) coordinates.
file = FILEPATH('avhrr.png', SUBDIRECTORY=['examples','data'])
data = READ_PNG(file, r, g, b)

; Resize the image data
red0 = REBIN(r[data], 360, 180)
green0 = REBIN(g[data], 360, 180)
blue0 = REBIN(b[data], 360, 180)
```

```

iImage, RED=red0, GREEN=green0, BLUE=blue0, $
    DIMENSIONS=[500,600], VIEW_GRID=[1,3]

; First map projection.
sMap = MAP_PROJ_INIT('Interrupted Goode')

; Warp each channel to the map projection. For the first
; channel also compute the MASK and the UVRANGE.
red1 = MAP_PROJ_IMAGE(red0, MAP_STRUCTURE=sMap, MASK=mask, $
    UVRANGE=uvrange)
green1 = MAP_PROJ_IMAGE(green0, MAP_STRUCTURE=sMap)
blue1 = MAP_PROJ_IMAGE(blue0, MAP_STRUCTURE=sMap)

iImage, RED=red1, GREEN=green1, BLUE=blue1, ALPHA=mask*255b, $
    /VIEW_NEXT

```

Now warp our image from the Interrupted Goode projection to a new projection:

```

; New map projection.
mapStruct = MAP_PROJ_INIT('Mollweide', /GCTP)

; Now use the IMAGE_STRUCTURE keyword to warp each channel
; from the Interrupted Goode projection to Mollweide.
; We also need to input the UVRANGE of the input image.
red2 = MAP_PROJ_IMAGE(red1, uvrange, IMAGE_STRUCTURE=sMap, $
    MAP_STRUCTURE=mapStruct, MASK=mask)
green2 = MAP_PROJ_IMAGE(green1, uvrange, IMAGE_STRUCTURE=sMap, $
    MAP_STRUCTURE=mapStruct)
blue2 = MAP_PROJ_IMAGE(blue1, uvrange, IMAGE_STRUCTURE=sMap, $
    MAP_STRUCTURE=mapStruct)

iImage, RED=red2, GREEN=green2, BLUE=blue2, ALPHA=mask*255b, $
    /VIEW_NEXT

```

The resulting iImage tool display is shown in the following figure.

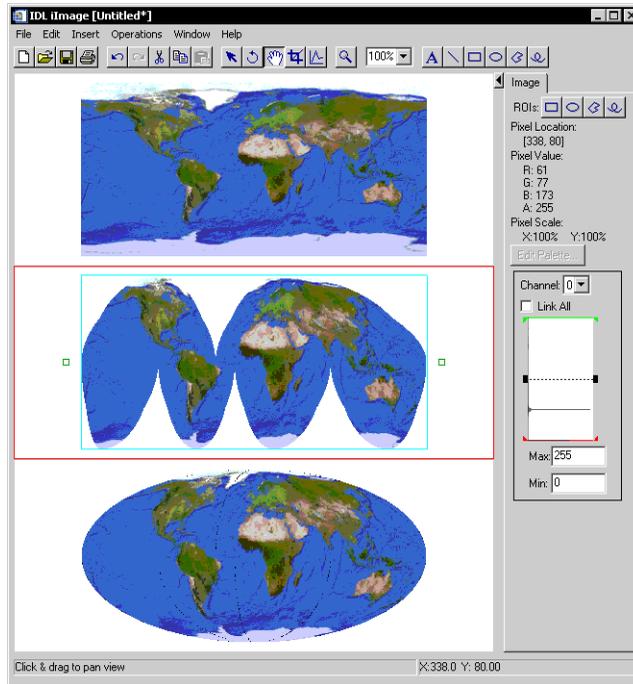


Figure 3-44: Result of the `MAP_PROJ_IMAGE` example.

Version History

Introduced: 6.1

See Also

[IMAP](#), [MAP_IMAGE](#), [MAP_PROJ_INIT](#), [MAP_SET](#)

MAP_PROJ_INFO

The MAP_PROJ_INFO procedure returns information about the current map and/or the available projections.

Note

The MAP_PROJ_INFO procedure is compiled along with the MAP_SET procedure. You *must* call the [MAP_SET](#) procedure to set up mapping parameters before calling MAP_PROJ_INFO.

This routine is written in the IDL language. Its source code can be found in `map_set.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
MAP_PROJ_INFO [, iproj] [, AZIMUTHAL=variable] [, CIRCLE=variable]
  [, CYLINDRICAL=variable] [, /CURRENT] [, LL\_LIMITS=variable]
  [, NAME=variable] [, PROJ\_NAMES=variable] [, UV\_LIMITS=variable]
  [, UV\_RANGE=variable]
```

Arguments

iproj

The projection index. If the `CURRENT` keyword is set, then the index of the current map projection is returned in `iproj`.

Keywords

AZIMUTHAL

Set this keyword to a named variable that, upon return, will be set to 1 if the projection is azimuthal and 0 otherwise.

CIRCLE

Set this keyword to a named variable that, upon return, will be set to 1 if the projection is circular or elliptical and 0 otherwise.

CURRENT

Set this keyword to use the current projection index and return that index in `iproj`.

CYLINDRICAL

Set this keyword to a named variable that, upon return, will be set to 1 if the projection is cylindrical and 0 otherwise.

LL_LIMITS

Set this keyword to a named variable that will contain the geocoordinate rectangle of the current map in degrees, [Latmin, Lonmin, Latmax, Lonmax]. This range may not always be available, especially if the LIMIT keyword was not specified in the call to MAP_SET. If either or both the longitude and latitude range are not available, the minimum and maximum values will be set to zero.

NAME

Set this keyword to a named variable that will contain the name of the projection.

PROJ_NAMES

Set this keyword to a named variable that will contain a string array containing the names of the available projections, ordered by their indices. The first projection name is stored in element one.

UV_LIMITS

Set this keyword to a named variable that will contain the UV bounding box of the current map, [Umin, Vmin, Umax, Vmax].

UV_RANGE

Set this keyword to a named variable that will contain the UV coordinate limits of the selected map projection, [Umin, Vmin, Umax, Vmax]. UV coordinates are mapped to normalized coordinates using the system variables !X.S and !Y.S. These limits are dependent upon the selected projection, but independent of the current map.

Examples

```
; Establish a projection
MAP_SET, /MERCATOR

; Obtain projection characteristics
MAP_PROJ_INFO, /CURRENT, NAME=name, AZIMUTHAL=az, $
CYLINDRICAL=cyl, CIRCLE=cir
```

On return, the variables will be set as follows:

```
AZIM      INT    = 0
CIRC      INT    = 0
CYL       INT    = 1
NAME      STRING 'Mercator'
```

Version History

Introduced: 5.0

See Also

[MAP_SET](#)

MAP_PROJ_INIT

The MAP_PROJ_INIT function initializes a mapping projection, using either IDL's own map projections or map projections from the U.S. Geological Survey's General Cartographic Transformation Package (GCTP). GCTP version 2.0 is included with IDL.

Note

The !MAP system variable is unaffected by MAP_PROJ_INIT. To use the map projection returned by MAP_PROJ_INIT for direct or object graphics, use the MAP_PROJ_FORWARD and MAP_PROJ_INVERSE functions to convert longitude/latitude values into Cartesian (x, y) coordinates before visualization.

This routine is written in the IDL language. Its source code can be found in `map_proj_init.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MAP_PROJ_INIT(Projection [, DATUM=value] [, /GCTP]
    [, LIMIT=vector] [, /RADIANS] [, /RELAXED] )
```

Keywords—Projection Parameters:

```
[, CENTER_AZIMUTH=value] [, CENTER_LATITUDE=value]
    [, CENTER_LONGITUDE=value] [, FALSE_EASTING=value]
    [, FALSE_NORTHING=value] [, HEIGHT=value]
    [, HOM_AZIM_LONGITUDE=value] [, HOM_AZIM_ANGLE=value]
    [, HOM_LATITUDE1=value] [, HOM_LATITUDE2=value]
    [, HOM_LONGITUDE1=value] [, HOM_LONGITUDE2=value]
    [, IS_ZONES=value] [, IS_JUSTIFY=value] [, MERCATOR_SCALE=value]
    [, OEA_ANGLE=value] [, OEA_SHAPEM=value] [, OEA_SHAPEN=value]
    [, ROTATION=value] [, SEMIMAJOR_AXIS=value]
    [, SEMIMINOR_AXIS=value] [, SOM_INCLINATION=value]
    [, SOM_LONGITUDE=value] [, SOM_PERIOD=value] [, SOM_RATIO=value]
    [, SOM_FLAG=value] [, SOM_LANDSAT_NUMBER=value]
    [, SOM_LANDSAT_PATH=value] [, SPHERE_RADIUS=value]
    [, STANDARD_PARALLEL=value] [, STANDARD_PAR1=value]
    [, STANDARD_PAR2=value] [, SAT_TILT=value]
    [, TRUE_SCALE_LATITUDE=value] [, ZONE=value]
```

Return Value

The result is a !MAP structure containing the map parameters, which can be used as input to the map transformation functions MAP_PROJ_FORWARD and MAP_PROJ_INVERSE.

Arguments

Projection

Set this argument to either a projection index or a scalar string containing the name of the map projection, as described in following tables:

#	Projection Name	Allowed Keyword Parameters
1	Stereographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
2	Orthographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
3	Lambert Conic	SPHERE_RADIUS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE
4	Lambert Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
5	Gnomonic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
6	Azimuthal Equidistant	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION

Table 3-77: IDL Projections

#	Projection Name	Allowed Keyword Parameters
7	Satellite	SPHERE_RADIUS, HEIGHT, SAT_TILT, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
8	Cylindrical	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
9	Mercator	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
10	Mollweide	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
11	Sinusoidal	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
12	Aitoff	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
13	Hammer Aitoff	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
14	Albers Equal Area Conic	SPHERE_RADIUS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE
15	Transverse Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
16	Miller Cylindrical	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION

Table 3-77: IDL Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
17	Robinson	SPHERE_RADIUS, CENTER_AZIMUTH, CENTER_LONGITUDE, CENTER_LATITUDE, ROTATION
18	Lambert Ellipsoid Conic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE
19	Goodes Homolosine	SPHERE_RADIUS, CENTER_LONGITUDE

Table 3-77: IDL Projections (Continued)

The following are GCTP projections:

#	Projection Name	Allowed Keyword Parameters
101	UTM	CENTER_LONGITUDE, CENTER_LATITUDE, ZONE
102	State Plane	ZONE
103	Albers Equal Area	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
104	Lambert Conformal Conic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
105	Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, TRUE_SCALE_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-78: GCTP Projections

#	Projection Name	Allowed Keyword Parameters
106	Polar Stereographic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
107	Polyconic	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
108	Equidistant Conic A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PARALLEL, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
208	Equidistant Conic B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, STANDARD_PAR1, STANDARD_PAR2, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
109	Transverse Mercator	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
110	Stereographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
111	Lambert Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-78: GCTP Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
112	Azimuthal	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
113	Gnomonic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
114	Orthographic	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
115	Near Side Perspective	SPHERE_RADIUS, HEIGHT, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
116	Sinusoidal	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
117	Equirectangular	SPHERE_RADIUS, CENTER_LONGITUDE, TRUE_SCALE_LATITUDE, FALSE_EASTING, FALSE_NORTHING
118	Miller Cylindrical	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
119	Van der Grinten	SPHERE_RADIUS, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING

Table 3-78: GCTP Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
120	Hotine Oblique Mercator A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING, HOM_LONGITUDE1, HOM_LATITUDE1, HOM_LONGITUDE2, HOM_LATITUDE2
220	Hotine Oblique Mercator B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, MERCATOR_SCALE, HOM_AZIM_ANGLE, HOM_AZIM_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING
121	Robinson	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
122	Space Oblique Mercator A	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, SOM_INCLINATION, SOM_LONGITUDE, FALSE_EASTING, FALSE_NORTHING, SOM_PERIOD, SOM_RATIO, SOM_FLAG
222	Space Oblique Mercator B	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, SOM_LANDSAT_NUMBER, SOM_LANDSAT_PATH, FALSE_EASTING, FALSE_NORTHING
123	Alaska Conformal	SEMIMAJOR_AXIS, SEMIMINOR_AXIS, FALSE_EASTING, FALSE_NORTHING
124	Interrupted Goode	SPHERE_RADIUS
125	Mollweide	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
126	Interrupted Mollweide	SPHERE_RADIUS

Table 3-78: GCTP Projections (Continued)

#	Projection Name	Allowed Keyword Parameters
127	Hammer	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
128	Wagner IV	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
129	Wagner VII	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING
130	Oblated Equal Area	SPHERE_RADIUS, OEA_SHAPEM, OEA_SHAPEP, CENTER_LONGITUDE, CENTER_LATITUDE, FALSE_EASTING, FALSE_NORTHING, OEA_ANGLE
131	Integerized Sinusoidal	SPHERE_RADIUS, CENTER_LONGITUDE, FALSE_EASTING, FALSE_NORTHING, IS_ZONES, IS_JUSTIFY

Table 3-78: GCTP Projections (Continued)

Keywords

Note

The following keywords apply to all projections.

DATUM

Set this keyword to either an integer code or a scalar string containing the name of the datum to use for the ellipsoid. The default value depends upon the projection selected, but is either the Clarke 1866 ellipsoid (datum 0), or a sphere of radius 6370.997 km (datum 19).

The following datums (or spheroids) are available for use with the DATUM keyword:

Index	Name	Semimajor axis (m)	Seminor axis (m)
0	Clarke 1866	6378206.4	6356583.8
1	Clarke 1880	6378249.145	6356514.86955
2	Bessel	6377397.155	6356078.96284
3	International 1967	6378157.5	6356772.2
4	International 1909	6378388.0	6356911.94613
5	WGS 72	6378135.0	6356750.519915
6	Everest	6377276.3452	6356075.4133
7	WGS 66	6378145.0	6356759.769356
8	GRS 1980/WGS 84	6378137.0	6356752.31414
9	Airy	6377563.396	6356256.91
10	Modified Everest	6377304.063	6356103.039
11	Modified Airy	6377340.189	6356034.448
12	Walbeck	6378137.0	6356752.314245
13	Southeast Asia	6378155.0	6356773.3205
14	Australian National	6378160.0	6356774.719
15	Krassovsky	6378245.0	6356863.0188
16	Hough	6378270.0	6356794.343479
17	Mercury 1960	6378166.0	6356784.283666
18	Modified Mercury 1968	6378150.0	6356768.337303
19	Sphere	6370997.0	6370997.0

Table 3-79: Datums available for use by MAP_PROJ_INIT.

Note

For many projections, you can specify your own datum by using either the SEMIMAJOR_AXIS and SEMIMINOR_AXIS or the SPHERE_RADIUS keywords.

GCTP

Set this keyword to indicate that the GCTP library should be used for the projection. By default, MAP_PROJ_INIT uses the IDL projection library. This keyword is ignored if the projection exists only in one system (GCTP or IDL), or if the *Projection* argument is specified as an index.

LIMIT

Set this keyword to a four-element vector of the form

```
[Latmin, Lonmin, Latmax, Lonmax]
```

that specifies the boundaries of the region to be mapped. (*Lonmin, Latmin*) and (*Lonmax, Latmax*) are the longitudes and latitudes of two points diagonal from each other on the region's boundary.

Note

When using MAP_PROJ_FORWARD, if the longitude range in LIMIT is less than or equal to 180 degrees, map clipping is performed in lat/lon coordinates *before* the transform. If the longitude range is greater than 180 degrees, map clipping is done in Cartesian coordinates *after* the transform. For non-cylindrical projections, clipping *after* the transformation to Cartesian coordinates means that some lat/lon points that fall outside the bounds specified by LIMIT may not be clipped. This occurs when the *transformed* lat/lon points fall inside the cartesian clipping rectangle.

RADIANS

Set this keyword to indicate that all parameters that represent angles are specified in radians rather than degrees.

RELAXED

If this keyword is set, any projection parameters which do not apply to the specified projection will be quietly ignored. By default, MAP_PROJ_INIT will issue errors for parameters that do not apply to the specified projection.

Projection Keywords

The following keywords apply only to some projections. Consult the list under “[Projection](#)” on page 1336 to determine which keywords apply to the projection you have selected.

Note

Unless the RADIANS keyword is set, all angles are measured in degrees, specified as a floating-point value.

CENTER_AZIMUTH

Set this keyword to the angle of the central azimuth, in degrees east of North. The default is 0 degrees. The pole is placed at an azimuth of `CENTRAL_AZIMUTH` degrees counterclockwise of North, as specified by the `ROTATION` keyword.

CENTER_LATITUDE

Set this keyword to the latitude of the point on the earth’s surface to be mapped to the center of the projection plane. Latitude is measured in degrees North of the equator and must be in the range: -90 to +90. The default value is zero.

CENTER_LONGITUDE

Set this keyword to the longitude of the point on the earth’s surface to be mapped to the center of the map projection. Longitude is measured in degrees east of the Greenwich meridian and must be in the range: -360 to +360. The default value is zero.

FALSE_EASTING

Set this keyword to the false easting value (in meters) to be added to each x coordinate for the forward transform, or subtracted from each x coordinate for the inverse transform.

FALSE_NORTHING

Set this keyword to the false northing value (in meters) to be added to each y coordinate for the forward transform, or subtracted from each y coordinate for the inverse transform.

HEIGHT

Set this keyword to the height (in meters) above the earth's surface for satellite projections.

HOM_AZIM_LONGITUDE

Set this keyword to the longitude in degrees of the central meridian point where the azimuth occurs.

HOM_AZIM_ANGLE

Set this keyword to the azimuth angle, measured in degrees, east of a north-south line that intersects the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LATITUDE1

Set this keyword to the latitude in degrees of the first point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LATITUDE2

Set this keyword to the latitude in degrees of the second point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LONGITUDE1

Set this keyword to the longitude in degrees of the first point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

HOM_LONGITUDE2

Set this keyword to the longitude in degrees of the second point on the center line. The center line is defined as the great circle path along which the Mercator cylinder touches the sphere.

IS_ZONES

Set this keyword to the number of longitudinal zones to include in the projection.

IS_JUSTIFY

Set this keyword to a flag indicating what to do with rows with an odd number of columns. The possible values are:

Value	Description
0	Indicates the extra column is on the right of the projection Y axis.
1	Indicates the extra column is on the left of the projection Y axis.
2	Calculate an even number of columns.

Table 3-80: IS_JUSTIFY Keyword Values

MERCATOR_SCALE

Set this keyword to the scale factor at the central meridian (Transverse Mercator projection) or the center of the projection (Hotine Oblique Mercator projection). For the Transverse Mercator projection, the default scale is 0.9996.

OEA_ANGLE

Set this keyword to the Oblated Equal Area oval rotation angle in degrees.

OEA_SHAPEM

Set this keyword to the Oblated Equal Area shape parameter m . The value of OEA_SHAPEM determines the horizontal flatness of the oblong region, and is usually set to a value between one and three.

OEA_SHAPEN

Set this keyword to the Oblated Equal Area oval shape parameter n . The value of OEA_SHAPEN determines the vertical flatness of the oblong region, and is usually set to a value between one and three.

Note

Setting both OEA_SHAPEM and OEA_SHAPEN equal to two is equivalent to using the Lambert Azimuthal projection.

ROTATION

Set this keyword to the angle through which the North direction should be rotated around the line between the earth's center and the point (CENTER_LONGITUDE, CENTER_LATITUDE). ROTATION is measured in degrees with the positive direction being clockwise rotation around the line. Values should be in the range -180 to +180. The default value is zero.

Note

If the center of the map is at the North pole, North is in the direction CENTER_LONGITUDE + 180. If the origin is at the South pole, North is in the direction CENTER_LONGITUDE.

SEMIMAJOR_AXIS

Set this keyword to the length (in meters) of the semimajor axis of the reference ellipsoid. The default is either the Clarke 1866 datum (6378206.4 m) or the Sphere radius (6370997 m), depending upon the projection.

SEMIMINOR_AXIS

Set this keyword to the length (in meters) of the semiminor axis of the reference ellipsoid. The default is either the Clarke 1866 datum (6356583.8 m) or the Sphere radius (6370997 m), depending upon the projection.

SOM_INCLINATION

Set this keyword to the orbit inclination angle in degrees of the ascending node, counter-clockwise from equator.

SOM_LONGITUDE

Set this keyword to the longitude in degrees of the ascending orbit at the equator.

SOM_PERIOD

Set this keyword to the period in minutes of the satellite revolution.

SOM_RATIO

Set this keyword to the Landsat ratio to compensate for confusion at the northern end of orbit. A typical value is 0.5201613.

SOM_FLAG

Set this keyword to the end of path flag for Landsat, where 0 is the start and 1 is the end.

SOM_LANDSAT_NUMBER

Set this keyword to the Landsat satellite number.

SOM_LANDSAT_PATH

Set this keyword to the Landsat path number (use 1 for Landsat 1, 2 and 3; use 2 for Landsat 4, 5 and 6).

SPHERE_RADIUS

Set this keyword to the radius (in meters) of the reference sphere. The default is 6370997 m.

STANDARD_PARALLEL

Set this keyword to the latitude in degrees of the standard parallel along which the scale is true.

STANDARD_PAR1

Set this keyword to the latitude in degrees of the first standard parallel along which the scale is true.

STANDARD_PAR2

Set this keyword to the latitude in degrees of the second standard parallel along which the scale is true.

SAT_TILT

Set this keyword to the downward tilt in degrees of the camera, in degrees from the projection horizontal.

TRUE_SCALE_LATITUDE

Set this keyword to the latitude in degrees of true scale.

ZONE

Set this keyword to an integer giving the zone for the GCTP UTM projection or GCTP State Plane projection.

Note

For the UTM projection, you may also use the `CENTER_LONGITUDE` and `CENTER_LATITUDE` keywords to set the zone. Internally, the `ZONE` value will be computed from the longitude and latitude.

Example

See [MAP_PROJ_FORWARD](#) for an example of using this function.

Version History

Introduced: 5.6

See Also

[MAP_PROJ_FORWARD](#), [MAP_PROJ_INVERSE](#), [MAP_SET](#)

MAP_PROJ_INVERSE

The MAP_PROJ_INVERSE function transforms map coordinates from Cartesian (x, y) coordinates to longitude and latitude, using either the !MAP system variable or a supplied map projection variable.

Syntax

```
Result = MAP_PROJ_INVERSE (X [, Y] [, MAP_STRUCTURE=value]  
                           [, /RADIANS] )
```

Return Value

The result is a (2, *n*) array containing the longitude/latitude coordinates.

Arguments

X

An *n*-element vector containing the x values. If the *Y* argument is omitted, *X* must be a (2, *n*) array of X and Y pairs.

Y

An *n*-element vector containing y values. If this argument is omitted, *X* must be a (2, *n*) array of X and Y pairs.

Keywords

MAP_STRUCTURE

Set this keyword to a !MAP structure variable containing the projection parameters, as constructed by the MAP_PROJ_INIT. If this keyword is omitted, the !MAP system variable is used.

RADIANS

Set this keyword to indicate that the returned longitude and latitude coordinates should be expressed in radians. By default, returned coordinates are expressed in degrees.

Version History

Introduced: 5.6

See Also

[MAP_PROJ_FORWARD](#), [MAP_PROJ_INIT](#)

MAP_SET

The MAP_SET procedure establishes the axis type and coordinate conversion mechanism for mapping points on the earth's surface, expressed in latitude and longitude, to points on a plane, according to one of several possible map projections.

The type of map projection, the map center, polar rotation and geographical limits can all be customized. The system variable !MAP retains the information needed to effect coordinate conversions to the plane and, inversely, from the projection plane to points on the earth in latitude and longitude. Users should not change the values of the fields in !MAP directly.

MAP_SET can also be made to plot the grid of latitude and longitude lines and continental boundaries by setting the keywords GRID and CONTINENTS. Many other types of boundaries can be overplotted on maps using the MAP_CONTINENTS procedure.

Note

Using MAP_SET changes the !X.TYPE system variable.

Note

If the graphics device is changed, MAP_SET (and all other mapping calls) must be re-called for the projection to be set up properly for the new device.

This routine is written in the IDL language. Its source code can be found in `map_set.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

MAP_SET [, P0lat, P0lon, Rot]

Keywords—Projection Types: [[, /AITOFF | /ALBERS | /AZIMUTHAL | /CONIC | /CYLINDRICAL | /GNOMIC | /GOODESHOMOLOSINE | /HAMMER | /LAMBERT | /MERCATOR | /MILLER_CYLINDRICAL | /MOLLWEIDE | /ORTHOGRAPHIC | /ROBINSON | /SATELLITE | /SINUSOIDAL | /STEREOGRAPHIC | /TRANSVERSE_MERCATOR] | [, NAME=*string*]]

Keywords—Map Characteristics: [/ADVANCE] [CHARSIZE=*value*] [/CLIP] [COLOR=*index*] [/CONTINENTS [CON_COLOR=*index*] [/HIRES]] [E_CONTINENTS=*structure*] [E_GRID=*structure*] [E_HORIZON=*structure*] [GLINestyle={0 | 1 | 2 | 3 | 4 | 5}] [GLINETHICK=*value*] [/GRID] [/HORIZON]

[, LABEL=*n*{label every *n*th gridline}] [, LATALIGN=*value*{0.0 to 1.0}]
 [, LATDEL=*degrees*] [, LATLAB=*longitude*] [, LONALIGN=*value*{0.0 to 1.0}]
 [, LONDEL=*degrees*] [, LONLAB=*latitude*] [, MLINESTYLE={0 | 1 | 2 | 3 | 4 | 5}]
 [, MLINETHICK=*value*] [, /NOBORDER] [, /NOERASE]
 [, REVERSE={0 | 1 | 2 | 3}] [, TITLE=*string*] [, /USA] [, XMARGIN=*value*]
 [, YMARGIN=*value*]

Keywords—Projection Parameters:

[, CENTRAL_AZIMUTH=*degrees_east_of_north*] [, ELLIPSOID=*array*]
 [, /ISOTROPIC] [, LIMIT=*vector*] [, SAT_P=*vector*] [, SCALE=*value*]
 [, STANDARD_PARALLELS=*array*]

Graphics Keywords: [, POSITION={*X*₀, *Y*₀, *X*₁, *Y*₁}] [, /T3D] [, ZVALUE=*value*{0 to 1}]

Arguments

P_{0lat}

The latitude of the point on the earth's surface to be mapped to the center of the projection plane. Latitude is measured in degrees North of the equator and P_{0lat} must be in the range: $-90^\circ \leq P_{0lat} \leq 90^\circ$.

If P_{0lat} is not set, the default value is 0.

P_{0lon}

The longitude of the point on the earth's surface to be mapped to the center of the map projection. Longitude is measured in degrees east of the Greenwich meridian and P_{0lon} must be in the range: $-180^\circ \leq P_{0lon} \leq 180^\circ$.

If P_{0lon} is not set, the default value is zero.

Rot

Rot is the angle through which the North direction should be rotated around the line *L* between the earth's center and the point (P_{0lat} , P_{0lon}). *Rot* is measured in degrees with the positive direction being clockwise rotation around line *L*. *Rot* can have values from -180° to 180° .

If the center of the map is at the North pole, North is in the direction $P_{0lon} + 180^\circ$. If the origin is at the South pole, North is in the direction P_{0lon} .

The default value of *Rot* is 0 degrees.

Keywords

Projection Type Keywords:

AITOFF

Set this keyword to select the Aitoff projection.

ALBERS

Set this keyword to select the Albers equal-area conic projection. To specify the latitude of the standard parallels, see [“STANDARD_PARALLELS”](#) on page 1363.

AZIMUTHAL

Set this keyword to select the azimuthal equidistant projection.

CONIC

Set this keyword to select Lambert’s conformal conic projection with one or two standard parallels. To specify the latitude of the standard parallels, see [“STANDARD_PARALLELS”](#) on page 1363. This keyword can be used with the ELLIPSOID keyword.

CYLINDRICAL

Set this keyword to select the cylindrical equidistant projection. Cylindrical is the default map projection.

GOODESHOMOLOSIONE

Set this keyword to select the Goode’s Homolosine Projection. The central latitude for this projection is fixed on the equator, 0 degrees latitude. This projection is interrupted, as the inventor originally intended, and is best viewed with the central longitude set to 0.

GNOMIC

Set this keyword to select the gnomonic projection. If default clipping is enabled, this projection will display a maximum of $\pm 60^\circ$ from the center of the projection area when the center is at either the equator or one of the poles.

HAMMER

Set this keyword to select the Hammer-Aitoff equal area projection.

LAMBERT

Set this keyword to select Lambert's azimuthal equal area projection.

MERCATOR

Set this keyword to select the Mercator projection. Note that this projection will not display regions within $\pm 10^\circ$ of the poles of projection.

MILLER_CYLINDRICAL

Set this keyword to select the Miller Cylindrical projection.

MOLLWEIDE

Set this keyword to select the Mollweide projection.

NAME

Set this keyword to a string indicating the projection that you wish to use. A list of available projections can be found using `MAP_PROJ_INFO`, `PROJ_NAMES=names`. This keyword will override any of the individual projection keywords.

ORTHOGRAPHIC

Set this keyword to select the orthographic projection. Note that this projection will display a maximum of $\pm 90^\circ$ from the center of the projection area.

ROBINSON

Set this keyword to select the Robinson pseudo-cylindrical projection.

SATELLITE

Set this keyword to select the satellite projection.

For the satellite projection, `POLAT` and `POLON` represent the latitude and longitude of the sub-satellite point. Three additional parameters, *P*, *Omega*, and *Gamma* (supplied as a three-element vector argument to the `SAT_P` keyword), are also required.

Note

Since all meridians and parallels are oblique lines or arcs, the **LIMIT** keyword must be supplied as an eight-element vector representing four points that delineate the limits of the map. The extent of the map limits, when expressed in latitude/longitude is a complicated polygon, rather than a simple quadrilateral.

SINUSOIDAL

Set this keyword to select the sinusoidal projection.

STEREOGRAPHIC

Set this keyword to select the stereographic projection. Note that if default clipping is enabled, this projection will display a maximum of $\pm 90^\circ$ from the center of the projection area.

TRANSVERSE_MERCATOR

Set this keyword to select the Transverse Mercator projection, also called the UTM or Gauss-Krueger projection. This projection works well with the ellipsoid form. The default ellipsoid is the Clarke 1866 ellipsoid. To change the default ellipsoid characteristics, see “[ELLIPSOID](#)” on page 1362.

Map Characteristic Keywords:**ADVANCE**

Set this keyword to advance to the next frame when the screen is set to display multiple plots. Otherwise the entire screen is erased.

CHARSIZE

The size of the characters used for the labels. The default is 1.

CLIP

Set this keyword to clip the map using the map-specific graphics technique. The default is to perform map-specific clipping. Set **CLIP=0** to disable clipping.

Note

Clipping controlled by the **CLIP** keyword applies only to the map itself. In order to disable general clipping within the plot window, you must set the system variable **!P.NOCLIP=1**. For more information, see “[NOCLIP](#)” on page 4338.

COLOR

The color index of the map border in the plotting window.

CONTINENTS

Set this keyword to plot the continental boundaries. Note that if you are using the low-resolution map database (if the HIRES keyword is *not* set), outlines for continents, islands, and lakes are drawn when the CONTINENTS keyword is set. If you are using the high-resolution map database (if the HIRES keyword *is* set), only continental outlines are drawn when the CONTINENTS keyword is set. To draw islands and lakes when using the high-resolution map database, use the COASTS keyword to the MAP_CONTINENTS procedure.

CON_COLOR

The color index for continent outlines if CONTINENTS is set.

E_CONTINENTS

Set this keyword to a structure containing extra keywords to be passed to MAP_CONTINENTS. For example, to fill continents, the FILL keyword of MAP_CONTINENTS is set to 1. To fill the continents with MAP_SET, specify E_CONTINENTS={FILL:1}.

E_GRID

Set this keyword to a structure containing extra keywords to be passed to MAP_GRID. For example, to label every other gridline on a grid of parallels and meridians, the LABEL keyword of MAP_GRID is set to 2. To do the same with MAP_SET, specify E_GRID={LABEL:2}.

E_HORIZON

Set this keyword to a structure containing extra keywords to be set as modifiers to the HORIZON keyword.

Example

To draw a Stereographic map, with the sphere filled in color index 3, enter:

```
MAP_SET, 0, 0, /STEREO, /HORIZON, /ISOTROPIC, $
      E_HORIZON={FILL:1, COLOR:3}
```

GLINESTYLE

Set this keyword to a line style index used to draw the grid of parallels and meridians. See [MLINESTYLE](#) for a list of available linestyles. The default is 1, drawing a grid of dotted lines.

GLINETHICK

Set this keyword to the thickness of the gridlines drawn if the `GRID` keyword is set. The default is 1.

GRID

Set this keyword to draw the grid of parallels and meridians.

HIRES

Set this keyword to use the high-resolution continent outlines when drawing continents. This keyword only has effect if the `CONTINENTS` keyword is also set.

HORIZON

Set this keyword to draw a horizon line, when the projection in use permits. The horizon delineates the boundary of the sphere. See [E_HORIZON](#) for more options.

LABEL

Set this keyword to label the parallels and meridians with their corresponding latitudes and longitudes. Setting this keyword to an integer will cause every `LABEL` gridline to be labeled (that is, if `LABEL=3` then every third gridline will be labeled). The starting point for determining which gridlines are labeled is the minimum latitude or longitude (-180 to 180).

LATALIGN

The alignment of the text baseline for latitude labels. A value of 0.0 left justifies the label, 1.0 right justifies it, and 0.5 centers it.

LATLAB

The longitude at which to place latitude labels. The default is the center longitude of the map.

LATDEL

Set this keyword equal to the spacing (in degrees) between parallels of latitude drawn by the MAP_GRID procedure. If this keyword is not set, a suitable value is determined from the current map projection.

LONALIGN

The alignment of the text baseline for longitude labels. A value of 0.0 left justifies the label, 1.0 right justifies it, and 0.5 centers it.

LONDEL

Set this keyword equal to the spacing (in degrees) between meridians of longitude drawn by the MAP_GRID procedure. If this keyword is not set, a suitable value is determined from the current map projection.

LONLAB

The latitude at which to place longitude labels. The default is the center latitude of the map.

MLINESTYLE

The line style index used for continental boundaries. Linestyles are described in the table below. The default is 0 for solid.

Index	Linestyle
0	Solid
1	Dotted
2	Dashed
3	Dash Dot
4	Dash Dot Dot
5	Long Dashes

Table 3-81: IDL Linestyles

MLINETHICK

The line thickness used for continental boundaries. The default is 2.

NOBORDER

Set this keyword to not draw a border around the map. The map will fill the extent of the plotting region. If **NOBORDER** is *not* specified, a margin equalling 1% of the plotting region will be placed between the map and the border.

NOERASE

Set this keyword to have **MAP_SET** not erase the current plot window. The default is to erase before drawing the map.

REVERSE

Set this keyword to one of the following values to reverse the X and/or Y axes:

- 0 = no reversal (the default)
- 1 = reverse X
- 2 = reverse Y
- 3 = reverse both.

TITLE

A string containing the main title for the map. The title appears centered above the map window.

USA

Set this keyword to draw borders for each state in the United States.

XMARGIN

A scalar or two-element vector that specifies the vertical margin between the map and screen border in character units. If a scalar is specified, the same margin will be used on both sides of the map.

YMARGIN

A scalar or two-element vector that specifies in the horizontal margin between the map and screen border in character units. If a scalar is specified, the same margin will be used on the top and bottom of the map.

Projection Parameter Keywords:

CENTRAL_AZIMUTH

Set this keyword to the angle of the central azimuth, in degrees east of North. This keyword can be used with the following projections: Cylindrical, Mercator, Miller, Mollweide, and Sinusoidal. The default is 0 degrees. The pole is placed at an azimuth of CENTRAL_AZIMUTH degrees CCW of North, as specified by the *Rot* argument.

ELLIPSOID

Set this keyword to a 3-element array, $[a, e^2, k_0]$, defining the ellipsoid for the Transverse Mercator or Lambert Conic projections.

- a : equatorial radius, in meters.
- e^2 : eccentricity squared. $e^2 = 2 * f - f^2$, where $f = 1 - b/a$ (a : equatorial radius, b : polar radius; in meters).
- k_0 : scale on the central meridian.

The default is the Clarke 1866 ellipsoid, $[6378206.4, 0.00676866, 0.9996]$.

This keyword can be used with either the CONIC or TRANSVERSE_MERCATOR keywords.

ISOTROPIC

Set this keyword to produce a map that has the same scale in the X and Y directions.

Note

The X and Y axes will be scaled isotropically and then fit within the rectangle defined by the POSITION keyword; one of the axes may be shortened. See [“POSITION”](#) on page 4339 for more information.

LIMIT

A four- or eight-element vector that specifies the limits of the map.

As a four-element vector, LIMIT has the form $[Lat_{min}, Lon_{min}, Lat_{max}, Lon_{max}]$ that specifies the boundaries of the region to be mapped. (Lat_{min}, Lon_{min}) and (Lat_{max}, Lon_{max}) are the latitudes and longitudes of two points diagonal from each other on the region's boundary.

As an eight-element vector, LIMIT has the form: [*Lat*₀, *Lon*₀, *Lat*₁, *Lon*₁, *Lat*₂, *Lon*₂, *Lat*₃, *Lon*₃]. These four latitude/longitude pairs describe, respectively, four points on the left, top, right, and bottom edges of the map extent.

Note

If the longitude range in LIMIT is less than or equal to 180 degrees, map clipping is performed in lat/lon coordinates before the transform. If the longitude range is greater than 180 degrees, map clipping is done in Cartesian coordinates after the transform. For non-cylindrical projections, clipping after the transformation to Cartesian coordinates means that some lat/lon points that fall outside the bounds specified by LIMIT may not be clipped. This occurs when the transformed lat/lon points fall inside the cartesian clipping rectangle.

SAT_P

A three-element vector containing three parameters, *P*, *Omega*, and *Gamma*, that must be supplied when using the SATELLITE projection where:

- *P* is the distance of the point of perspective (camera) from the center of the globe, expressed in units of the radius of the globe.
- *Omega* is the downward tilt of the camera, in degrees from the new horizontal. If both *Gamma* and *Omega* are 0, a Vertical Perspective projection results.
- *Gamma* is the angle, expressed in degrees clockwise from north, of the Earth's rotation. This parameter is equivalent to the *Rot* argument.

SCALE

Set this keyword to construct an isotropic map with the given scale, set to the ratio of 1:*scale*. If SCALE is not specified, the map is fit to the window. The typical scale for global maps is in the ratio of between 1:100 million and 1:200 million. For continents, the typical scale is in the ratio of approximately 1:50 million. For example, SCALE=100E6 sets the scale at the center of the map to 1:100 million, which is in the same ratio as 1 inch to 1578 miles (1 cm to 1000 km).

STANDARD_PARALLELS

Set this keyword to a one- or two-element array defining, respectively, one or two standard parallels for conic projections.

Graphics Keywords Accepted

See [Appendix B, "Graphics Keywords"](#), for descriptions of graphics and plotting keywords not listed above. POSITION, T3D, ZVALUE.

Examples

To draw a Stereographic map, with the sphere filled in color index 3:

```
MAP_SET, 0, 0, /STEREO, /HORIZON, /ISOTROPIC, E_HORIZON={FILL:1,  
COLOR:3}
```

Version History

Introduced: Pre 4.0

See Also

[MAP_CONTINENTS](#), [MAP_GRID](#), [MAP_IMAGE](#), [MAP_PROJ_INIT](#)

MATRIX_MULTIPLY

The `MATRIX_MULTIPLY` function calculates the IDL `#` operator of two (possibly transposed) arrays. The transpose operation (if desired) is done simultaneously with the multiplication, thus conserving memory and increasing the speed of the operation. If the arrays are not transposed, then `MATRIX_MULTIPLY` is equivalent to using the `#` operator.

The # Operator vs. MATRIX_MULTIPLY

The following table illustrates how various operations are performed using the `#` operator versus the `MATRIX_MULTIPLY` function:

# Operator	Function
<code>A # B</code>	<code>MATRIX_MULTIPLY(A, B)</code>
<code>transpose(A) # B</code>	<code>MATRIX_MULTIPLY(A, B, /ATRANSPOSE)</code>
<code>A # transpose(B)</code>	<code>MATRIX_MULTIPLY(A, B, /BTRANSPOSE)</code>
<code>transpose(A) # transpose(B)</code>	<code>MATRIX_MULTIPLY(A, B, /ATRANSPOSE, /BTRANSPOSE)</code>

Table 3-82: The # Operator vs. MATRIX_MULTIPLY

Note

`MATRIX_MULTIPLY` can also be used in place of the `##` operator. For example, `A ## B` is equivalent to `MATRIX_MULTIPLY(B, A)`, and `A ## TRANSPOSE(B)` is equivalent to `MATRIX_MULTIPLY(B, A, /ATRANSPOSE)`.

Syntax

Result = `MATRIX_MULTIPLY(A, B [, /ATRANSPOSE] [, /BTRANSPOSE])`

Return Value

The type for the result depends upon the input type. For byte or integer arrays, the result has the type of the next-larger integer type that could contain the result (for example, byte, integer, or long input returns type long integer). For floating-point, the result has the same type as the input.

For the case of no transpose, the resulting array has the same number of columns as the first array and the same number of rows as the second array. The second array must have the same number of columns as the first array has rows.

Note

If A and B arguments are vectors, then $C = \text{MATRIX_MULTIPLY}(A, B)$ is a matrix with $C_{ij} = A_i B_j$. Mathematically, this is equivalent to the outer product, usually denoted by $A \otimes B$.

Arguments

A

The left operand for the matrix multiplication. Dimensions higher than two are ignored.

B

The right operand for the matrix multiplication. Dimensions higher than two are ignored.

Keywords

ATRANSPOSE

Set this keyword to multiply using the transpose of A .

BTRANSPOSE

Set this keyword to multiply using the transpose of B .

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the `!CPU` system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Version History

Introduced: 5.4

See Also

[MATRIX_POWER](#), “Multiplying Arrays” in Chapter 22 of the *Using IDL* manual .

MATRIX_POWER

The `MATRIX_POWER` function computes the product of a matrix with itself. For example, the fifth power of array *A* is *A* # *A* # *A* # *A* # *A*. Negative powers are computed using the matrix inverse of the positive power.

Syntax

Result = `MATRIX_POWER(Array, N [, /DOUBLE] [, STATUS=value])`

Return Value

The result is a square array containing the value of the matrix raised to the specified power. A power of zero returns the identity matrix.

Arguments

Array

A square, two-dimensional array of any numeric type.

N

An integer representing the power. *N* may be positive or negative.

Keywords

DOUBLE

Set this keyword to return a double-precision result. Explicitly set this keyword equal to zero to return a single-precision result. The default return type depends upon the precision of *Array*.

Note

Computations are always performed using double-precision arithmetic.

STATUS

Set this keyword equal to a named variable that will contain the status of the matrix inverse for negative powers. Possible values are:

Value	Description
0	Successful completion.
1	Singular array (which indicates that the inversion is invalid).
2	Warning that a small pivot element was used and that significant accuracy was probably lost.

Table 3-83: STATUS Keyword Values

For non-negative powers, STATUS is always set to 0.

Example

Print an array to the one millionth power:

```
array = [ [0.401d, 0.600d], $
          [0.525d, 0.475d] ]
PRINT, MATRIX_POWER(array, 1e6)
```

IDL prints:

```
2.4487434e+202  2.7960773e+202
2.4465677e+202  2.7935929e+202
```

Version History

Introduced: 5.6

See Also

[MATRIX_MULTIPLY](#), “[Multiplying Arrays](#)” in Chapter 22 of the *Using IDL* manual

MAX

The MAX function returns the value of the largest element of *Array*.

Syntax

```
Result = MAX( Array [, Max_Subscript] [, /ABSOLUTE] [, DIMENSION=value]
             [, MIN=variable] [, /NAN] [, SUBSCRIPT_MIN=variable])
```

Return Value

Return the largest array element value. The type of the result is the same as the type of *Array*.

Arguments

Array

The array to be searched.

Max_Subscript

A named variable that, if supplied, is converted to a long integer containing the one-dimensional subscript of the maximum element. Otherwise, the system variable !C is set to the one-dimensional subscript of the maximum element.

Keywords

ABSOLUTE

If this keyword is set, then the absolute value of each element is used when determining the maximum values. This keyword has no effect for arrays of type byte or unsigned integer.

Note

If ABSOLUTE is set then the sign of each value is ignored when searching for the maximum. However, the return value retains the negative sign if the value was indeed negative.

Tip

For complex input, MAX by default only compares the real component of each value. Use the ABSOLUTE keyword to force MAX to compare the absolute value

of each value, and to return the complex value corresponding to the maximum absolute value.

DIMENSION

Set this keyword to the dimension over which to find the maximum values of an array. If this keyword is not present or is zero, the maximum is found over the entire array and is returned as a scalar value. If this keyword is present and nonzero, the result is the “slice” of the input array that contains the maximum value element, and the return values for *Result*, *Max_Subscript*, *MIN*, and *SUBSCRIPT_MIN* will all be arrays of one dimension less than the input array. That is, if the dimensions of *Array* are *N1*, *N2*, *N3*, and *DIMENSION=2*, the dimensions of the result are (*N1*, *N3*), and element (*i,j*) of the result contains the maximum value of *Array*[*i*, *, *j*].

For example:

```
arr = FINDGEN(2,3,2)
PRINT, arr
```

IDL prints:

```
0.00000    1.00000
2.00000    3.00000
4.00000    5.00000

6.00000    7.00000
8.00000    9.00000
10.00000   11.00000
```

```
PRINT, MAX(arr, DIMENSION=2)
```

IDL prints:

```
4.00000    5.00000
10.00000   11.00000
```

```
PRINT, MAX(arr, DIMENSION=1)
```

IDL prints:

```
1.00000    3.00000    5.00000
7.00000    9.00000    11.00000
```

MIN

A named variable to receive the value of the minimum array element. If you need to find both the minimum and maximum array values, use this keyword to avoid scanning the array twice with separate calls to *MAX* and *MIN*.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Note

If the MAX function is run on an array containing NaN values and the NAN keyword is not set, an invalid result will occur.

SUBSCRIPT_MIN

Set this keyword equal to a named variable that will contain the one-dimensional subscript of the minimum element, the value of which is available via the MIN keyword.

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Example 1

This example prints the maximum value in an array, and the subscript of that value:

```
; Create a simple two-dimensional array:
D = DIST(100)

; Print the maximum value in array D and its linear subscript:
PRINT, 'Maximum value in array D is:', MAX(D, I)
PRINT, 'The subscript of the maximum value is', I
```

IDL Output

```
Maximum value in array D is:          70.7107
The subscript of the maximum value is          5050
```

Example 2

To convert I to a two-dimensional subscript, use the commands:

```
IX = I MOD 100
IY = I/100
PRINT, 'The maximum value of D is at location ('+ STRTRIM(IX, 1) $
      + ', ' + STRTRIM(IY, 1) + ')
```

IDL Output

```
The maximum value of D is at location (50, 50)
```

Version History

Introduced: Original

ABSOLUTE keyword added: 6.1

See Also

[ARRAY_INDICES](#), [MIN](#), [WHERE](#)

MD_TEST

The MD_TEST function tests the hypothesis that a sample population is random against the hypothesis that it is not random. This two-tailed function is an extension of the “Runs Test for Randomness” and is often referred to as the Median Delta Test.

This routine is written in the IDL language. Its source code can be found in the file `md_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MD_TEST( X [, ABOVE=variable] [, BELOW=variable]  
                [, MDC=variable] )
```

Return Value

The result is a two-element vector containing the nearly-normal test statistic Z and its associated probability.

Arguments

X

An n -element integer, single- or double-precision floating-point vector.

Keywords

ABOVE

Use this keyword to specify a named variable that will contain the number of sample population values greater than the median of X .

BELOW

Use this keyword to specify a named variable that will contain the number of sample population values less than the median of X .

MDC

Use this keyword to specify a named variable that will contain the number of Median Delta Clusters (sequential values of X above and below the median).

Examples

This example tests the hypothesis that X represents a random population against the hypothesis that it does not represent a random population at the 0.05 significance level:

```
; Define a sample population:
X = [ 2.00,  0.90, -1.44, -0.88, -0.24,  0.83, -0.84, -0.74, $
      0.99, -0.82, -0.59, -1.88, -1.96,  0.77, -1.89, -0.56, $
      -0.62, -0.36, -1.01, -1.36]

; Test the hypothesis that X represents a random population against
; the hypothesis that it does not represent a random population at
; the 0.05 significance level:
result = MD_TEST(X, MDC = mdc)
PRINT, result
```

IDL prints:

```
0.459468    0.322949
```

The computed probability (0.322949) is greater than the 0.05 significance level and therefore we do not reject the hypothesis that X represents a random population.

Version History

Introduced: 4.0

See Also

[CTI_TEST](#), [FV_TEST](#), [KW_TEST](#), [R_TEST](#), [RS_TEST](#), [S_TEST](#), [TM_TEST](#),
[XSQ_TEST](#)

MEAN

The MEAN function computes the mean of a numeric vector. MEAN calls the IDL function MOMENT.

Syntax

```
Result = MEAN( X [, /DOUBLE] [, /NAN] )
```

Return Value

Returns the average value of a set of numbers.

Arguments

X

An n -element, integer, double-precision or floating-point vector.

Keywords

DOUBLE

If this keyword is set, computations are done in double precision arithmetic.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Examples

```
; Define the n-element vector of sample data:
x = [65, 63, 67, 64, 68, 62, 70, 66, 68, 67, 69, 71, 66, 65, 70]

; Compute the average:
result = MEAN(x)

; Print the result:
PRINT, result
```

IDL prints:

66.7333

Version History

Introduced: 5.1

See Also

[KURTOSIS](#), [MEANABSDEV](#), [MOMENT](#), [STDDEV](#), [SKEWNESS](#), [VARIANCE](#)

MEANABSDEV

The MEANABSDEV function computes the mean absolute deviation of an n -element vector.

Syntax

```
Result = MEANABSDEV( X [, /DOUBLE] [, /MEDIAN] [, /NAN] )
```

Return Value

Returns the average deviation.

Arguments

X

An n -element, floating-point or double-precision vector.

Keywords

DOUBLE

Set this keyword to force computations to be done in double precision arithmetic and to return a double precision result. If this keyword is not set, the computations and result depend upon the type of the input data (integer and float data return float results, while double data returns double results). This has no effect if the MEDIAN keyword is set.

MEDIAN

Set this keyword to return the average deviation from the median. By default, if MEDIAN is not set, MEANABSDEV will return the average deviation from the mean.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Examples

```
; Define an n-element vector:  
x = [1, 1, 1, 2, 5]  
  
; Compute average deviation from the mean:  
result = MEANABSDEV(x)  
  
; Print the result:  
PRINT, result
```

IDL prints:

```
1.20000
```

Version History

Introduced: 5.1

See Also

[KURTOSIS](#), [MEAN](#), [MOMENT](#), [STDDEV](#), [SKEWNESS](#), [VARIANCE](#)

MEDIAN

The MEDIAN function computes the median value. In an ordered set of values, the median is a value with an equal number of values above and below it. Median smoothing replaces each point with the median of the one- or two-dimensional neighborhood of a given width. It is similar to smoothing with a boxcar or average filter but does not blur edges larger than the neighborhood.

In addition, median filtering is effective in removing salt and pepper noise, (isolated high or low values). The scalar median is simply the middle value, which should not be confused with the average value (e.g., the median of the array [1,10,4] is 4, while the average is 5.)

Note

The MEDIAN function treats NaN values as missing data.

Syntax

```
Result = MEDIAN ( Array [, Width] [, /DOUBLE] [, DIMENSION=value]
                [, /EVEN] )
```

Return Value

Returns the median value of *Array* if one parameter is present, or applies a one- or two-dimensional median filter of the specified width to *Array* and returns the result.

Arguments

Array

The array to be processed. If *Width* is also supplied, and *Array* is of byte type, the result is of byte type. If *Array* is double precision or double complex, the computation is done using double precision and the result is double-precision. All other types are converted to single-precision floating-point, and the result is floating-point, unless the DOUBLE keyword is set. *Array* can have only one or two dimensions.

If *Width* is not given, *Array* can have any valid number of dimensions. The array is converted to single-precision floating-point, and the median value is returned as a floating-point value.

Width

The size of the one or two-dimensional neighborhood to be used for the median filter. The neighborhood has the same number of dimensions as *Array*.

Keywords

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic and to return a double-precision result.

DIMENSION

Set this keyword to the dimension over which to find the median values of an array. If this keyword is not present or is zero, the median is found over the entire array and is returned as a scalar value. If this keyword is present and nonzero, the result is a “slice” of the input array that contains the median value elements, and the return value will be an array of one dimension less than the input array.

For example, if the dimensions of *Array* are N_1 , N_2 , N_3 , and $DIMENSION = 2$, the dimensions of the result are (N_1, N_3) , and element (i,j) of the result contains the median value of $Array[i, *, j]$. This keyword is ignored if the *Width* argument is present.

EVEN

If the **EVEN** keyword is set when *Array* contains an even number of points (i.e. there is no middle number), **MEDIAN** returns the average of the two middle numbers. The returned value may not be an element of *Array*. If *Array* contains an odd number of points, **MEDIAN** returns the median value. The returned value will always be an element of *Array*—even if the **EVEN** keyword is set—since an odd number of points will always have a single middle value.

Examples

```
; Create a simple image and display it:
D = SIN(DIST(200)^0.8) & TVSCL, D

; Display D median-filtered with a width of 9:
TVSCL, MEDIAN(D, 9)

; Print the median of a four-element array, with and without
; the EVEN keyword:
PRINT, MEDIAN([1, 2, 3, 4], /EVEN)
```

```
PRINT, MEDIAN([1, 2, 3, 4])
```

IDL prints:

```
2.50000  
3.00000
```

Version History

Introduced: Original

DIMENSION keyword added: 5.6

DOUBLE keyword added: 6.1

See Also

[DIGITAL_FILTER](#), [LEEFILT](#), [MOMENT](#), [SMOOTH](#)

MEMORY

The MEMORY function returns information on the amount of dynamic memory currently in use by the IDL session if no keywords are set. If a keyword is set, MEMORY returns the specified quantity.

Syntax

```
Result = MEMORY( [ , /CURRENT | , /HIGHWATER | , /NUM_ALLOC |
                , /NUM_FREE | , /STRUCTURE ] [ , /L64 ] )
```

Return Value

The return value is a vector that is always of integer type. The following table describes the information returned if no keywords are set:

Element	Contents
<i>Result</i> [0]	Amount of dynamic memory (in bytes) currently in use by the IDL session.
<i>Result</i> [1]	The number of times IDL has made a memory allocation request from the underlying system.
<i>Result</i> [2]	The number of times IDL has made a request to free memory from the underlying system.
<i>Result</i> [3]	High water mark: The maximum amount of dynamic memory used since the last time the MEMORY function or HELP, /MEMORY procedure was called.

Table 3-84: MEMORY Function Return Values

Arguments

None.

Keywords

Note

The following keywords determine the return value of the MEMORY function. Except for L64, all of the keywords are mutually exclusive — specify at most one of the following.

CURRENT

Set this keyword to return the amount of dynamic memory (in bytes) currently in use by the IDL session.

HIGHWATER

Set this keyword to return the maximum amount of dynamic memory used since the last time the MEMORY function or HELP/MEMORY procedure was called. This can be used to determine maximum memory use of a code sequence as shown in the example below.

L64

By default, the result of MEMORY is 32-bit integer when possible, and 64-bit integer if the size of the returned values requires it. Set L64 to force 64-bit integers to be returned in all cases.

Note

Only 64-bit versions of IDL are capable of using enough memory to require 64-bit MEMORY output. Check the value of !VERSION.MEMORY_BITS to see if your IDL is 64-bit or not.

NUM_ALLOC

Returns the number of times IDL has requested dynamic memory from the underlying system.

NUM_FREE

Returns the number of times IDL has returned dynamic memory to the underlying system.

STRUCTURE

Set this keyword to return all available information about Expression in a structure. The result will be an IDL_MEMORY (32-bit) structure when possible, and an IDL_MEMORY64 structure otherwise. Set L64 to force an IDL_MEMORY64 structure to be returned in all cases.

The following are descriptions of the fields in the returned structure:

Field	Description
CURRENT	Current dynamic memory in use.
NUM_ALLOC	Number of calls to allocate memory.
NUM_FREE	Number of calls to free memory.
HIGHWATER	Maximum dynamic memory used since last call for this information.

Table 3-85: STRUCTURE Field Descriptions

Examples

To determine how much dynamic memory is required to execute a sequence of IDL code:

```

; Get current allocation and reset the high water mark:
start_mem = MEMORY(/CURRENT)

; Arbitrary code goes here.

PRINT, 'Memory required: ', MEMORY(/HIGHWATER) - start_mem

```

The MEMORY function can also be used in conjunction with DIALOG_MESSAGE as follows:

```

; Get current dynamic memory in use:
mem = MEMORY(/CURRENT)
; Prepare dialog message:
message = 'Current amount of dynamic memory used is '
sentence = message + STRTRIM(mem,2)+' bytes.'
; Display the dialog message containing memory usage statement:
status = DIALOG_MESSAGE (sentence, /INFORMATION)

```

Version History

Introduced: Original

See Also

[HELP](#)

MESH_CLIP

The MESH_CLIP function clips a polygonal mesh to an arbitrary plane in space and returns a polygonal mesh of the remaining portion. An auxiliary array of data may also be passed and clipped. This array can have multiple values for each vertex. The portion of the mesh below the plane, satisfying $ax+by+cz+d<0$, remains after the clipping operation.

Syntax

```
Result = MESH_CLIP (Plane, Vertsin, Connin, Vertsout, Connout  
[, AUXDATA_IN=array, AUXDATA_OUT=variable]  
[, CUT_VERTS=variable] )
```

Return Value

The return value is the number of triangles in the returned mesh.

Arguments

Plane

Input four element array describing the equation of the plane to be clipped to. The elements are the coefficients (a,b,c,d) of the equation $ax+by+cz+d=0$.

Vertsin

Input array of polygonal vertices $[3, n]$.

Connin

Input polygonal mesh connectivity array.

Vertsout

Output array of polygonal vertices.

Connout

Output polygonal mesh connectivity array.

Keywords

AUXDATA_IN

Input array of auxiliary data. If present, these values are interpolated and returned through AUXDATA_OUT. The trailing array dimension must match the number of vertices in the Vertsin array.

AUXDATA_OUT

Set this keyword to a named variable that will contain an output array of interpolated auxiliary data.

CUT_VERTS

Set this keyword to a named variable that will contain an output array of vertex indices (into Vertsout) of the vertices which are considered to be “on” the clipped surface.

Example

This example clips an octahedral mesh (an eight-sided, three-dimensional shape similar to a cut diamond). The original mesh contains one rectangle and eight triangles. The connectivity list is formed with the rectangle listed first followed by the triangles. The mesh is placed in a polygon object, which is added to a model. The model is displayed in the XOBJVIEW utility, which allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear. This display shows the mesh clipped with an oblique plane. The final XOBJVIEW display shows the results of using the TRIANGULATE routine to cover the clipped area. See [TRIANGULATE](#) in the *IDL Reference Guide* for more information in this routine.

```
PRO ClippingAMesh

; Create a mesh of an octahedron.
vertices = [[0, -1, 0], [1, 0, 0], [0, 1, 0], $
           [-1, 0, 0], [0, 0, 1], [0, 0, -1]]
connectivity = [4, 0, 1, 2, 3, 3, 0, 1, 4, 3, 1, 2, 4, $
              3, 2, 3, 4, 3, 3, 0, 4, 3, 1, 0, 5, 3, 2, 1, 5, $
              3, 3, 2, 5, 3, 0, 3, 5]
```

```

; Initialize model for display.
oModel = OBJ_NEW('IDLgrModel')

; Initialize polygon and polyline outline to contain
; the mesh of the octahedron.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, SHADING = 1, $
    COLOR = [0, 255, 0])
oPolyline = OBJ_NEW('IDLgrPolyline', vertices, $
    POLYLINES = connectivity, COLOR = [0, 0, 0])

; Add the polygon and the polyline to the model.
oModel->Add, oPolygon
oModel->Add, oPolyline

; Rotate model for better initial perspective.
oModel->Rotate, [-1, 0, 1], 22.5

; Display model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Original Octahedron Mesh'

; Clip mesh.
clip = MESH_CLIP([1., 1., 1., 0.], vertices, connectivity, $
    clippedVertices, clippedConnectivity, $
    CUT_VERTS = cutVerticesIndex)

; Update polygon with the resulting clipped mesh.
oPolygon->SetProperty, DATA = clippedVertices, $
    POLYGONS = clippedConnectivity

; Display the updated model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Clipped Octahedron Mesh'

; Determine the vertices of the clipped plane.
cutVertices = clippedVertices[* , cutVerticesIndex]

; Derive the x and y components of the clipped plane's
; vertices.
x = cutVertices[0, *]
y = cutVertices[1, *]

; Triangulate the connectivity of the clipped plane.
TRIANGULATE, x, y, triangles

; Derive the connectivity of the clipped plane from the
; results of the triangulation.
arraySize = SIZE(triangles, /DIMENSIONS)

```

```
array = FLTARR(4, arraySize[1])
array[0, *] = 3
array[1, 0] = triangles
cutConnectivity = REFORM(array, N_ELEMENTS(array))

; Initialize the clipped plane's polygon and polyline.
oCutPolygon = OBJ_NEW('IDLgrPolygon', cutVertices, $
    POLYGONS = cutConnectivity, SHADING = 1, $
    COLOR = [0, 0, 255])
oCutPolyline = OBJ_NEW('IDLgrPolyline', cutVertices, $
    POLYLINES = cutConnectivity, COLOR = [255, 0, 0], $
    THICK = 3.)

; Add polyline and polygon to model.
oModel->Add, oCutPolyline
oModel->Add, oCutPolygon

; Display updated model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Clipped Octahedron Mesh with Clipping Plane'

; Clean-up object references.
OBJ_DESTROY, [oModel]

END
```

The results for this example are shown in the following figure. The original octahedron is on the left and the two clipped results are shown to the right.

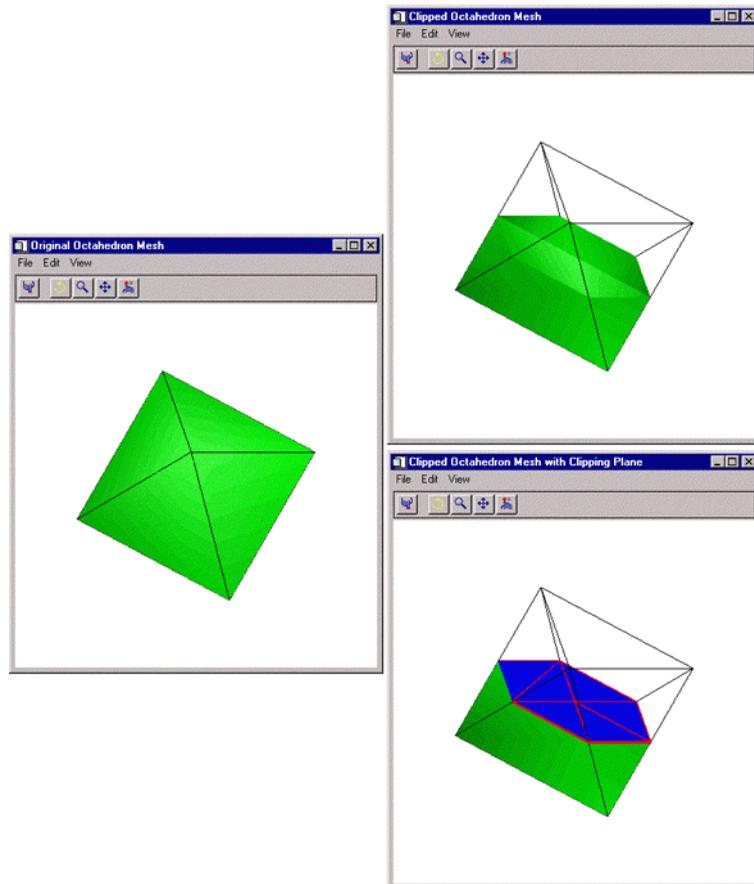


Figure 3-45: The Original Octahedron (left) and the Two Clipped Results (right)

Version History

Introduced: 5.5

See Also

[MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_MERGE](#),
[MESH_NUMTRIANGLES](#), [MESH_OBJ](#), [MESH_SMOOTH](#),
[MESH_SURFACEAREA](#), [MESH_VALIDATE](#), [MESH_VOLUME](#)

MESH_DECIMATE

The MESH_DECIMATE simplifies a polygonal mesh by combining neighboring polygons in the mesh in such a way that the shape of the original mesh is maintained as closely as possible. The decimation process is accomplished by locating the two most coplanar polygons, combining them by removing the edge between them, and then retriangulating the combined area. The process is repeated until the mesh contains the requested number of vertices or polygons. The resulting mesh approximates the original mesh with fewer vertices and polygons.

MESH_DECIMATE uses the algorithm described in the paper, "Surface Simplification Using Quadric Error Metrics" by Garland and Heckbert.

Decimation is a memory and CPU intensive process. Expect the decimation of large models to require large amounts of memory and dozens of seconds to complete. As a reference, a model with approximately 36,000 vertices and 70,000 faces requires 20-30 seconds to decimate to 10% of its original size on a typical Windows PC with 64Mb RAM and 333MHz Pentium processor.

MESH_DECIMATE converts the original mesh to a triangular mesh before performing decimation. The resulting mesh returned by MESH_DECIMATE is always a triangular mesh. For best results, all the polygons in the original mesh should be convex.

If the original mesh is not a triangular mesh, the mesh returned by MESH_DECIMATE might contain more triangles than the number of polygons in the original mesh, due to the conversion of the polygonal mesh to a triangular mesh. MESH_DECIMATE applies the PERCENT_POLYGONS keyword value to the number of triangles in the mesh after it is converted to a triangular mesh in order to perform the same amount of decimation.

MESH_DECIMATE uses fourth-order polynomials to locate the polygons it combines. Despite using double-precision in these calculations, mesh coordinate data that varies greatly in magnitude can cause MESH_DECIMATE to produce unacceptable results. For example, a 200x200 height field might have X and Y vertex coordinate data in the range [0-199] and Z data with a magnitude of about 10^{20} . In cases like these, the data should be offset or scaled so that all dimensions of the vertex data are of similar magnitude.

Syntax

```
Result = MESH_DECIMATE (Verts, Conn, Connout [, VERTICES=variable]
  [, PERCENT_VERTICES=percent | , PERCENT_POLYGONS=percent]
  [, PROGRESS_CALLBACK=string] [, PROGRESS_METHOD=string]
```

```
[, PROGRESS_OBJECT=objref] [, PROGRESS_PERCENT=percent{0 to 100}]  
[, PROGRESS_USERDATA=value] [, VERTICES=variable] )
```

Return Value

The return value is the number of triangles in the output connectivity array.

Arguments

Verts

Input array of polygonal vertices [3, *n*].

Conn

Input polygonal mesh connectivity array.

Connout

Output polygonal mesh connectivity array.

Note

Some of the vertices in the Verts array may not be referenced by the Connout array.

Keywords

PERCENT_VERTICES

Set this keyword to the percent of the original vertices to be returned in the Connout array. It specifies the amount of decimation to perform.

PERCENT_POLYGONS

Set this keyword to the percent of the original polygons to be returned in the Connout array. It specifies the amount of decimation to perform.

Note

PERCENT_VERTICES and PERCENT_POLYGONS are mutually exclusive keywords. If neither keyword is supplied, MESH_DECIMATE performs no decimation and returns the original mesh after converting it to a triangular mesh.

PROGRESS_CALLBACK

Set this keyword to a scalar string containing the name of the IDL function that MESH_DECIMATE calls at PROGRESS_PERCENT intervals as it decimates the polygonal mesh.

The PROGRESS_CALLBACK function returns a zero to signal MESH_DECIMATE to stop decimating, which causes MESH_DECIMATE to return the partially decimated mesh. If the callback function returns a non-zero, MESH_DECIMATE continues to decimate the mesh.

The PROGRESS_CALLBACK function must specify a single argument, *Percent*, which MESH_DECIMATE sets to an integer between 0 and 100, inclusive.

The PROGRESS_CALLBACK function may specify an optional USERDATA keyword, which MESH_DECIMATE sets to the variable provided in the PROGRESS_USERDATA keyword.

The following code show an example of a progress callback function:

```
FUNCTION myProgressCallback, percent,$      USERDATA = myStruct

  oProgressBar = myStruct.oProgressBar

  ; This method updates the progress bar
  ; graphic and returns TRUE if the user has
  ; NOT pressed the cancel button.
  keepGoing = oProgressBar->$
              UpdateProgressValue(percent)

  RETURN, keepGoing

END
```

PROGRESS_METHOD

Set this keyword to a scalar string containing the name of a function method that MESH_DECIMATE calls at PROGRESS_PERCENT intervals as it decimates the mesh. If this keyword is set, then the PROGRESS_OBJECT keyword must be set to an object reference that is an instance of a class that defines the specified method.

The PROGRESS_METHOD function method callback has the same specification as the callback described in the PROGRESS_CALLBACK keyword, except that it is defined as an object class method:

```
FUNCTION myClass::myProgressCallback, $
  percent, USERDATA = myStruct
```

PROGRESS_OBJECT

Set this keyword to an object reference that is an instance of a class that defines the method specified with the `PROGRESS_METHOD` keyword. If this keyword is set, then the `PROGRESS_METHOD` keyword must also be set.

PROGRESS_PERCENT

Set this keyword to a scalar in the range [1, 100] to specify the interval between invocations of the callback function. The default value is 5 and IDL silently clamps other values to the range [1, 100].

For example, a value of 5 (the default) specifies `MESH_DECIMATE` will call the callback function when the decimation is 0% complete, 5% complete, 10% complete, ..., 95% complete, and 100% complete.

PROGRESS_USERDATA

Set this property to any IDL variable that `MESH_DECIMATE` passes to the callback function in the callback function's `USERDATA` keyword parameter. If this keyword is specified, then the callback function must be able to accept keyword parameters.

VERTICES

Set this keyword to a named variable that will contain an output array of the vertices generated by the `MESH_DECIMATE` function. If this keyword is specified, the function is not restricted to using the original set of vertices specified in the `Verts` parameter when generating the decimated mesh, allowing it to generate a more optimal mesh by determining its own placement of vertices. If this keyword is specified, the `Connout` argument will consist of a polygon connectivity list whose indices refer to the vertex list stored in the named variable specified with this keyword. Otherwise, the `Connout` argument will consist of a polygon connectivity list that refers to the original vertex list `Verts`.

Examples

This example decimates a DEM (digital elevation model) mesh. The DEM in this example comes from the `elevbin.dat` file found in the `examples/data` directory. The DEM is converted into a mesh with the `MESH_OBJ` procedure. The results of this routine are placed in a polygon object, which is added to a model. The models are displayed in the `XOBJVIEW` utility. The `XOBJVIEW` utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

The first display contains a wire outline of the DEM as a mesh. When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear showing a filled mesh. The colors correspond to the change in elevation. The third display is the result of decimating the mesh down to 20 percent of the original number of vertices. The final display is the resulting mesh filled with the elevation colors.

```

PRO DecimatingAMesh

; Determine path to data file.
elevbinFile = FILEPATH('elevbin.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize data parameters.
elevbinSize = [64, 64]
elevbinData = BYTARR(elevbinSize[0], elevbinSize[1])

; Open file, read in data, and close file.
OPENR, unit, elevbinFile, /GET_LUN
READU, unit, elevbinData
FREE_LUN, unit

; Convert data into a mesh, which is defined by
; vertice locations and their connectivity.
MESH_OBJ, 1, vertices, connectivity, elevbinData

; Initialize a model for display.
oModel = OBJ_NEW('IDLgrModel')

; Form a polygon from the mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, SHADING = 1.5, $
    COLOR = [0, 255, 0], STYLE = 1)

; Add polygon to model.
oModel->Add, oPolygon

; Rotate model for better initial perspective.
oModel->Scale, 1, 1, 0.25
oModel->Rotate, [-1, 0, 1], 45.

; Display model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Original Mesh from Elevation Data'

; Derive a color table for the filled polygon.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette->LOADCT, 29

```

```
; Fill in the polygon mesh with the colors of the table
; (the colors correspond to the z-values of the polygon).
oPolygon->SetProperty, STYLE = 2, $
    VERT_COLORS = BYTSCL(vertices[2, *]), $
    PALETTE = oPalette

; Display model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Filled Original Mesh'

; Decimate the mesh down to 20 percent of the original
; number of vertices.
numberVertices = MESH_DECIMATE(vertices, connectivity, $
    decimatedConnectivity, VERTICES = decimatedVertices, $
    PERCENT_VERTICES = 20)

; Update the polygon with the resulting decimated mesh.
oPolygon->SetProperty, DATA = decimatedVertices, $
    POLYGONS = decimatedConnectivity, STYLE = 1, $
    VERT_COLORS = 0, COLOR = [0, 255, 0]

; Display updated model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Decimation Results (by 80%)'

; Fill in the updated polygon mesh with the colors of
; the table (the colors correspond to the z-values of
; the updated polygon).
oPolygon->SetProperty, STYLE = 2, $
    VERT_COLORS = BYTSCL(decimatedVertices[2, *]), $
    PALETTE = oPalette

; Display model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Filled Decimation Results'

; Cleanup all the objects by destroying the model.
OBJ_DESTROY, [oModel, oPalette]

END
```

The results of the decimation are shown in the bottom row of the following figure.

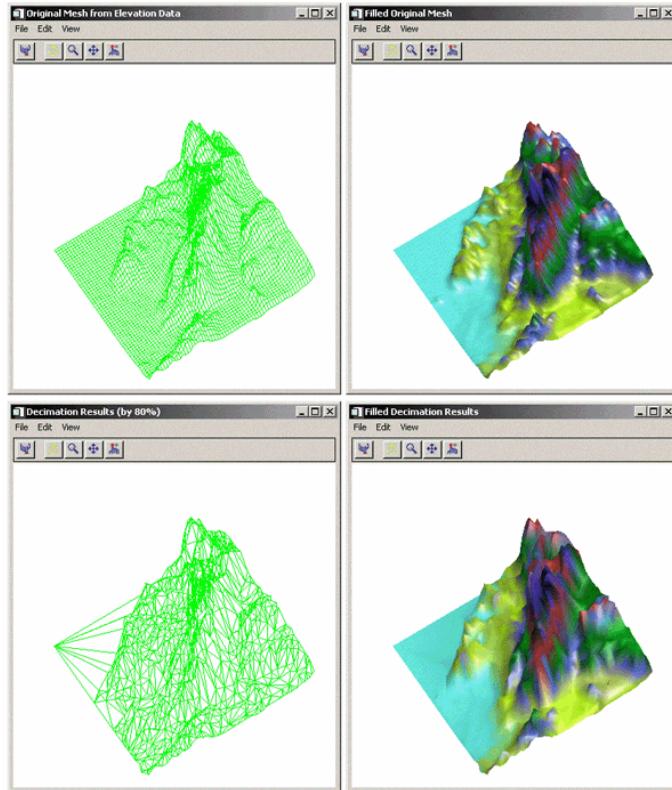


Figure 3-46: Before Decimating (top row) and After Decimating (bottom row)

Version History

Introduced: 5.5

PROGRESS_CALLBACK, PROGRESS_METHOD, PROGRESS_OBJECT,
PROGRESS_PERCENT, and PROGRESS_USERDATA keywords added: 6.0

See Also

[MESH_CLIP](#), [MESH_ISSOLID](#), [MESH_MERGE](#), [MESH_NUMTRIANGLES](#),
[MESH_OBJ](#), [MESH_SMOOTH](#), [MESH_SURFACEAREA](#), [MESH_VALIDATE](#),
[MESH_VOLUME](#)

MESH_ISSOLID

The MESH_ISSOLID function computes various mesh properties and enables IDL to determine if a mesh encloses space (is a solid). If the mesh can be considered a solid, routines can compute the volume of the mesh.

Syntax

Result = MESH_ISSOLID (*Conn*)

Return Value

Returns 1 if the input mesh fully encloses space (assuming no polygonal interpenetration) or 0 otherwise. A mesh is defined to fully enclose space if each edge in the input mesh appears an even number of times in the mesh.

Note

The input polygonal mesh is assumed to contain only planar, convex polygons.

Arguments

Conn

This is an integer or longword array that represents a series of polygon descriptions. Each polygon description takes the form $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0 \dots i_{n-1}$ are indices into the vertex array.

Keywords

None.

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_MERGE](#), [MESH_NUMTRIANGLES](#), [MESH_OBJ](#), [MESH_SMOOTH](#), [MESH_SURFACEAREA](#), [MESH_VALIDATE](#), [MESH_VOLUME](#)

MESH_MERGE

The MESH_MERGE function merges two polygonal meshes.

Syntax

```
Result = MESH_MERGE (Verts, Conn, Verts1, Conn1 [, /COMBINE_VERTICES]  
[, TOLERANCE=value])
```

Return Value

The function return value is the number of triangles in the modified polygonal mesh connectivity array.

Arguments

Verts

Input/Output array of polygonal vertices [3, *n*]. These are potentially modified and returned to the user.

Conn

Input/Output polygonal mesh connectivity array. This array is modified and returned to the user.

Verts1

Additional input polygonal vertex array [3, *n*].

Conn1

Additional input polygonal mesh connectivity array.

Keywords

COMBINE_VERTICES

If this keyword is set, the routine will attempt to collapse vertices which are at the same location in space into single vertices. If the expression

$$\max(|x_i - x_{i+1}|, |y_i - y_{i+1}|, |z_i - z_{i+1}|) < tolerance$$

is true, the points (i) and ($i+1$) can be collapsed into a single vertex. The result is returned as a modification of the *Verts* argument.

TOLERANCE

This keyword is used to specify the tolerance value used with the COMBINE_VERTICES keyword. The default value is 0.0.

Examples

This example merges two simple meshes: a single square and a single right triangle. The right side of the square is in the same location as the left side of the triangle. Each mesh is originally its own polygon object. These objects are then added to a model object. The model is displayed in the XOBJVIEW utility. The XOBJVIEW utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear. The meshes are merged into a single polygon object. After you quit out of the second display, the final display shows the results of decimating the merged mesh to obtain the least number connections for these vertices. Decimation can often be used to refine the results of merging.

```
PRO MergingMeshes

; Create a mesh of a single square (4 vertices
; connected counter-clockwise from the lower left
; corner of the mesh.
vertices = [[-2., -1., 0.], [0., -1., 0.], $
           [0., 1., 0.], [-2., 1., 0.]]
connectivity = [4, 0, 1, 2, 3]
```

```

; Create a separate mesh of a single triangle (3
; vertices connected counter-clockwise from the lower
; left corner of the mesh.
triangleVertices = [[0., -1., 0.], [2., -1., 0.], $
  [0., 1., 0.]]
triangleConnectivity = [3, 0, 1, 2]

; Initialize model for display.
oModel = OBJ_NEW('IDLgrModel')

; Initialize polygon for the square mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
  POLYGONS = connectivity, COLOR = [0, 128, 0], $
  STYLE = 1)

; Initialize polygon for the triangle mesh.
oTrianglePolygon = OBJ_NEW('IDLgrPolygon', $
  triangleVertices, POLYGONS = triangleConnectivity, $
  COLOR = [0, 0, 255], STYLE = 1)

; Add both polygons to the model.
oModel->Add, oPolygon
oModel->Add, oTrianglePolygon

; Display the model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, $
  TITLE = 'Two Separate Meshes'

; Merge the square and triangle into a single mesh.
numberTriangles = MESH_MERGE(vertices, $
  connectivity, triangleVertices, $
  triangleConnectivity, /COMBINE_VERTICES)

; Output number of resulting vertices and triangles.
numberVertices = SIZE(vertices, /DIMENSIONS)
PRINT, 'numberVertices = ', numberVertices[1]
PRINT, 'numberTriangles = ', numberTriangles

; Cleanup triangle polygon object, which is no longer
; needed.
OBJ_DESTROY, [oTrianglePolygon]

; Update remaining polygon object with the results from
; merging the two meshes together.
oPolygon->SetProperty, DATA = vertices, $
  POLYGONS = connectivity, COLOR = [0, 128, 128]

```

```

; Display results.
XOBJVIEW, oModel, /BLOCK, $
  TITLE = 'Result of Merging the Meshes into One'

; Decimate polygon to 75 percent of the original
; number of vertices.
numberTriangles = MESH_DECIMATE(vertices, connectivity, $
  decimatedConnectivity, PERCENT_POLYGONS = 75)

; Output number of resulting triangles.
PRINT, 'After Decimation: numberTriangles = ', numberTriangles

; Update polygon with results from decimating.
oPolygon->SetProperty, DATA = vertices, $
  POLYGONS = decimatedConnectivity, COLOR = [0, 0, 0]

; Display decimation results.
XOBJVIEW, oModel, /BLOCK, $
  TITLE = 'Decimation of Mesh'

; Cleanup object references.
OBJ_DESTROY, [oModel]

END

```

The results for this example are shown in the following figure: original, separate meshes (left), merged mesh (center) and decimated mesh (right).

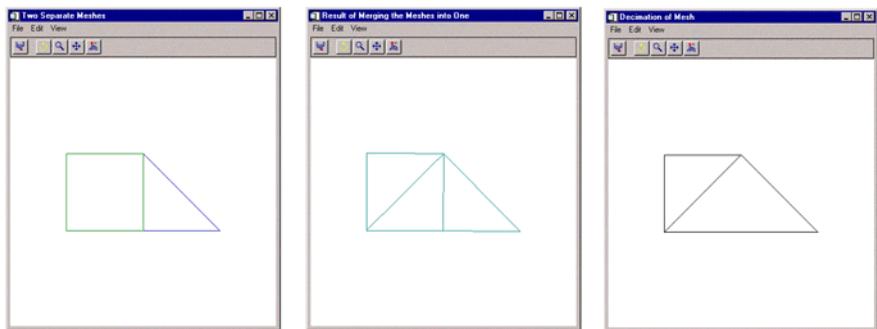


Figure 3-47: Original (left), Merged (center), and Decimated Meshes (right)

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_NUMTRIANGLES](#),
[MESH_OBJ](#), [MESH_SMOOTH](#), [MESH_SURFACEAREA](#), [MESH_VALIDATE](#),
[MESH_VOLUME](#)

MESH_NUMTRIANGLES

The MESH_NUMTRIANGLES function computes the number of triangles in a polygonal mesh.

Syntax

Result = MESH_NUMTRIANGLES (*Conn*)

Return Value

Returns the number of triangles in the mesh (a quad is considered two triangles).

Arguments

Conn

Polygonal mesh connectivity array.

Keywords

None.

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_MERGE](#),
[MESH_OBJ](#), [MESH_SMOOTH](#), [MESH_SURFACEAREA](#), [MESH_VALIDATE](#),
[MESH_VOLUME](#)

MESH_OBJ

The MESH_OBJ procedure generates a polygon mesh (vertex list and polygon list) that represent the desired primitive object. The available primitive objects are: triangulated surface, rectangular surface, polar surface, cylindrical surface, spherical surface, surface of extrusion, surface of revolution, and ruled surface.

This routine is written in the IDL language. Its source code can be found in the file `mesh_obj.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
MESH_OBJ, Type, Vertex_List, Polygon_List, Array1 [, Array2] [, /CLOSED]
      [, /DEGREES] [, P1 - P5=value]
```

Arguments

Type

An integer that specifies what type of object to create. The various surface types are described in the table below.

Type	Surface Type
0	Triangulated
1	Rectangular
2	Polar
3	Cylindrical
4	Spherical
5	Extrusion
6	Revolution
7	Ruled
Other values	None

Table 3-86: Surface Types

Vertex_List

A named variable that will contain the mesh vertices. *Vertex_List* has the same format as the lists returned by the SHADE_VOLUME procedure.

Polygon_List

A named variable that will contain the mesh indexes. *Polygon_List* has the same format as the lists returned by the SHADE_VOLUME procedure.

Array1

An array whose use depends on the type of object being created. The following table describes the differences.

Surface Type	Array1 Type
Triangulated	A (3, n) array containing random $[x, y, z]$ points to build a triangulated surface from. The resulting polygon mesh will have n vertices. When shading a triangulated mesh, the shading array should have (n) elements.
Rectangular	A two dimensional (n, m) array containing z values. The resulting polygon mesh will have $n \times m$ vertices. When shading a rectangular mesh, the shading array should have (n, m) elements.
Polar	A two dimensional (n, m) array containing z values. The resulting polygon mesh will have $n \times m$ vertices. The n dimension of the array is mapped to the polar angle, and the m dimension is mapped to the polar radius. When shading a polar mesh, the shading array should have (n, m) elements.
Cylindrical	A two dimensional (n, m) array containing radius values. The resulting polygon mesh will have $n \times m$ vertices. The n dimension of the array is mapped to the polar angle, and the m dimension is mapped to the Z axis. When shading a cylindrical mesh, the shading array should have (n, m) elements.

Table 3-87: Array 1 Type

Surface Type	Array1 Type
Spherical	A two dimensional (n, m) array containing radius values. The resulting polygon mesh will have $n \times m$ vertices. The n dimension of the array is mapped to the longitude (0.0 to 360.0 degrees), and the m dimension is mapped to the latitude (-90.0 to +90.0 degrees). When shading a spherical mesh, the shading array should have (n, m) elements.
Extrusion	A ($3, n$) array of connected 3D points which define the shape to extrude. The resulting polygon mesh will have $n \times (\text{steps}+1)$ vertices (where steps is the number of “segments” in the extrusion). (See the P1 keyword). If the order of the elements in <i>Array1</i> is reversed, then the polygon facing is reversed. When shading an extrusion mesh, the shading array should have ($n, \text{steps}+1$) elements.
Revolution	A ($3, n$) array of connected 3D points which define the shape to revolve. The resulting polygon mesh will have $n \times ((\text{steps}>3)+1)$ vertices (where steps is the number of “steps” in the revolution). (See the P1 keyword). If the order of the elements in <i>Array1</i> is reversed, then the polygon facing is reversed. When shading a revolution mesh, the shading array should have ($n, (\text{steps}>3)+1$) elements.
Ruled	A ($3, n$) array of connected 3D points which define the shape of the first ruled vector. The optional ($3, m$) <i>Array2</i> parameter defines the shape of the second ruled vector. The resulting polygon mesh will have $(n > m) \times (\text{steps}+1)$ vertices (where steps is the number of intermediate “steps”). (See the P1 keyword). When shading a ruled mesh, the shading array should have ($n > m, \text{steps}+1$) elements.

Table 3-87: Array 1 Type (Continued)

Array2

If the object type is 7 (Ruled Surface) then *Array2* is a ($3, m$) array containing the 3D points which define the second ruled vector. If *Array2* has fewer elements than *Array1* then *Array2* is processed with CONGRID to give it the same number of elements as *Array1*. If *Array1* has fewer elements than *Array2* then *Array1* is processed with CONGRID to give it the same number of elements as *Array2*. *Array2* must be supplied if the object type is 7. Otherwise, *Array2* is ignored.

Keywords

CLOSED

Set this keyword to “close” the polygonal mesh topologically by using the first vertex in a given row for both the first and last polygons in that row. This keyword is only applicable to the CYLINDRICAL, SPHERICAL, REVOLUTION, and EXTRUSION surface types. Setting the CLOSED keyword removes the discontinuity when the mesh wraps back around on itself, which can improve the mesh’s appearance when viewing it as a shaded object. For the EXTRUSION surface type, this procedure handles input polygons that form a closed loop with the last vertex being a copy of the first vertex, as well as those that do not.

DEGREES

If set, then the input parameters are in degrees (where applicable). Otherwise, the angles are in radians.

P1 - P5

The meaning of the keywords P1 through P5 vary depending upon the object type. The table below describes the differences.

Surface Type	Keywords
Triangulated	P1 through P5 are ignored.
Rectangular	If <i>Array1</i> is an (n, m) array, and if P1 has n elements, then the values contained in P1 are the X coordinates for each column of vertices. Otherwise, FINDGEN(n) is used for the X coordinates. If P2 has m elements, then the values contained in P2 are the Y coordinates for each row of vertices. Otherwise, FINDGEN(m) is used for the Y coordinates. The polygon facing is reversed if the order of either P1 or P2 (but not both) is reversed. P3, P4, and P5 are ignored.

Table 3-88: P1-P5 Keywords

Surface Type	Keywords
Polar	P1 specifies the polar angle of the first column of <i>Array1</i> (the default is 0). P2 specifies the polar angle of the last column of <i>Array1</i> (the default is $2*PI$). If P2 is less than P1 then the polygon facing is reversed. P3 specifies the radius of the first row of <i>Array1</i> (the default is 0). P4 specifies the radius of the last row of <i>Array1</i> (the default is $m-1$). If P4 is less than P3 then the polygon facing is reversed. P5 is ignored.
Cylindrical	P1 specifies the polar angle of the first column of <i>Array1</i> (the default is 0). P2 specifies the polar angle of the last column of <i>Array1</i> (the default is $2*PI$). If P2 is less than P1 then the polygon facing is reversed. P3 specifies the Z coordinate of the first row of <i>Array1</i> (the default is 0). P4 specifies the Z coordinate of the last row of <i>Array1</i> (the default is $m-1$). If P4 is less than P3 then the polygon facing is reversed. P5 is ignored.
Spherical	P1 specifies the longitude of the first column of <i>Array1</i> (the default is 0). P2 specifies the longitude of the last column of <i>Array1</i> (the default is $2*PI$). If P2 is less than P1 then the polygon facing is reversed. P3 specifies the latitude of the first row of <i>Array1</i> (the default is $-PI/2$). P4 specifies the latitude of the last row of <i>Array1</i> (the default is $+PI/2$). If P4 is less than P3 then the polygon facing is reversed. P5 is ignored.
Extrusion	P1 specifies the number of steps in the extrusion (the default is 1). P2 is a three element vector specifying the direction (and length) of the extrusion (the default is $[0, 0, 1]$). P3, P4, and P5 are ignored.
Revolution	P1 specifies the number of “facets” in the revolution (the default is 3). If P1 is less than 3 then 3 is used. P2 is a three element vector specifying a point that the rotation vector passes through (the default is $[0, 0, 0]$). P3 is a three element vector specifying the direction of the rotation vector (the default is $[0, 0, 1]$). P4 specifies the starting angle for the revolution (the default is 0). P5 specifies the ending angle for the revolution (the default is $2*PI$). If P5 is less than P4 then the polygon facing is reversed.

Table 3-88: P1-P5 Keywords (Continued)

Surface Type	Keywords
Ruled	P1 specifies the number of “steps” in the ruling (the default is 1). P2, P3, P4, and P5 are ignored.

Table 3-88: P1-P5 Keywords (Continued)

Examples

```

; Create a 48x64 cylinder with a constant radius of 0.25:
MESH_OBJ, 3, Vertex_List, Polygon_List, $
    Replicate(0.25, 48, 64), P4=0.5

; Transform the vertices:
T3D, /RESET
T3D, ROTATE=[0.0, 30.0, 0.0]
T3D, ROTATE=[0.0, 0.0, 40.0]
T3D, TRANSLATE=[0.25, 0.25, 0.25]
VERTEX_LIST = VERT_T3D(Vertex_List)

; Create the window and view:
WINDOW, 0, XSIZE=512, YSIZE=512
CREATE_VIEW, WINX=512, WINY=512

; Render the mesh:
SET_SHADING, LIGHT=[-0.5, 0.5, 2.0], REJECT=0
TVSCL, POLYSHADE(Vertex_List, Polygon_List, /NORMAL)

; Create a cone (surface of revolution):
MESH_OBJ, 6, Vertex_List, Polygon_List, $
    [[0.75, 0.0, 0.25], [0.5, 0.0, 0.75]], $
    P1=16, P2=[0.5, 0.0, 0.0]

; Create the window and view:
WINDOW, 0, XSIZE=512, YSIZE=512
CREATE_VIEW, WINX=512, WINY=512, AX=30.0, AY=(140.0), ZOOM=0.5

; Render the mesh:
SET_SHADING, LIGHT=[-0.5, 0.5, 2.0], REJECT=0
TVSCL, POLYSHADE(Vertex_List, Polygon_List, /DATA, /T3D)

```

Version History

Introduced: Pre 4.0

See Also

[CREATE_VIEW](#), [MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#),
[MESH_MERGE](#), [MESH_NUMTRIANGLES](#), [MESH_SMOOTH](#),
[MESH_SURFACEAREA](#), [MESH_VALIDATE](#), [MESH_VOLUME](#),
[SET_SHADING](#), [VERT_T3D](#)

MESH_SMOOTH

The MESH_SMOOTH function performs spatial smoothing on a polygon mesh. This function smooths a mesh by applying Laplacian smoothing to each vertex, as described by the following formula:

$$\vec{x}_{i_{(n+1)}} = \vec{x}_{i_n} + \frac{\lambda}{M} \sum_{j=0}^M (\vec{x}_{j_n} - \vec{x}_{i_n})$$

where:

- \vec{x}_{i_n} is vertex i for iteration n
- λ is the smoothing factor
- M is the number of vertices that share a common edge with x_{i_n} .

Syntax

```
Result = MESH_SMOOTH (Verts, Conn [, ITERATIONS=value]
  [, FIXED_VERTICES=array] [, /FIXED_EDGE_VERTICES]
  [, LAMBDA=value])
```

Return Value

The output of this function is resulting $[3, n]$ array of modified vertices.

Arguments

Verts

Input array of polygonal vertices $[3, n]$.

Conn

Input polygonal mesh connectivity array.

Keywords

ITERATIONS

Number of iterations to smooth. The default value is 50.

FIXED_VERTICES

Set this keyword to an array of vertex indices which are not to be modified by the smoothing.

FIXED_EDGE_VERTICES

Set this keyword to specify that mesh outer edge vertices are not to be modified by the smoothing.

LAMBDA

Smoothing factor. The default value is 0.05.

Examples

This example smooths a rectangular mesh containing a spike. First, we create a rectangular mesh made up 10 columns and 5 rows of vertices. The vertices are connected with right triangles. The mesh is placed in a polygon object, which is added to a model object. The model is displayed in the XOBJVIEW utility. The XOBJVIEW utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear. The center vertex of the top row is displaced in the y-direction. This displacement causes the center of the top to spike out away from the mesh. After you quit out of the second display, the third display shows the result of smoothing the entire mesh. The final display shows the results of smoothing the spike with all the other vertices fixed.

```
PRO SmoothingMeshes

; Initialize mesh size parameters.
nX = 10
nY = 5

; Initialize the x coordinates of the mesh's vertices.
xVertices = FINDGEN(nX) # REPLICATE(1., nY)
```

```

PRINT, 'xVertices: '
PRINT, xVertices, FORMAT = '(10F6.1)'

; Initialize the y coordinates of the mesh's vertices.
yVertices = REPLICATE(1., nX) # FINDGEN(nY)
PRINT, 'yVertices: '
PRINT, yVertices, FORMAT = '(10F6.1)'

; Derive the overall vertices of the mesh.
vertices = FLTARR(3, (nX*nY))
vertices[0, *] = xVertices
vertices[1, *] = yVertices
PRINT, 'vertices: '
PRINT, vertices, FORMAT = '(3F6.1)'

; Triangulate the mesh to establish connectivity.
TRIANGULATE, xVertices, yVertices, triangles
trianglesSize = SIZE(triangles, /DIMENSIONS)
polygons = LONARR(4, trianglesSize[1])
polygons[0, *] = 3
polygons[1, 0] = triangles
PRINT, 'polygons: '
PRINT, polygons, FORMAT = '(4I6)'

; Derive connectivity from the resulting triangulation.
connectivity = REFORM(polygons, N_ELEMENTS(polygons))

; Initialize a model for the display.
oModel = OBJ_NEW('IDLgrModel')

; Initialize a polygon object to contain the mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, COLOR = [0, 128, 0], $
    STYLE = 1)

; Add the polygon to the model.
oModel->Add, oPolygon

; Display the model.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Original Mesh'

; Introduce an irregular vertex by drastically changing
; a single y coordinate.
vertices[1, 45] = 10.

; Update polygon with new vertices.
oPolygon->SetProperty, DATA = vertices

```

```
; Display change.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Mesh with New Irregular Vertex'

; Smooth entire mesh to reduce the effect of the
; irregular vertex.
smoothedVertices = MESH_SMOOTH(vertices, connectivity)

; Update polygon and display results.
oPolygon->SetProperty, DATA = smoothedVertices
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Smoothing with No Fixed Vertices'

; Determine which vertices should be fixed. Basically,
; all of the vertices should be fixed except for the
; irregular vertex.
fixed = LINDGEN((nX*nY) - 1)
fixed[45] = fixed[45:*] + 1

; Smooth mesh with resulting fixed vertices.
smoothedVertices = MESH_SMOOTH(vertices, connectivity, $
    FIXED_VERTICES = fixed)

; Update polygon and display results.
oPolygon->SetProperty, DATA = smoothedVertices
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Smoothing with Almost All Vertices Fixed'

; Cleanup object references.
OBJ_DESTROY, [oModel]

END
```

The results for this example are shown in the following figure: the spiked mesh (left), and the two smoothed meshes (right).

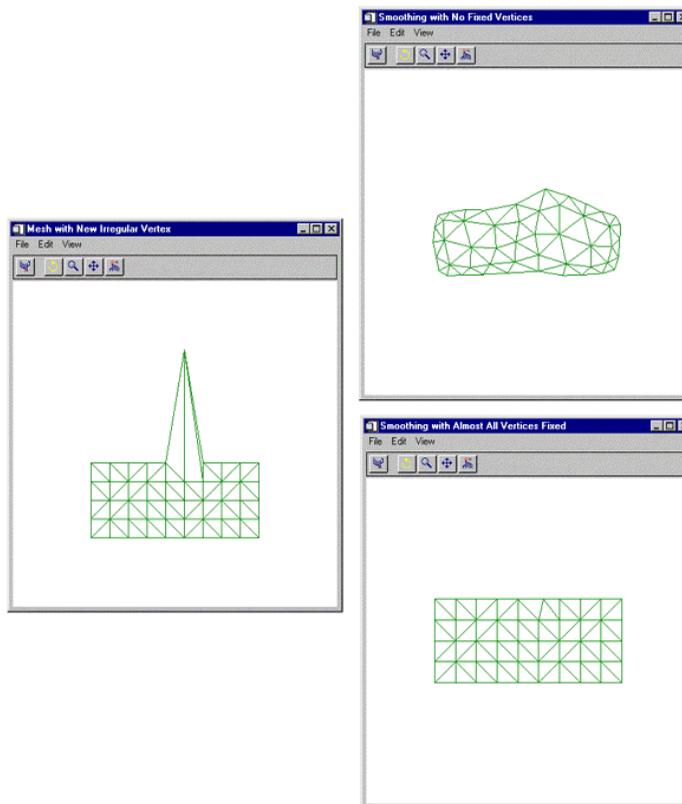


Figure 3-48: The Spiked Mesh (left) and the Two Smoothed Meshes (right)

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_MERGE](#),
[MESH_NUMTRIANGLES](#), [MESH_OBJ](#), [MESH_SURFACEAREA](#),
[MESH_VALIDATE](#), [MESH_VOLUME](#)

MESH_SURFACEAREA

The MESH_SURFACEAREA function computes various mesh properties to determine the mesh surface area, including integration of other properties interpolated on the surface of the mesh.

Syntax

```
Result = MESH_SURFACEAREA ( Verts, Conn [, AUXDATA=array]  
[, MOMENT=variable] )
```

Return Value

Returns the cumulative (weighted) surface area of the polygons in the mesh.

Note

The input polygonal mesh is assumed to contain only planar, convex polygons.

Arguments

Verts

Array of polygonal vertices [3, *n*].

Conn

Polygonal mesh connectivity array.

Keywords

AUXDATA

Array of input auxiliary data (one value per vertex). If present, these values are used to weight a vertex for the purpose of the area computation. The surface area integral will linearly interpolate these values over the surface of each triangle. The default weight is 1.0 which results in the basic polygon area.

MOMENT

If this keyword is present, it will return a three element float vector which corresponds to the first order moments computed with respect to the X, Y and Z axis. The computation is:

$$\vec{m} = \sum_{ntris} a_i \vec{c}_i$$

where a is the (weighted) area of the triangle and c is the centroid of the triangle, thus

$$\vec{m}/sarea$$

yields the (weighted) centroid of the polygon mesh.

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_MERGE](#),
[MESH_NUMTRIANGLES](#), [MESH_OBJ](#), [MESH_SMOOTH](#), [MESH_VALIDATE](#),
[MESH_VOLUME](#)

MESH_VALIDATE

The MESH_VALIDATE function checks for NaN values in vertices, removes unused vertices, and combines close vertices.

Syntax

```
Result = MESH_VALIDATE ( Verts, Conn [, /REMOVE_NAN]
                        [, /PACK_VERTICES] [, /COMBINE_VERTICES] [, TOLERANCE=value] )
```

Return Value

The function return value is the number of triangles in the modified polygonal mesh connectivity array.

Arguments

Verts

Input/Output array of polygonal vertices [3, *n*]. These are potentially modified and returned to the user.

Conn

Input/Output polygonal mesh connectivity array. This array is modified and returned to the user.

Keywords

COMBINE_VERTICES

If this keyword is set, the routine will attempt to collapse vertices which are at the same location in space into single vertices. If the expression

$$\max(|x_i - x_{i+1}|, |y_i - y_{i+1}|, |z_i - z_{i+1}|) < tolerance$$

is true, the points (*i*) and (*i*+1) can be collapsed into a single vertex. The result is returned as a modification of the *Verts* argument.

PACK_VERTICES

If this keyword is set, the Verts input array will be packed to exclude any non-referenced vertices. The result is returned in the Verts argument.

REMOVE_NAN

If this keyword is set, the function will remove any polygons from CONN which reference vertices containing NaN values.

TOLERANCE

This keyword is used to specify the tolerance value used with the COMBINE_VERTS keyword. The default value is 0.0.

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_MERGE](#),
[MESH_NUMTRIANGLES](#), [MESH_OBJ](#), [MESH_SMOOTH](#),
[MESH_SURFACEAREA](#), [MESH_VOLUME](#)

MESH_VOLUME

The MESH_VOLUME function computes the volume that the mesh encloses. A region that a mesh encloses has a positive value for its volume when it is enclosed by mesh polygons that face outward from the enclosed region. Outward-facing polygons follow the convention of their vertices being ordered in a counter-clockwise direction while observing the polygon from the outside of the enclosed region. Likewise, a region has a negative value for its volume when it is enclosed by polygons that face inward to the enclosed region.

A single mesh may contain regions that have positive and negative volume values. This function adds these signed values together to produce a single volume value that takes into account the total of all positive regions minus any "holes" or subtractions specified by the negative regions.

If the SIGNED keyword is not specified, IDL returns the absolute value of the volume, which may be useful in situations where the polygon vertex ordering convention is unknown or opposite of the convention described above.

Syntax

Result = MESH_VOLUME (*Verts*, *Conn* [, /SIGNED])

Return Value

Returns the volume that the mesh encloses. If the mesh does not enclose space (i.e. MESH_ISSOLID() would return 0), this function returns 0.0.

Note

The input polygonal mesh is assumed to contain only planar, convex polygons.

Arguments

Verts

Array of polygonal vertices [3, *n*].

Conn

Polygonal mesh connectivity array.

Keywords

SIGNED

Set this keyword to compute the signed volume. The sign will be negative for a mesh consisting of inward facing polygons.

Version History

Introduced: 5.5

See Also

[MESH_CLIP](#), [MESH_DECIMATE](#), [MESH_ISSOLID](#), [MESH_MERGE](#),
[MESH_NUMTRIANGLES](#), [MESH_OBJ](#), [MESH_SMOOTH](#),
[MESH_SURFACEAREA](#), [MESH_VALIDATE](#)

MESSAGE

The MESSAGE procedure issues error and informational messages using the same mechanism employed by built-in IDL routines. By default, the message is issued as an error, the message is output, and IDL performs the required error-handling actions (which can be controlled via the CATCH, ON_ERROR, and ON_IOERROR procedures). As a side-effect of issuing the error, the system variable !ERROR_STATE is set and the text of the error message is placed in !ERROR_STATE.MSG. (If there is an operating system component to the error message, !ERROR_STATE.SYS_MSG is updated as well).

The MESSAGE procedure supports the following uses:

1. To issue a simple error message containing user-specified text.
2. To issue a specific error from a *message block* by name, with optional arguments. The NAME keyword is required in this case; if the error is not defined in the default IDL message block, the BLOCK keyword is also required. See [DEFINE_MSGBLK](#) and [DEFINE_MSGBLK_FROM_FILE](#) for examples of this usage.
3. To reissue the most recent error encountered by IDL. If the CATCH procedure is used to trap errors, the REISSUE_LAST keyword to MESSAGE can be used within the CATCH block to pass the error up to its caller.

If the call to the MESSAGE procedure causes execution to halt, traceback information is displayed automatically.

Syntax

To issue a simple error message:

```
MESSAGE, [Text] [, /CONTINUE] [, LEVEL=CallLevel] [, /INFORMATIONAL]
      [, /IOERROR] [, /NONAME] [, /NOPREFIX] [, /NOPRINT] [, /RESET]
```

To issue a named message from a message block:

```
MESSAGE, [Arg1, ... ArgN] NAME=ErrorName [, BLOCK=BlockName]
      [, /CONTINUE] [, LEVEL=CallLevel] [, /INFORMATIONAL] [, /IOERROR]
      [, /NONAME] [, /NOPREFIX] [, /NOPRINT] [, /RESET]
```

To reissue the most recent error:

```
MESSAGE, /REISSUE_LAST
```

Arguments

Text

A string value specifying the text to be displayed in a simple error message.

Note

The *Text* argument is only used when MESSAGE is issuing a simple error message; that is, if neither the NAME keyword nor the REISSUE_LAST keyword is present. If *none* of the *Text* argument, the NAME keyword, or the REISSUE_LAST keyword are present, MESSAGE returns quietly.

Arg_{*i*}

When issuing a named error using the NAME (and possibly BLOCK) keyword, the *Arg_{*i*}* arguments are substituted into the error format string, as described in the documentation for [DEFINE_MSGBLK](#) and [DEFINE_MSGBLK_FROM_FILE](#).

Keywords

BLOCK

Set this keyword to a string containing the name of the IDL message block to use. This keyword is ignored unless the NAME keyword is also present to specify a message name.

By default, MESSAGE throws the IDL_M_USER_ERR message from the IDL_MBLK_CORE message block. If you wish to provide something other than the default error message, you can define your own message blocks and error messages. See the [DEFINE_MSGBLK](#) and [DEFINE_MSGBLK_FROM_FILE](#) procedures for details. You can use the HELP, /MESSAGES command to see the currently defined message blocks.

CONTINUE

Set this keyword to return after issuing the error instead of taking the action specified by ON_ERROR. Use this option when it is desirable to report an error and then continue processing.

INFORMATIONAL

Set this keyword to issue informational text instead of an error. In this case, !ERROR_STATE is not set. The !QUIET system variable controls the printing of informational messages.

IOERROR

Set this keyword to indicate that the error occurred while performing I/O. The action specified by the ON_IOERROR procedure is executed instead of ON_ERROR.

LEVEL

Many messages include the name of the routine that called MESSAGE at the beginning of the message text. Use the LEVEL keyword to an integer value to specify that the name of a routine further up in the current call chain should be used instead. Specify the value of LEVEL as described in the following table:

LEVEL value	Meaning
0	The currently active routine.
> 0	The absolute index of the routine to indicate. A value of 1 specifies the main level (\$MAIN\$), 2 indicates the routine called by \$MAIN\$, and so forth.
< 0	Negative values indicate the relative index of the desired routine moving backwards from the current one. Hence, -1 indicates the caller of the current routine.

Table 3-89: LEVEL Keyword Values

The LEVEL keyword can be used to hide error handling helper routines from user view. The following procedure will issue an error on behalf of its caller. The calling routine's name will appear in the resulting message, and not that of the error routine:

```

pro THROW_ERROR, text
  on_error, 2 ; Stop in caller
  MESSAGE, LEVEL=-1, text
end

```

NAME

Set this keyword to a string containing the name of the message throw. By default, MESSAGE throws the IDL_M_USER_ERR message from the IDL_MBLK_CORE

message block. NAME is often used in conjunction with the BLOCK keyword to throw a non-default message from a non-default message block.

NONAME

Set this keyword to suppress printing of the issuing routine's name at the beginning of the error message.

NOPREFIX

Usually, the message includes the message prefix string (as specified by the MSG_PREFIX field of the !ERROR_STATE system variable) at the beginning. Set this keyword to omit the prefix.

NOPRINT

Set this keyword to prevent the message from printing to the screen and cause the other actions to proceed quietly. The error system variables are updated as usual.

REISSUE_LAST

Set this keyword to reissue the last error issued by IDL. By using this keyword in conjunction with the CATCH procedure, your code can catch an error caused by called code, perform recovery actions, and issue the error normally. See the [Examples](#) below for a demonstration of this approach.

Note

If this keyword is specified, no plain arguments or other keywords may be specified.

RESET

Set this keyword to set the !ERROR_STATE system variable back to the "success" state and clear any internal traceback information being saved for use by the LAST_ERROR keyword to the [HELP](#) procedure.

TRACEBACK

This keyword is obsolete and is included for compatibility with existing code only. Traceback information is provided by default.

Examples

Example 1

As an example, assume the statement:

```
message, 'Unexpected value encountered.'
```

is executed in a procedure named `CALC`. If an error occurs, the following message would be printed:

```
% CALC: Unexpected value encountered.
```

and execution would halt.

Example 2

This example demonstrates the use of the `CATCH` procedure and the `REISSUE_LAST` keyword to the `MESSAGE` procedure to control errors. In this example, we write a procedure named `GET_TWO_POINTERS`, which creates and returns two image variables of identical size via pointer heap variables. One possible problem with such an operation is that the system may not have enough memory to allocate both images. We want this operation to be all or nothing, so if we fail to get both variables we need to free the variable we did get before allowing our caller to see the error:

```
PRO GET_TWO_POINTERS, D1, D2, P1, P2
; [D1, D2] - Input dimensions
; P1, P2 - Variables to receive pointers to images

ON_ERROR, 2                                ; When we reissue error, have
                                           ; control returned to caller.

nullPtr = PTR_NEW()                        ; Create a NULL pointer.
P1 = (P2 = nullPtr)                        ; Set both pointers to NULL.

CATCH, error                               ; Establish catch block.
IF (error NE 0) THEN BEGIN                 ; An error occurs.
    CATCH, /CANCEL                         ; Cancel catch block so an error
                                           ; here will not cause looping.
    PTR_FREE, P1, P2                      ; If P1 or P2 are non-NULL, free
    P1 = (P2 = nullPtr)                   ; them so caller sees NULL pointers.
    MESSAGE, /REISSUE_LAST                ; Reissue the error. The caller
                                           ; will get control.
    ; This line is never reached, because MESSAGE causes an
    ; implicit return to the calling routine.
ENDIF
```

```
P1 = PTR_NEW(BYTARR(D1, D2)) ; Get first image.  
P2 = PTR_NEW(BYTARR(D1, D2)) ; Get second image.  
  
; We now have both images and can safely return.  
END
```

Version History

Introduced: Pre 4.0

REISSUE_LAST keyword added: 6.0

See Also

[CATCH](#), [DEFINE_MSGBLK](#), [DEFINE_MSGBLK_FROM_FILE](#), [ON_ERROR](#),
[ON_IOERROR](#), [STRMESSAGE](#)

MIN

The MIN function returns the value of the smallest element of *Array*. The type of the result is the same as that of *Array*.

Syntax

```
Result = MIN( Array [, Min_Subscript] [, /ABSOLUTE] [, DIMENSION=value]  
            [, MAX=variable] [, /NAN] [, SUBSCRIPT_MAX=variable])
```

Return Value

Returns the smallest array element value.

Arguments

Array

The array to be searched.

Min_Subscript

A named variable that, if supplied, is converted to a long integer containing the one-dimensional subscript of the minimum element. Otherwise, the system variable !C is set to the one-dimensional subscript of the minimum element.

Keywords

ABSOLUTE

If this keyword is set, then the absolute value of each element is used when determining the minimum values. This keyword has no effect for arrays of type byte or unsigned integer.

Note

If ABSOLUTE is set then the sign of each value is ignored when searching for the minimum. However, the return value retains the negative sign if the value was indeed negative.

Tip

For complex input, MIN by default only compares the real component of each value. Use the ABSOLUTE keyword to force MIN to compare the absolute value of

each value, and to return the complex value corresponding to the minimum absolute value.

DIMENSION

Set this keyword to the dimension over which to find the minimum values of an array. If this keyword is not present or is zero, the minimum is found over the entire array and is returned as a scalar value. If this keyword is present and nonzero, the result is the “slice” of the input array that contains the minimum value element, and the return values for *Result*, *Min_Subscript*, *MAX*, and *SUBSCRIPT_MAX* will all be arrays of one dimension less than the input array. That is, if the dimensions of *Array* are *N1*, *N2*, *N3*, and *DIMENSION=2*, the dimensions of the result are (*N1*, *N3*), and element (*i,j*) of the result contains the minimum value of *Array*[*i*, *, *j*].

For example:

```
arr = FINDGEN(2,3,2)
PRINT, arr
```

IDL prints:

```
0.00000    1.00000
2.00000    3.00000
4.00000    5.00000

6.00000    7.00000
8.00000    9.00000
10.00000   11.00000
```

```
PRINT, MIN(arr, DIMENSION=2)
```

IDL prints:

```
0.00000    1.00000
6.00000    7.00000
```

```
PRINT, MIN(arr, DIMENSION=1)
```

IDL prints:

```
0.00000    2.00000    4.00000
6.00000    8.00000    10.00000
```

MAX

The name of a variable to receive the value of the maximum array element. If you need to find both the minimum and maximum array values, use this keyword to avoid scanning the array twice with separate calls to *MAX* and *MIN*.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Note

If the MIN function is run on an array containing NaN values and the NAN keyword is not set, an invalid result will occur.

SUBSCRIPT_MAX

Set this keyword to a named variable that will contain the one-dimensional subscript of the maximum element, the value of which is available via the MAX keyword.

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

```
; Create a simple two-dimensional array:
D = DIST(100)
; Find the minimum value in array D and print the result:
PRINT, MIN(D)
```

Version History

Introduced: Original

ABSOLUTE keyword added: 6.1

See Also

[ARRAY_INDICES](#), [MAX](#), [WHERE](#)

MIN_CURVE_SURF

The MIN_CURVE_SURF function interpolates a regularly- or irregularly-gridded set of points, over either a plane or a sphere, with either a minimum curvature surface or a thin-plate-spline surface.

Note

The accuracy of this function is limited by the single-precision floating-point accuracy of the machine.

Theory

A minimum curvature spline surface is fitted to the data points described by x , y , and z . The basis function is:

$$C(x_0, x_1, y_0, y_1) = d^2 \log(d^k)$$

where d is the distance between (x_0, y_0) , (x_1, y_1) and $k = 1$ for minimum curvature surface or $k = 2$ for Thin Plate Splines. For n data points, a system of $n+3$ simultaneous equations are solved for the coefficients of the surface. For any interpolation point, the interpolated value is:

$$f(x, y) = b_0 + b_1 \cdot x + b_2 \cdot y + \sum a_i \cdot C(x_i, x, y_i, y)$$

For a sphere the value is:

$$f(l, t) = b_0 + b_1 \cdot x + b_2 \cdot y + b_3 \cdot z + \sum a_i \cdot C(L_i, l, T_i, t)$$

On the sphere, l and t are longitude and latitude. $C(L_i, l, T_i, t)$ is the basis function above, with distance between the two points, (L_i, T_i) , and (l, t) , measured in radians of arc length. x , y , and z are the 3D cartesian coordinates of the point (l, t) on the unit sphere.

For a sphere with the CONST keyword set, the value is:

$$f(l, t) = b_0 + \sum a_i \cdot CL_i, l, T_i, t$$

The results obtained with the thin plate spline (TPS) and the minimum curvature surface (MCS) methods are very similar. The only difference is in the basis functions: TPS uses $d^2 \cdot \log(d^2)$, and MCS uses $d^2 \cdot \log(d)$, where d is the distance from point (x_i, y_i) .

This routine is written in the IDL language. Its source code can be found in the file `min_curve_surf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MIN_CURVE_SURF(Z [, X, Y] [, /DOUBLE] [, /TPS] [, /REGULAR]
  [, /SPHERE [, /CONST]] [, XGRID=[xstart, xspacing] | , XVALUES=array]
  [, YGRID=[ystart, yspacing] | , YVALUES=array] [, GS=[xspace, yspace]]
  [, BOUNDS=[xmin, ymin, xmax, ymax]] [, NX=value] [, NY=value]
  [, XOUT=vector] [, YOUT=vector] [, XPOUT, YPOUT=array])
```

Return Value

This function returns a two-dimensional floating-point array containing the interpolated surface, sampled at the grid points.

Arguments

Z, X, Y

Arrays containing the *Z*, *X*, and *Y* coordinates of the data points on the surface. Points need not be regularly gridded. For regularly gridded input data, *X* and *Y* are not used: the grid spacing is specified via the `XGRID` and `YGRID` (or `XVALUES` and `YVALUES`) keywords, and *Z* must be a two-dimensional array. For irregular grids, all three parameters must be present and have the same number of elements. If *Z* is specified as a double-precision value, the computation will be performed in double-precision arithmetic. If the `SPHERE` keyword is set, *X* and *Y* are given in degrees of longitude and latitude, respectively.

Keywords

CONST

Set this keyword to fit data on the sphere with a constant baseline, otherwise, data on the sphere is fit with a baseline that contains a constant term plus linear *X*, *Y*, and *Z* terms. This keyword has an effect only if `SPHERE` is set. See Theory above for the formulae.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic. If *Z* is double precision, the computations will also be done in double precision.

SPHERE

Set this keyword to perform interpolation on the surface of a sphere. The inputs X and Y should be given in degrees of longitude and latitude, respectively.

TPS

Set this keyword to use the thin-plate-spline method. The default is to use the minimum curvature surface method.

Input Grid Description Keywords:

REGULAR

If set, the Z parameter is a two-dimensional array of dimensions (n,m) , containing measurements over a regular grid. If any of XGRID, YGRID, XVALUES, or YVALUES are specified, REGULAR is implied. REGULAR is also implied if there is only one parameter, Z. If REGULAR is set, and no grid specifications are present, the grid is set to (0, 1, 2, ...).

XGRID

A two-element array, $[xstart, xspacing]$, defining the input grid in the x direction. Do not specify both XGRID and XVALUES.

XVALUES

An n -element array defining the x locations of $Z[i,j]$. Do not specify both XGRID and XVALUES.

YGRID

A two-element array, $[ystart, yspacing]$, defining the input grid in the y direction. Do not specify both YGRID and YVALUES.

YVALUES

An n -element array defining the y locations of $Z[i,j]$. Do not specify both YGRID and YVALUES.

Output Grid Description Keywords:

GS

The output grid spacing. If present, GS must be a two-element vector $[xs, ys]$, where xs is the horizontal spacing between grid points and ys is the vertical spacing. The default is based on the extents of x and y . If the grid starts at x value $xmin$ and ends at $xmax$, then the default horizontal spacing is $(xmax - xmin)/(NX-1)$. ys is computed in the same way. The default grid size, if neither NX or NY are specified, is 26 by 26.

BOUNDS

If present, BOUNDS must be a four-element array containing the grid limits in x and y of the output grid: $[xmin, ymin, xmax, ymax]$. If not specified, the grid limits are set to the extent of x and y .

NX

The output grid size in the x direction. NX need not be specified if the size can be inferred from GS and BOUNDS. The default value is 26.

NY

The output grid size in the y direction. NY need not be specified if the size can be inferred from GS and BOUNDS. The default value is 26.

XOUT

Use the XOUT keyword to specify a vector containing the output grid x values. If this parameter is supplied, GS, BOUNDS, and NX are ignored for the x output grid. XOUT allows you to specify irregularly-spaced output grids.

YOUT

Use the YOUT keyword to specify a vector containing the output grid y values. If this parameter is supplied, GS, BOUNDS, and NY are ignored for the y output grid. YOUT allows you to specify irregularly-spaced output grids.

XPOUT, YPOUT

Use the XPOUT and YPOUT keywords to specify arrays that contain the x and y values for the output points. If these keywords are used, the output grid need not be regular, and all other output grid parameters are ignored. XPOUT and YPOUT must have the same number of points, which is also the number of points returned in the result.

Examples

Example 1: Irregularly gridded cases

```

; Make a random set of points that lie on a Gaussian:
N = 15
X = RANDOMU(seed, N)
Y = RANDOMU(seed, N)

; The Gaussian:
Z = EXP(-2 * ((X-.5)^2 + (Y-.5)^2))

```

Use a 26 by 26 grid over the rectangle bounding x and y:

```

;Get the surface.
R = MIN_CURVE_SURF(Z, X, Y)

```

Alternatively, get a surface over the unit square, with spacing of 0.05:

```

R = MIN_CURVE_SURF(Z, X, Y, GS=[0.05, 0.05], BOUNDS=[0,0,1,1])

```

Alternatively, get a 10 by 10 surface over the rectangle bounding x and y:

```

R = MIN_CURVE_SURF(Z, X, Y, NX=10, NY=10)

```

Example 2: Regularly gridded cases

```

; Make some random data:
z = RANDOMU(seed, 5, 6)

; Interpolate to a 26 x 26 grid:
CONTOUR, MIN_CURVE_SURF(z, /REGULAR)

```

Version History

Introduced: Pre 4.0

See Also

[CONTOUR](#), [GRID_TPS](#), [TRI_SURF](#)

MK_HTML_HELP

The `MK_HTML_HELP` procedure, given a list of IDL procedure filenames (`.pro` files) or the names of directories containing such files, generates a file in HTML (HyperText Markup Language) format that contains documentation for those routines that contain standard IDL documentation headers. The resulting file can then be viewed with a web browser such as Microsoft Internet Explorer or Netscape Navigator.

`MK_HTML_HELP` procedure makes a single HTML file that starts with a list of the routines documented in the file. The names of routines in that list are hypertext links to the documentation for those routines. The documentation for each routine is simply the text of the documentation header copied from the corresponding `.pro` file—no reformatting is performed.

The documentation headers of the `.pro` files in question must have the following format:

- The first line of the documentation block contains only the characters `; +`, starting in column 1.
- The last line of the documentation block contains only the characters `; -`, starting in column 1.
- All other lines in the documentation block contain a `;` in column 1.
- If a line containing the string “NAME:” exists in the documentation block, the contents of the following line are used as the name of the routine being described. If the NAME: field is not present, the name of the source file is used as the routine name.

The file `template.pro` in the `examples` subdirectory of the IDL distribution contains a template for creating your own documentation headers.

This routine is supplied to allow users to make online documentation from their own IDL programs. Although it could be used to create an HTML documentation file from the `lib` subdirectory of the IDL distribution, we do not recommend doing so. The documentation headers on the files in the `lib` directory are used for historical purposes—most do not contain the most current or accurate documentation for those routines. The most current documentation for IDL’s built-in and library routines is found in IDL’s online help system (enter `?` at the IDL prompt).

This routine is written in the IDL language. Its source code can be found in the file `mk_html_help.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

MK_HTML_HELP, *Sources*, *Filename* [, /STRICT] [, TITLE=*string*] [, /VERBOSE]

Arguments

Sources

A string array containing the names of IDL procedure files (`.pro` files) or directories containing such files. The *Sources* array may contain both individual file and directory names. Each IDL procedure file must have the file extension `.pro`. Elements of the *Sources* array that do not have either of these extensions are assumed to be directories.

All `.pro` files found in *Sources* are searched for documentation headers. The documentation headers are extracted and saved in HTML format in the file specified by *Filename*.

Note

More than one documentation block may exist in a single input file.

Filename

A string containing the name of the output file to be generated. HTML files are usually saved in files named with a `.html` or `.htm` extension.

Keywords

STRICT

Set this keyword to force MK_HTML_HELP to adhere strictly to the HTML format by scanning the documentation blocks for HTML reserved characters and replacing them in the output file with the appropriate HTML syntax. HTML reserved characters include `<`, `>`, `&`, and `"`. By default, this keyword is set to zero to allow for faster processing of the input files.

TITLE

A string that supplies the name to be used as the title of the HTML document. The default is “Extended IDL Help”.

VERBOSE

Set this keyword to display informational messages as `MK_HTML_HELP` generates the HTML file. Normally, `MK_HTML_HELP` works silently.

Examples

To generate an HTML help file named `myhelp.html` from the `.pro` files in the directory `/usr/home/dave/myroutines`, use the command:

```
MK_HTML_HELP, '/usr/home/dave/myroutines', 'myhelp.html'
```

To generate an HTML help file for all routines in a given directory whose file names contain the word “plot”, use the following commands:

```
plotfiles=FILE_SEARCH('/usr/home/dave/myroutines/*plot*.pro')
MK_HTML_HELP, plotfiles, 'myplot.html'
```

Version History

Introduced: 4.0.1

See Also

[DOC_LIBRARY](#)

MODIFYCT

The `MODIFYCT` procedure updates the distribution color table file `colors1.tbl`, located in the `\resource\colors` subdirectory of the main IDL directory, or a user-designated file with a new, or modified, colortable.

This routine is written in the IDL language. Its source code can be found in the file `modifyct.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
MODIFYCT, Itab, Name, R, G, B [, FILE=filename]
```

Arguments

Itab

The index of the table to be updated, numbered from 0 to 255. If the specified entry is greater than the next available location in the table, the entry will be added to the table in the available location rather than the index specified by *Itab*. On return, *Itab* contains the index for the location that was modified or extended. The modified table can be then be loaded with the IDL command: `LOADCT, Itab`.

Name

A string, up to 32 characters long, that contains the name for the new color table.

R

A 256-element vector that contains the values for the red colortable.

G

A 256-element vector that contains the values for the green colortable.

B

A 256-element vector that contains the values for the blue colortable.

Keywords

FILE

Set this keyword to the name of a colortable file to be modified instead of the file `colors1.tbl`.

Version History

Introduced: Original

See Also

[LOADCT](#), [XLOADCT](#)

MOMENT

The **MOMENT** function computes the mean, variance, skewness, and kurtosis of a sample population contained in an n -element vector X . When $x = (x_0, x_1, x_2, \dots, x_{n-1})$, the various moments are defined as follows:

$$\text{Mean} = \bar{x} = \frac{1}{N} \sum_{j=0}^{N-1} x_j$$

$$\text{Variance} = \frac{1}{N-1} \sum_{j=0}^{N-1} (x_j - \bar{x})^2$$

$$\text{Skewness} = \frac{1}{N} \sum_{j=0}^{N-1} \left(\frac{x_j - \bar{x}}{\sqrt{\text{Variance}}} \right)^3$$

$$\text{Kurtosis} = \frac{1}{N} \sum_{j=0}^{N-1} \left(\frac{x_j - \bar{x}}{\sqrt{\text{Variance}}} \right)^4 - 3$$

$$\text{Mean Absolute Deviation} = \frac{1}{N} \sum_{j=0}^{N-1} |x_j - \bar{x}|$$

$$\text{Standard Deviation} = \sqrt{\text{Variance}}$$

This routine is written in the IDL language. Its source code can be found in the file `moment.pro` in the `lib` subdirectory of the IDL distribution.

Return Value

If the vector contains n identical elements, **MOMENT** computes the mean and variance, and returns the IEEE value NaN for the skewness and kurtosis, which are

not defined. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual.)

Syntax

```
Result = MOMENT( X [, /DOUBLE] [, MDEV=variable] [, /NAN]
                [, SDEV=variable] )
```

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

MDEV

Set this keyword to a named variable that will contain the mean absolute deviation of X .

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for information on IEEE floating-point values.)

SDEV

Set this keyword to a named variable that will contain the standard deviation of X .

Examples

```
; Define an n-element sample population:
X = [65, 63, 67, 64, 68, 62, 70, 66, 68, 67, 69, 71, 66, 65, 70]
; Compute the mean, variance, skewness and kurtosis:
result = MOMENT(X)
PRINT, 'Mean: ', result[0] & PRINT, 'Variance: ', result[1] & $
      PRINT, 'Skewness: ', result[2] & PRINT, 'Kurtosis: ', result[3]
```

IDL prints:

```
Mean:      66.7333
Variance:  7.06667
Skewness:  -0.0942851
Kurtosis:  -1.18258
```

Version History

Introduced: 4.0

See Also

[KURTOSIS](#), [HISTOGRAM](#), [MAX](#), [MEAN](#), [MEANABSDEV](#), [MEDIAN](#), [MIN](#),
[MOMENT](#), [STDDEV](#), [SKEWNESS](#), [VARIANCE](#)

MORPH_CLOSE

The MORPH_CLOSE function applies the closing operator to a binary or grayscale image. MORPH_CLOSE is simply a dilation operation followed by an erosion operation. The closing operation is an idempotent operator—applying it more than once produces no further effect.

Both the opening and the closing operators have the effect of smoothing the image, with the opening operation removing pixels, and the closing operation adding pixels.

This routine is written in the IDL language. Its source code can be found in the file `morph_close.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MORPH_CLOSE (Image, Structure [, /GRAY]  
[, PRESERVE_TYPE=bytearray | /UINT | /ULONG] [, VALUES=array] )
```

Return Value

The result of a closing operation is that small holes and gaps within the image (smaller than the size of *Structure*) are filled, yet the original sizes of the primary foreground features are maintained.

Arguments

Image

A one-, two-, or three-dimensional array upon which the closing operation is to be performed. If neither of the keywords GRAY or VALUES is present, the image is treated as a binary image with all nonzero pixels considered as 1.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values - either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

GRAY

Set this keyword to perform a grayscale, rather than binary, operation. Nonzero elements of the *Structure* parameter determine the shape of the structuring element. If the **VALUES** keyword is not present, all elements of the structuring element are 0.

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type **BYTE**, **UINT**, or **ULONG**. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of **UINT** and **ULONG**.

UINT

Set this keyword to return an unsigned integer array. This keyword only applies for grayscale operations, and is mutually exclusive of the **ULONG** and **PRESERVE_TYPE** keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword only applies for grayscale operations, and is mutually exclusive of the **UINT** and **PRESERVE_TYPE** keywords.

VALUES

An array of the same dimensions as *Structure* providing the values of the structuring element. The presence of this keyword implies a grayscale operation.

Examples

The following code reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then applies a threshold and a morphological closing operator with a 3 by 3 square kernel to the original image. Notice that most of the holes in the pollen grains have been filled by the closing operator.

```
;Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Read the image
path=FILEPATH('pollens.jpg',SUBDIR=['examples','demo','demodata'])
READ_JPEG, path, img
```

```
;Create window:
WINDOW, 0, XSIZE=700, YSIZE=540

;Show original image
XYOUTS, 180, 525, 'Original Image', ALIGNMENT=.5, /DEVICE
TVSCL, img, 20, 280

;Apply the threshold creating a binary image
thresh = img GE 140B

;Load a simple color table
TEK_COLOR

;Display Edges
XYOUTS, 520, 525, 'Edges', ALIGNMENT=.5, /DEVICE
TV, thresh, 360, 280

;Apply closing operator
closing = MORPH_CLOSE(thresh, REPLICATE(1,3,3))

;Show the result
XYOUTS, 180, 265, 'Closing Operator', ALIGNMENT=.5, /DEVICE
TV, closing, 20, 20

;Show added pixels in white
XYOUTS, 520, 265, 'Added Pixels in White', ALIGNMENT=.5, /DEVICE
TV, closing + thresh, 360, 20
```

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_DISTANCE](#), [MORPH_GRADIENT](#),
[MORPH_HITORMISS](#), [MORPH_OPEN](#), [MORPH_THIN](#), [MORPH_TOPHAT](#)

MORPH_DISTANCE

The MORPH_DISTANCE function estimates N -dimensional distance maps, which contain for each foreground pixel the distance to the nearest background pixel, using a given norm. Available norms include: Euclidean, which is exact and is also known as the Euclidean Distance Map (EDM), and two more efficient approximations, chessboard and city block.

The distance map is useful for a variety of morphological operations: thinning, erosion and dilation by discs of radius “ r ”, and granulometry.

Syntax

```
Result = MORPH_DISTANCE (Data [, /BACKGROUND]  
[, NEIGHBOR_SAMPLING={1 | 2 | 3 }] [, /NO_COPY])
```

Return Value

The returned variable is an array of the same dimension as the input array.

Arguments

Data

An input binary array. Zero-valued pixels are considered to be part of the background.

Keywords

BACKGROUND

By default, the EDM is computed for the foreground (non-zero) features in the *Data* argument. Set this keyword to compute the EDM of the background features instead of the foreground features. If the keyword is set, elements of *Result* that are on an edge are set to 0.

NEIGHBOR_SAMPLING

Set this keyword to indicate how the distance of each neighbor from a given pixel is determined. Valid values include:

- 0 = default. No diagonal neighbors. Each neighbor is assigned a distance of 1.
- 1 = chessboard. Each neighbor is assigned a distance of 1.
- 2 = city block. Each neighbor is assigned a distance corresponding to the number of pixels to be visited when travelling from the current pixel to the neighbor. (The path can only take 90 degree turns; no diagonal paths are allowed.)
- 3 = actual distance. Each neighbor is assigned its actual distance from the current pixel (within the limitations of floating point representations).

Default Two Dimensional Example

```

      1
  1   X   1
      1

```

Chessboard Two-Dimensional Example

```

  1   1   1
  1   X   1
  1   1   1

```

City Block Two-Dimensional Example:

```

  2   1   2
  1   X   1
  2   1   2

```

Actual Distance Two-Dimensional Example

```

sqrt(2)  1  sqrt(2)
      1   X   1
sqrt(2)  1  sqrt(2)

```

NO_COPY

Set this keyword to request that the input array be reused, if possible. If this keyword is set, the input argument is undefined upon return.

Examples

The following code reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then applies a threshold and the morphological distance operator. Thresholding the result distance operator with a value of “n” produces the equivalent of eroding the thresholded image with a disc of radius “n”.

```

;Handle TrueColor displays:

```

```

DEVICE, DECOMPOSED=0

;Read the image
path=FILEPATH('pollens.jpg',SUBDIR=['examples','demo','demodata'])
READ_JPEG, path, img

; Create window:
WINDOW, 0, XSIZE=700, YSIZE=540

; Display the original image
XYOUTS, 180, 525, 'Original Image', ALIGNMENT=.5, /DEVICE
TV, img, 20, 280

; Apply the threshold:
thresh = img GE 140B

; Display the thresholded image
XYOUTS, 520, 525, 'Thresholded Image', ALIGNMENT=.5, /DEVICE
TVSCL, thresh, 360, 280

;Create Euclidean distance function
edist = MORPH_DISTANCE(thresh, NEIGHBOR_SAMPLING = 3)

; Display the distance function
XYOUTS, 180, 265, 'Distance Function', ALIGNMENT=.5, /DEVICE
TVSCL, edist, 20, 20

; Display image after erosion with a disc of radius 5:
XYOUTS, 520, 265, 'After erosion with disc of radius 5',
ALIGNMENT=.5, /DEVICE
TVSCL, edist GT 5, 360, 20

```

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_CLOSE](#), [MORPH_GRADIENT](#),
[MORPH_HITORMISS](#), [MORPH_OPEN](#), [MORPH_THIN](#), [MORPH_TOPHAT](#)

MORPH_GRADIENT

The MORPH_GRADIENT function applies the morphological gradient operator to a grayscale image. MORPH_GRADIENT is the subtraction of an eroded version of the original image from a dilated version of the original image.

This routine is written in the IDL language. Its source code can be found in the file `morph_gradient.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MORPH_GRADIENT (Image, Structure [, PRESERVE_TYPE=bytearray |  
/UINT | /ULONG] [, VALUES=array] )
```

Return Value

The practical result of a morphological gradient operation is that the boundaries of features are highlighted.

Arguments

Image

A one-, two-, or three-dimensional array upon which the morphological gradient operation is to be performed.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values - either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type BYTE, UINT, or ULONG. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of the `UINT` and `ULONG` keywords.

UINT

Set this keyword to return an unsigned integer array. This keyword is mutually exclusive of the `ULONG` and `PRESERVE_TYPE` keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword is mutually exclusive of the `UINT` and `PRESERVE_TYPE` keywords.

VALUES

An array of the same dimensions as the *Structure* argument providing the values of the structuring element. If the `VALUES` keyword is not present, all elements of the structuring element are 0.

Examples

The following code reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then creates disc of radius 2, in a 5 by 5 array, with all elements within a radius of 2 from the center set to 1. This disc is used as the structuring element for the morphological gradient which is then displayed as both a gray scale image, and as a thresholded image.

```
;Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Read the image
path=FILEPATH('pollens.jpg',SUBDIR=['examples','demo','demodata'])
READ_JPEG, path, img

; Create window:
WINDOW, 0, XSIZE=700, YSIZE=540

;Show original image
XYOUTS, 180, 525, 'Original Image', ALIGNMENT=.5, /DEVICE
TVSCL, img, 20, 280

;Define disc radius
r = 2

;Create a binary disc of given radius.
disc = SHIFT(DIST(2*r+1), r, r) LE r

bdisc = MORPH_GRADIENT(img, disc)

;Show edges
```

```
XYOUTS, 520, 525, 'Edges', ALIGNMENT=.5, /DEVICE
TVSCL, bdisc, 360, 280

;Show thresholded edges
XYOUTS, 180, 265, 'Threshold Edges', ALIGNMENT=.5, /DEVICE
TVSCL, bdisc ge 100, 20, 20
```

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#),
[MORPH_HITORMISS](#), [MORPH_OPEN](#), [MORPH_THIN](#), [MORPH_TOPHAT](#)

MORPH_HITORMISS

The MORPH_HITORMISS function applies the hit-or-miss operator to a binary image. The hit-or-miss operator is implemented by first applying an erosion operator with a *hit* structuring element to the original image. Then an erosion operator is applied to the complement of the original image with a secondary *miss* structuring element. The result is the intersection of the two results.

This routine is written in the IDL language. Its source code can be found in the file `morph_hitormiss.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = MORPH_HITORMISS (*Image*, *HitStructure*, *MissStructure*)

Return Value

The resulting image corresponds to the positions where the hit structuring element lies within the image, and the miss structure lies completely outside the image. The two structures must not overlap.

Arguments

Image

A one-, two-, or three-dimensional array upon which the morphological operation is to be performed. The image is treated as a binary image with all nonzero pixels considered as 1.

HitStructure

A one-, two-, or three-dimensional array to be used as the hit structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

MissStructure

A one-, two-, or three-dimensional array to be used as the miss structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

Note

It is assumed that the HitStructure and the MissStructure arguments are disjoint.

Keywords

None.

Example

The following code snippet identifies blobs with a radius of at least 2, but less than 4 in the pollen image. These regions totally enclose a disc of radius 2, contained in the 5 x 5 kernel named “hit”, and in turn, fit within a hole of radius 4, contained in the 9 x 9 array named “miss”. Executing this specific example identifies four blobs in the image with these attributes.

```

;Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Read the image
path=FILEPATH('pollens.jpg',SUBDIR=['examples','demo','demodata'])
READ_JPEG, path, img

WINDOW, 0, XSIZE=700, YSIZE=540

; Display the original image
XYOUTS, 180, 525, 'Original Image', ALIGNMENT=.5, /DEVICE
TV, img, 20, 280

rh = 2 ;Radius of hit disc
rm = 4 ;Radius of miss disc

;Create a binary disc of given radius.
hit = SHIFT(DIST(2*rh+1), rh, rh) LE rh

;Complement of disc for miss
miss = SHIFT(DIST(2*rm+1), rm, rm) GT rm

;Load discrete color table
TEK_COLOR

;Apply the threshold
thresh = img GE 140B

; Display the thresholded image
XYOUTS, 520, 525, 'Thresholded Image', ALIGNMENT=.5, /DEVICE
TV, thresh, 360, 280

```

```
;Compute matches
matches = MORPH_HITORMISS(thresh, hit, miss)

;Expand matches to size of hit disc
matches = DILATE(matches, hit)

;Show matches.
XYOUTS, 180, 265, 'Matches', ALIGNMENT=.5, /DEVICE
TV, matches, 20, 20

;Superimpose, showing hit regions in blue.
;(Blue = color index 4 for tek_color.)
XYOUTS, 520, 265, 'Superimposed, hit regions in blue', $
    ALIGNMENT=.5, /DEVICE
TV, thresh + 3*matches, 360, 20
```

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#),
[MORPH_GRADIENT](#), [MORPH_OPEN](#), [MORPH_THIN](#), [MORPH_TOPHAT](#)

MORPH_OPEN

The MORPH_OPEN function applies the opening operator to a binary or grayscale image. MORPH_OPEN is simply an erosion operation followed by a dilation operation. The opening operation is an idempotent operator, applying it more than once produces no further effect.

An alternative definition of the opening, is that it is the union of all sets containing the structuring element in the original image. Both the opening and the closing operators have the effect of smoothing the image, with the opening operation removing pixels, and the closing operation adding pixels.

This routine is written in the IDL language. Its source code can be found in the file `morph_open.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MORPH_OPEN (Image, Structure [, /GRAY]  
[, PRESERVE_TYPE=bytearray | /UINT | /ULONG] [, VALUES=array] )
```

Return Value

The result of an opening operation is that small features (e.g., noise) within the image are removed, yet the original sizes of the primary foreground features are maintained.

Arguments

Image

A one-, two-, or three-dimensional array upon which the opening operation is to be performed. If neither of the keywords GRAY or VALUES is present, the image is treated as a binary image with all nonzero pixels considered as 1.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values — either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

GRAY

Set this keyword to perform a grayscale, rather than binary, operation. Nonzero elements of the *Structure* parameter determine the shape of the structuring element. If the **VALUES** keyword is not present, all elements of the structuring element are 0.

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type **BYTE**, **UINT**, or **ULONG**. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of the **UINT** and **ULONG** keywords.

UINT

Set this keyword to return an unsigned integer array. This keyword only applies for grayscale operations, and is mutually exclusive of the **ULONG** and **PRESERVE_TYPE** keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword only applies for grayscale operations and is mutually exclusive of the **UINT** and **PRESERVE_TYPE** keywords.

VALUES

An array of the same dimensions as *Structure* providing the values of the structuring element. The presence of this keyword implies a grayscale operation.

Examples

The following code reads a data file in the IDL Demo data directory containing a magnified image of grains of pollen. It then applies a threshold and a morphological opening operator with a 3 by 3 square kernel to the original image. Notice that much of the irregular borders of the grains have been smoothed by the opening operator.

```
; Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Read the image
path=FILEPATH('pollens.jpg',SUBDIR=['examples','demo','demodata'])
READ_JPEG, path, img
```

```
; Create window:
WINDOW, 0, XSIZE=700, YSIZE=540

;Show original image
XYOUTS, 180, 525, 'Original Image', ALIGNMENT=.5, /DEVICE
TV, img, 20, 280

;Apply the threshold
thresh = img GE 140B

;Load a simple color table
TEK_COLOR

;Display edges
XYOUTS, 520, 525, 'Edges', ALIGNMENT=.5, /DEVICE
TV, thresh, 360, 280

;Apply opening operator
open = MORPH_OPEN(thresh, REPLICATE(1,3,3))

;Show the result
XYOUTS, 180, 265, 'Opening Operator', ALIGNMENT=.5, /DEVICE
TV, open, 20, 20

;Show pixels that have been removed in white
XYOUTS, 520, 265, 'Removed Pixels in White', ALIGNMENT=.5, /DEVICE
TV, open + thresh, 360, 20
```

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#),
[MORPH_GRADIENT](#), [MORPH_HITORMISS](#), [MORPH_THIN](#),
[MORPH_TOPHAT](#)

MORPH_THIN

The MORPH_THIN function performs a thinning operation on binary images. The thinning operator is implemented by first applying a hit or miss operator to the original image with a pair of structuring elements, and then subtracting the result from the original image.

This routine is written in the IDL language. Its source code can be found in the file `morph_thin.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = MORPH_THIN (*Image*, *HitStructure*, *MissStructure*)

Return Value

In typical applications, this operator is repeatedly applied with the two structuring elements, rotating them after each application, until the result remains unchanged.

Arguments

Image

A one-, two-, or three-dimensional array upon which the thinning operation is to be performed. The image is treated as a binary image with all nonzero pixels considered as 1.

HitStructure

A one-, two-, or three-dimensional array to be used as the hit structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

MissStructure

A one-, two-, or three-dimensional array to be used as the miss structuring element. The elements are interpreted as binary values — either zero or nonzero. This structuring element must have the same number of dimensions as the *Image* argument.

Note

It is assumed that the *HitStructure* and the *MissStructure* arguments are disjoint.

Keywords

None.

Examples

See “[Thinning Image Objects](#)” in Chapter 11 [Image Processing in IDL](#) manual for examples.

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#),
[MORPH_GRADIENT](#), [MORPH_HITORMISS](#), [MORPH_OPEN](#),
[MORPH_TOPHAT](#)

MORPH_TOPHAT

The MORPH_TOPHAT function applies the top-hat operator to a grayscale image. The top-hat operator is implemented by first applying the opening operator to the original image, then subtracting the result from the original image. Applying the top-hat operator provides a result that shows the bright peaks within the image.

This routine is written in the IDL language. Its source code can be found in the file `morph_tophat.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MORPH_TOPHAT ( Image, Structure [, PRESERVE_TYPE=bytearray |  
/UINT | /ULONG] [, VALUES=array] )
```

Return Value

Returns the resulting one-, two-, or three-dimensional array.

Arguments

Image

A one-, two-, or three-dimensional array upon which the top-hat operation is to be performed.

Structure

A one-, two-, or three-dimensional array to be used as the structuring element. The elements are interpreted as binary values — either zero or nonzero. The structuring element must have the same number of dimensions as the *Image* argument.

Keywords

PRESERVE_TYPE

Set this keyword to return the same type as the input array. The input array must be of type BYTE, UINT, or ULONG. This keyword only applies for grayscale erosion/dilation, and is mutually exclusive of the UINT and ULONG keywords.

UINT

Set this keyword to return an unsigned integer array. This keyword is mutually exclusive of the `ULONG` and `PRESERVE_TYPE` keywords.

ULONG

Set this keyword to return an unsigned longword integer array. This keyword is mutually exclusive of the `UINT` and `PRESERVE_TYPE` keywords.

VALUES

An array of the same dimensions as the *Structure* argument providing the values of the structuring element. If the `VALUES` keyword is not present, all elements of the structuring element are 0.

Examples

The following example illustrates an application of the top-hat operator to an image in the `examples/demo/demodata` directory:

```

; Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Read the image
path=FILEPATH('pollens.jpg',SUBDIR=['examples','demo','demodata'])
READ_JPEG, path, img

; Create window:
WINDOW, 0, XSIZE=700, YSIZE=280

;Show original
XYOUTS, 180, 265, 'Original Image', ALIGNMENT=.5, /DEVICE
TVSCL, img, 20, 20

;Radius of disc
r = 2

;Create a binary disc of given radius.
disc = SHIFT(DIST(2*r+1), r, r) LE r

;Apply top-hat operator
tophat = MORPH_TOPHAT(img, disc)

;Display stretched result.
XYOUTS, 520, 265, 'Stretched Result', ALIGNMENT=.5, /DEVICE
TVSCL, tophat < 50, 360, 20

```

Version History

Introduced: 5.3

See Also

[DILATE](#), [ERODE](#), [MORPH_CLOSE](#), [MORPH_DISTANCE](#),
[MORPH_GRADIENT](#), [MORPH_HITORMISS](#), [MORPH_OPEN](#), [MORPH_THIN](#)

MPEG_CLOSE

The `MPEG_CLOSE` procedure closes an MPEG sequence opened with the `MPEG_OPEN` routine. Note that `MPEG_CLOSE` does not save the MPEG file associated with the MPEG sequence; use `MPEG_SAVE` to save the file. The specified MPEG sequence identifier will no longer be valid after calling `MPEG_CLOSE`.

This routine is written in the IDL language. Its source code can be found in the file `mpeg_close.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

`MPEG_CLOSE`, *mpegID*

Arguments

mpegID

The unique identifier of the MPEG sequence to be freed. (MPEG sequence identifiers are returned by the `MPEG_OPEN` routine.)

Keywords

None.

Examples

See [MPEG_OPEN](#) for an example using this routine.

Version History

Introduced: 5.1

See Also

[MPEG_OPEN](#), [MPEG_PUT](#), [MPEG_SAVE](#), [XINTERANIMATE](#)

MPEG_OPEN

The MPEG_OPEN function initializes an IDLgrMPEG object for MPEG encoding and returns the object reference. The MPEG routines provide a wrapper around the IDL Object Graphics IDLgrMPEG object, eliminating the need to use the Object Graphics interface to create MPEG files.

Note

The MPEG standard does not allow movies with odd numbers of pixels to be created.

This routine is written in the IDL language. Its source code can be found in the file `mpeg_open.pro` in the `lib` subdirectory of the IDL distribution.

Note

MPEG support in IDL requires a special license. For more information, contact your RSI sales representative or technical support.

Syntax

```
mpegID = MPEG_OPEN( Dimensions [, BITRATE=value] [, FILENAME=string]
  [, IFRAME_GAP=integer value] [, MOTION_VEC_LENGTH={1 | 2 | 3}]
  [ QUALITY=value{0 to 100} ] )
```

Return Value

Returns the reference to the IDLgrMPEG object.

Arguments

Dimensions

A two-element vector of the form `[xsize, ysize]` indicating the dimensions of the images to be used as frames in the MPEG movie file. All images in the MPEG file must have the same dimensions.

Note

When creating MPEG files, you must be aware of the capabilities of the MPEG decoder you will be using to view it. Some decoders only support a limited set of sampling and bitrate parameters to normalize computational complexity, buffer size, and memory bandwidth. For example, the Windows Media Player supports a

limited set of sampling and bitrate parameters. In this case, it is best to use 352 x 240 x 30 fps or 352 x 288 x 25 fps when determining the dimensions and frame rate for your MPEG file. When opening a file in Windows Media Player that does not use these dimensions, you will receive a “Bad Movie File” error message. The file is not “bad”, this decoder just doesn’t support the dimensions of the MPEG.

Keywords

BITRATE

Set this keyword to a double-precision value to specify the MPEG movie bit rate. Higher bit rates will create higher quality MPEGs but will increase file size. The following table describes the valid values:

MPEG Version	Range
MPEG 1	0.1 to 104857200.0
MPEG 2	0.1 to 429496729200.0

Table 3-90: BITRATE Value Range

If you do not set this keyword, IDL computes the BITRATE value based upon the value you have specified for the QUALITY keyword.

Note

Only use the BITRATE keyword if changing the QUALITY keyword value does not produce the desired results. It is highly recommended to set the BITRATE to at least several times the frame rate to avoid unusable MPEG files or file generation errors.

FILENAME

Set this keyword equal to a string representing the name of the file in which the encoded MPEG sequence is to be saved. The default file name is `idl.mpg`.

IFRAME_GAP

Set this keyword to a positive integer value that specifies the number of frames between I frames to be created in the MPEG file. I frames are full-quality image frames that may have a number of predicted or interpolated frames between them.

If you do not specify this keyword, IDL computes the `IFRAME_GAP` value based upon the value you have specified for the `QUALITY` keyword.

Note

Only use the `IFRAME_GAP` keyword if changing the `QUALITY` keyword value does not produce the desired results.

MOTION_VEC_LENGTH

Set this keyword to an integer value specifying the length of the motion vectors to be used to generate predictive frames. Valid values include:

- 1 = Small motion vectors.
- 2 = Medium motion vectors.
- 3 = Large motion vectors.

If you do not set this keyword, IDL computes the `MOTION_VEC_LENGTH` value based upon the value you have specified for the `QUALITY` keyword.

Note

Only use the `MOTION_VEC_LENGTH` keyword if changing the `QUALITY` value does not produce the desired results.

QUALITY

Set this keyword to an integer value between 0 (low quality) and 100 (high quality) inclusive to specify the quality at which the MPEG stream is to be stored. Higher quality values result in lower rates of time compression and less motion prediction which provide higher quality MPEGs but with substantially larger file size. Lower quality factors may result in longer MPEG generation times. The default is 50.

Note

Since MPEG uses JPEG (lossy) compression, the original picture quality can't be reproduced even when setting `QUALITY` to its highest setting.

Examples

The following sequence of IDL commands illustrates the steps needed to create an MPEG movie file from a series of image arrays named `image0`, `image1`, ..., `imageN`, where n is the zero-based index of the last image in the movie:

```
; Open an MPEG sequence:
mpegID = MPEG_OPEN()

; Add the first frame:
MPEG_PUT, mpegID, IMAGE=image0, FRAME=0
MPEG_PUT, mpegID, IMAGE=image1, FRAME=1

; Subsequent frames:
...

; Last frame:
MPEG_PUT, mpegID, IMAGE=imagen, FRAME=n

; Save the MPEG sequence in the file myMovie.mpg:
MPEG_SAVE, mpegID, FILENAME='myMovie.mpg'

; Close the MPEG sequence:
MPEG_CLOSE, mpegID
```

Version History

Introduced: 5.1

See Also

[MPEG_CLOSE](#), [MPEG_PUT](#), [MPEG_SAVE](#), [XINTERANIMATE](#)

MPEG_PUT

The MPEG_PUT procedure stores the specified image array at the specified frame index in an MPEG sequence.

This routine is written in the IDL language. Its source code can be found in the file `mpeg_put.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
MPEG_PUT, mpegID [, /COLOR] [, FRAME=frame_number] [, IMAGE=array | ,  
WINDOW=index] [, /ORDER]
```

Arguments

mpegID

The unique identifier of the MPEG sequence into which the image will be inserted. (MPEG sequence identifiers are returned by the MPEG_OPEN routine.)

Keywords

COLOR

Set this keyword to read off an 8-bit display and pass the information through the current color table to create a 24-bit image.

FRAME

Set this keyword equal to an integer specifying the frame at which the image is to be loaded. If the frame number matches a previously loaded frame, the previous frame is overwritten. The default is 0.

IMAGE

Set this keyword equal to an $m \times n$ image array or a $3 \times m \times n$ True Color image array representing the image to be loaded at the specified frame. This keyword is ignored if the WINDOW keyword is specified.

ORDER

Set this keyword to indicate that the rows of the image should be drawn from top to bottom. By default, the rows are drawn from bottom to top.

WINDOW

Set this keyword to the index of a Direct Graphics Window (or to an object reference to an `IDLgrWindow` or `IDLgrBuffer` object) to indicate that the image to be loaded is to be read from the given window or buffer. If this keyword is specified, it overrides the value of the `IMAGE` keyword.

Examples

See [MPEG_OPEN](#) for an example using this routine.

Version History

Introduced: 5.1

See Also

[MPEG_CLOSE](#), [MPEG_OPEN](#), [MPEG_SAVE](#), [XINTERANIMATE](#)

MPEG_SAVE

The MPEG_SAVE procedure encodes and saves an open MPEG sequence.

This routine is written in the IDL language. Its source code can be found in the file `mpeg_save.pro` in the `lib` subdirectory of the IDL distribution.

Note

MPEG support in IDL requires a special license. For more information, contact your RSI sales representative or technical support.

Syntax

```
MPEG_SAVE, mpegID [, FILENAME=string]
```

Arguments

mpegID

The unique identifier of the MPEG sequence to be saved to a file. (MPEG sequence identifiers are returned by the MPEG_OPEN routine.)

Keywords

FILENAME

Set this keyword to a string representing the name of the file to which the encoded MPEG sequence is to be saved. The default is `idl.mpg`.

Examples

See [MPEG_OPEN](#) for an example using this routine.

Version History

Introduced: 5.1

MULTI

The MULTI procedure expands the current color table to “wrap around” some number of times.

This routine is written in the IDL language. Its source code can be found in the file `multi.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

MULTI, *N*

Arguments

N

The number of times the color table will wrap. This parameter need not be an integer.

Keywords

None.

Examples

Display an image, load color table 1, and make that color table “wrap around” 3 times. Enter:

```
;Display a simple image.
TVSCL, DIST(256)

;Load color table 1.
LOADCT, 1

;See how the new color table affects the image.
MULTI, 3
```

Version History

Introduced: Original

See Also

[STRETCH](#), [XLOADCT](#)

N_ELEMENTS

The `N_ELEMENTS` function returns the number of elements contained in an expression or variable.

Syntax

Result = `N_ELEMENTS(Expression)`

Return Value

Returns the number of elements.

Arguments

Expression

The expression for which the number of elements is to be returned. Scalar expressions always have one element. The number of elements in an array is equal to the product of its dimensions. If *Expression* is an undefined variable, `N_ELEMENTS` returns zero.

Keywords

None.

Examples

Example 1

This example finds the number of elements in an array:

```
; Create an integer array:  
I = INTARR(4, 5, 3, 6)  
; Find the number of elements in I and print the result:  
PRINT, N_ELEMENTS(I)
```

Example 2

A typical use of `N_ELEMENTS` is to check if an optional input is defined, and if not, set it to a default value:

```
IF (N_ELEMENTS(roo) EQ 0) THEN roo=rooDefault
```

The original value of `roo` may be altered by a called routine, passing a different value back to the caller. Unless you intend for the routine to behave in this manner, you should prevent it by differentiating `N_ELEMENTS`' parameter from your routine's variable:

```
IF (N_ELEMENTS(roo) EQ 0) THEN rooUse=rooDefault $
ELSE rooUse=roo
```

Version History

Introduced: Original

See Also

[N_TAGS](#)

N_PARAMS

The N_PARAMS function returns the number of parameters used in calling an IDL procedure or function. This function is only useful within IDL procedures or functions. User-written procedures and functions can use N_PARAMS to determine if they were called with optional parameters.

Note

In the case of object method procedures and functions, the SELF argument is not counted by N_PARAMS.

Syntax

Result = N_PARAMS()

Return Value

Returns the number of non-keyword parameters.

Arguments

None. This function always returns the number of parameters that were used in calling the procedure or function from which N_PARAMS is called.

Keywords

None.

Version History

Introduced: Original

See Also

[KEYWORD_SET](#)

N_TAGS

The N_TAGS function returns the number of structure tags contained in a structure expression.

Syntax

```
Result = N_TAGS( Expression [, /DATA_LENGTH] [, /LENGTH] )
```

Return Value

Returns the number of structure tags and optionally returns the size, in bytes, of the structure.

Arguments

Expression

The expression for which the number of structure tags is to be returned. Expressions that are not of structure type are considered to have no tags. N_TAGS does not search for tags recursively, so if *Expression* is a structure containing nested structures, only the number of tags in the outermost structure are counted.

Keywords

DATA_LENGTH

Set this keyword to return the total length of all the data fields contained within the structure, in bytes. This differs from LENGTH in that it does not include any alignment padding required by the structure.

The length of the data for a given structure will be the same on any system, *as long as the data does not contain strings*.

If a data field contains a string, the DATA_LENGTH keyword returns the length of the string's internal *descriptor*, rather than the length of the string data. (In IDL, strings are represented internally using a *descriptor structure* that contains information about the string, including its length and a pointer to the string data.) This means that on a given machine, the DATA_LENGTH keyword will report the same length value for all fields containing string data, regardless of the contents of the strings.

Note

The length of a string descriptor can vary between machines, and also between 32-bit and 64-bit versions of IDL. Length differences are caused by differences in structure alignment rules or the size of a machine pointer on the system, and are fundamental properties of the underlying system.

LENGTH

Set this keyword to return the length of the structure, in bytes.

Note

The length of a structure is machine dependent. The length of a given structure will vary depending upon the host machine. IDL pads and aligns structures in a manner consistent with the host machine's C compiler.

Examples

Find the number of tags in the system variable !P and print the result by entering:

```
PRINT, N_TAGS(!P)
```

Find the length of !P, in bytes:

```
PRINT, N_TAGS(!P, /LENGTH)
```

Version History

Introduced: Original

See Also

[CREATE_STRUCT](#), [N_ELEMENTS](#), [TAG_NAMES](#), Chapter 9, “Structures” in the *Building IDL Applications* manual

NCDF_* Routines

For information, see [Chapter 6, “Network Common Data Format”](#) in the *IDL Scientific Data Formats* manual.

NEWTON

The NEWTON function solves a system of n non-linear equations in n dimensions using a globally-convergent Newton's method.

NEWTON is based on the routine `newt` described in section 9.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = NEWTON( X, Vecfunc [, CHECK=variable] [, /DOUBLE]
                [, ITMAX=value] [, STEPMAX=value] [, TOLF=value] [, TOLMIN=value]
                [, TOLX=value] )
```

Return Value

The result is an n -element vector containing the solution.

Arguments

X

An n -element vector containing an initial guess at the solution of the system.

Vecfunc

A scalar string specifying the name of a user-supplied IDL function that defines the system of non-linear equations. This function must accept an n -element vector argument X and return an n -element vector result.

For example, suppose the non-linear system is defined by the following equations:

$$y_0 = x_0 + x_1 - 3, \quad y_1 = x_0^2 + x_1^2 - 9$$

We write a function `NEWTFUNC` to express these relationships in the IDL language:

```
FUNCTION newtfunc, X
    RETURN, [X[0] + X[1] - 3.0, X[0]^2 + X[1]^2 - 9.0]
END
```

Keywords

CHECK

NEWTON calls an internal function named `fmin()` to determine whether the routine has converged to a local minimum rather than to a global minimum (see *Numerical Recipes*, section 9.7). Use the CHECK keyword to specify a named variable which will be set to 1 if the routine has converged to a local minimum or to 0 if it has not. If the routine does converge to a local minimum, try restarting from a different initial guess to obtain the global minimum.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ITMAX

The maximum allowed number of iterations. The default value is 200.

STEPMAX

The scaled maximum step length allowed in line search. The default value is 100.0.

TOLF

Set the convergence criterion on the function values. The default value is 1.0×10^{-4} .

TOLMIN

Set the criterion for deciding whether spurious convergence to a minimum of the function `fmin()` has occurred. The default value is 1.0×10^{-6} .

TOLX

Set the convergence criterion on X . The default value is 1.0×10^{-7} .

Examples

Use NEWTON to solve an n -dimensional system of n non-linear equations. Systems of non-linear equations may have multiple solutions; starting the algorithms with different initial guesses enables detection of different solutions.

```
FUNCTION newtfunc, X
RETURN, [X[0] + X[1] - 3, X[0]^2 + X[1]^2 - 9]

END
```

```
PRO TEST_NEWTON

; Provide an initial guess as the algorithm's starting point:
X = [1d, 5d]

; Compute the solution:
result = NEWTON(X, 'newtfunc')

; Print the result:
PRINT, 'For X=[1.0, 5.0], result = ', result

;Try a different starting point.
X = [1d, -1d]

; Compute the solution:
result = NEWTON(X, 'newtfunc')

;Print the result.
PRINT, 'For X=[1.0, -1.0], result = ', result

END
```

IDL prints:

```
For X=[1.0, 5.0], result = -2.4871776e-006      3.0000025
For X=[1.0, -1.0], result =      3.0000000 -2.9985351e-008
```

Version History

Introduced: 4.0

See Also

[BROYDEN](#), [FX_ROOT](#), [FZ_ROOTS](#)

NORM

The NORM function computes the norm of a vector or a two-dimensional array.

By default, NORM computes the L_2 (Euclidean) norm for vectors, and the L_∞ norm for arrays. You may use the LNORM keyword to specify different norms.

This routine is written in the IDL language. Its source code can be found in the file `norm.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = NORM(*A* [, /DOUBLE] [, LNORM={0 | 1 | 2 | *n*}])

Return Value

Returns the Euclidean or infinity norm of a vector or an array.

Arguments

A

A can be either a real or complex vector, or a real or complex two-dimensional array.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

LNORM

Set this keyword to indicate which norm to compute. If *A* is a vector, then the possible values of this keyword are:

Value	Description
0	Compute the L_∞ norm, defined as <code>MAX(ABS(A))</code> .
1	Compute the L_1 norm, defined as <code>TOTAL(ABS(A))</code> .

Table 3-91: LNORM Keyword Values (Vector)

Value	Description
2	Compute the L_2 norm, defined as $\text{SQRT}(\text{TOTAL}(\text{ABS}(A)^2))$. This is the default.
n	Compute the L_n norm, defined as $(\text{TOTAL}(\text{ABS}(A)^n))^{(1/n)}$ where n is any number, float-point or integer.

Table 3-91: LNORM Keyword Values (Vector)

If A is a two-dimensional array, then the possible values of this keyword are:

Value	Description
0	Compute the L_∞ norm (the maximum absolute row sum norm), defined as $\text{MAX}(\text{TOTAL}(\text{ABS}(A), 1))$. This is the default.
1	Compute the L_1 norm (the maximum absolute column sum norm), defined as $\text{MAX}(\text{TOTAL}(\text{ABS}(A), 2))$.
2	Compute the L_2 norm (the spectral norm) defined as the largest singular value, computed from LA_SVD .

Table 3-92: LNORM Keyword Values (Two-Dimensional Array)

Examples

```

; Define an n-element complex vector A:
A = [COMPLEX(1, 0), COMPLEX(2,-2), COMPLEX(-3,1)]

; Compute the Euclidean norm of A and print:
PRINT, 'Euclidian Norm of A =', NORM(A)

; Define an m by n complex array B:
B = [[COMPLEX(1, 0), COMPLEX(2,-2), COMPLEX(-3,1)], $
      [COMPLEX(1,-2), COMPLEX(2, 2), COMPLEX(1, 0)]]

;Compute the Infinity norm of B and print.
PRINT, 'Infinity Norm of B =', NORM(B, /DOUBLE)

```

IDL prints:

```

Euclidian Norm of A =      4.35890
Infinity Norm of B =      6.9907048

```

Version History

Introduced: Pre 4.0

See Also

[COND](#), [LA_SVD](#)

OBJ_CLASS

The OBJ_CLASS function identifies the name of the object class or superclass of the given argument.

Syntax

```
Result = OBJ_CLASS( [Arg] [, COUNT=variable] [, /SUPERCLASS {must specify Arg}] )
```

Return Value

Returns a string containing the name of the class or superclass. If the supplied argument is not an object, a null string is returned. If no argument is supplied, OBJ_CLASS returns an array containing the names of all known object classes in the current IDL session.

Arguments

Arg

A scalar object reference or string variable for which the object class name is desired. If *Arg* is an object reference, its object class definition is used. If *Arg* is a string, it is taken to be the name of the class for which information is desired. Passing a string argument is primarily useful in conjunction with the SUPERCLASS keyword.

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of names returned by OBJ_CLASS. It can be used to determine how many superclasses a class has when the SUPERCLASS keyword is specified.

SUPERCLASS

Set this keyword to cause OBJ_CLASS to return the names of the object's *direct* superclasses as a string array, one element per superclass. The superclasses are ordered in the order they appear in the class structure declaration. In the case where the class has no superclasses, a scalar null string is returned, and the COUNT

keyword (if specified) returns the value 0. If SUPERCLASS is specified, the Arg argument must also be supplied.

Note

Some IDL objects are subclassed from more generic IDL objects. You may see references to the generic IDL objects when using the SUPERCLASS keyword to the OBJ_CLASS function to get information on an object. You may also notice that the generic objects are not documented in the “[Object Class and Method Reference](#)” in the *IDL Reference Guide* manual. This is not an oversight.

We have chosen not to document the workings of some of the generic objects from which IDL objects are subclassed because we reserve the right to make changes to their operation. We strongly recommend that you do not use the undocumented object classes directly, or subclass your own object classes from them. RSI does not guarantee that user-written code that uses undocumented features will continue to function in future releases of IDL.

Version History

Introduced: 5.0

OBJ_DESTROY

The OBJ_DESTROY procedure is used to destroy an object. If the class (or one of its superclasses) supplies a procedure method named CLEANUP, the method is called and all arguments and keywords passed by the user are passed to it. This method should perform any required cleanup on the object and return. Whether a CLEANUP method actually exists or not, IDL will destroy the heap variable representing the object and return.

Note

OBJ_DESTROY does not recurse. That is, if `object1` contains a reference to `object2`, destroying `object1` will *not* destroy `object2`. Take care not to lose the only reference to an object by destroying an object that contains that reference. Recursive cleanup of object hierarchies is a good job for a CLEANUP method.

Syntax

OBJ_DESTROY, *ObjRef* [, *Arg*₁, ..., *Arg*_{*n*}]

Arguments

ObjRef

The object reference for the object to be destroyed. *ObjRef* can be an array, in which case all of the specified objects are destroyed in turn. If the NULL object reference is passed, OBJ_DESTROY ignores it quietly.

Arg₁...Arg_{*n*}

Any arguments accepted by the CLEANUP method for the object being destroyed can be specified as additional arguments to OBJ_DESTROY.

Keywords

Any keywords accepted by the CLEANUP method for the object being destroyed can be specified as keywords to OBJ_DESTROY.

Version History

Introduced: 5.0

OBJ_ISA

When one object class is subclassed (inherits) from another class, there is an “Is A” relationship between them. The OBJ_ISA function is used to determine if an object instance is subclassed from the specified class.

Syntax

Result = OBJ_ISA(*ObjectInstance*, *ClassName*)

Return Value

OBJ_ISA returns True (1) if the specified variable is an object and has the specified class in its inheritance graph, or False (0) otherwise.

Arguments

ObjectInstance

A scalar or array variable for which the OBJ_ISA test should be performed. The result is of type byte, and has the same size and organization as *ObjectInstance*.

ClassName

A string giving the name of the class for which *ObjectInstance* is being tested.

Keywords

None.

Version History

Introduced: 5.0

OBJ_NEW

Given the name of a structure that defines an object class, the OBJ_NEW function returns an object reference to a new instance of the specified object type by carrying out the following operations in order:

1. If the class structure has not been defined, IDL will attempt to find and call a procedure to define it automatically. (See [Chapter 23, “Object Basics”](#) in the *Building IDL Applications* manual for details.) If the structure is still not defined, OBJ_NEW fails and issues an error.
2. If the class structure has been defined, OBJ_NEW creates an object heap variable containing a zeroed instance of the class structure.
3. Once the new object heap variable has been created, OBJ_NEW looks for a *method* function named *Class::INIT* (where *Class* is the actual name of the class). If an INIT method exists, it is called with the new object as its implicit SELF argument, as well as any arguments and keywords specified in the call to OBJ_NEW. If the class has no INIT method, the usual method-searching rules are applied to find one from a superclass. For more information on methods and method-searching rules, see [“Method Routines”](#) in Chapter 23 of the *Building IDL Applications* manual.

The INIT method is expected to initialize the object instance data as necessary to meet the needs of the class implementation. INIT should return a scalar TRUE value (such as 1) if the initialization is successful, and FALSE (such as 0) if the initialization fails.

Note

OBJ_NEW does not call all the INIT methods in an object’s class hierarchy. Instead, it simply calls the first one it finds. Therefore, the INIT method for a class should call the INIT methods of its direct superclasses as necessary.

4. If the INIT method returns true, or if no INIT method exists, OBJ_NEW returns an object reference to the heap variable. If INIT returns false, OBJ_NEW destroys the new object and returns the NULL object reference, indicating that the operation failed. Note that in this case the CLEANUP method is not called. See [“Destruction”](#) in Chapter 23 of the *Building IDL Applications* manual for more on CLEANUP methods.

If called without arguments, OBJ_NEW returns a NULL object reference. The NULL object reference is a special value that never refers to a value object. It is primarily used as a placeholder in structure definitions, and as the initial value for elements of

object arrays created via OBJARR. The null object reference is useful as an indicator that an object reference is currently not usable.

Syntax

```
Result = OBJ_NEW( [ObjectClassName [, Arg1.....Argn]] )
```

Return Value

Returns a reference to a new instance of the specified object type. If called without arguments, OBJ_NEW returns a NULL object reference. The NULL object reference is a special value that never refers to a value object. It is primarily used as a placeholder in structure definitions, and as the initial value for elements of object arrays created via OBJARR. The null object reference is useful as an indicator that an object reference is currently not usable.

Arguments

ObjectClassName

String giving the name of the structure type that defines the object class for which a new object should be created.

If *ObjectClassName* is not provided, OBJ_NEW does not create a new heap variable, and returns the *Null Object*, which is a special object reference that is guaranteed to never point at a valid object heap variable. The null object is a convenient value to use when defining structure definitions for fields that are object references, since it avoids the need to have a pre-existing valid object reference.

Arg₁...Arg_n

Any arguments accepted by the INIT method for the class of object being created can be specified when the object is created.

Keywords

Any keywords accepted by the INIT method for the class of object being created can be specified when the object is created.

Version History

Introduced: 5.0

OBJ_VALID

The OBJ_VALID function verifies the validity of its argument object references, or alternatively returns a vector of references to valid objects.

Syntax

```
Result = OBJ_VALID( [Arg] [, /CAST] [, COUNT=variable] )
```

Return Value

If called with an integer or array of object references as its argument, OBJ_VALID returns a byte array of the same size as the argument. Each element of the result is set to True (1) if the corresponding object reference in the argument refers to an existing object, and False (0) otherwise.

If called with an integer or array of integers as its argument and the CAST keyword is set, OBJ_VALID returns an array of object references. Each element of the result is a reference to the heap variable indexed by the integer value. Integers used to index heap variables are shown in the output of the HELP and PRINT commands. This is useful primarily in programming/debugging when the you have lost a reference but see it with HELP and need to get a reference to it interactively in order to determine what it is and take steps to fix the code. See the “[Examples](#)” section below for an example.

If no argument is specified, OBJ_VALID returns a vector of references to all existing valid objects. If no valid objects exist, a scalar null object reference is returned.

Arguments

Arg

An object reference or array of object references. If the CAST keyword is set, then *Arg* should contain a scalar integer or array of integers.

Keywords

CAST

Set this keyword to cast integers contained in *Arg* to their corresponding object references, and to return the object references in *Result*. If an integer contained in *Arg* does not correspond to a valid object reference, a null object is returned in that

integer's place. If *Arg* contains object references rather than integers, the object references are simply returned. See the “[Examples](#)” section below for an example.

COUNT

Set this keyword equal to a named variable that will contain the number of currently valid objects. This value is returned as a longword integer.

Examples

To determine if a given object reference refers to a valid heap variable, use:

```
IF (OBJ_VALID(obj)) THEN ...
```

To destroy all existing pointer heap variables:

```
OBJ_DESTROY, OBJ_VALID()
```

You can use the `CAST` keyword to “reclaim” lost object references. For example:

```
; Create a class structure:
junk = {junk, data1:0, data2:0.0}

; Create an object:
A = OBJ_NEW('junk')

; Find the integer index:
PRINT, A

; In this case, the integer index to the heap variable is 3. If we
; reassign the variable A, we will "lose" the object reference, but
; the heap variable will still exist.
; Lose the object reference:
A = 0
PRINT, A, OBJ_VALID()

; We can reclaim the lost heap variable using the CAST keyword:
A = OBJ_VALID(3, /CAST)
PRINT, A
```

IDL prints:

```
<ObjHeapVar3(JUNK)>
0 <ObjHeapVar3(JUNK)>
<ObjHeapVar3(JUNK)>
```

Version History

Introduced: 5.0

OBJARR

The OBJARR function returns an object reference vector or array. The individual elements of the array are set to the NULL object reference.

Syntax

$$Result = \text{OBJARR}(D_1 [, \dots, D_8] [, \text{NOZERO}])$$

Return Value

Returns an object reference to an array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

OBJARR sets every element of the result to the null object reference. If NOZERO is nonzero, this initialization is not performed and OBJARR executes faster.

Warning

If you specify NOZERO, the resulting array will have whatever value happens to exist at the system memory location that the array is allocated from. You should be careful to initialize such an array to valid object reference values.

Examples

Create a 3 element by 3 element object reference array with each element containing the null object reference:

```
A = OBJARR(3, 3)
```

Version History

Introduced: 5.0

ON_ERROR

The ON_ERROR procedure determines the action taken when an error is detected inside an IDL user procedure or function by setting state information applying to the current routine and all nested routines. If an override exists within the nested routine, it takes precedence over the ON_ERROR call.

Syntax

ON_ERROR, *N*

Arguments

N

An integer that specifies the action to take. Valid values for *N* are:

- 0: Stop at the statement in the procedure that caused the error, the default action.
- 1: Return all the way back to the main program level.
- 2: Return to the caller of the program unit that established the ON_ERROR condition.
- 3: Return to the program unit that established the ON_ERROR condition.

Keywords

None.

Version History

Introduced: Original

See Also

[CATCH](#), [MESSAGE](#), [ON_IOERROR](#), and [Chapter 19, “Controlling Errors”](#) in the *Building IDL Applications* manual.

ON_IOERROR

The ON_IOERROR procedure specifies a statement to be jumped to if an I/O error occurs in the current procedure. Normally, when an I/O error occurs, an error message is printed and program execution is stopped. If ON_IOERROR is called and an I/O related error later occurs in the same procedure activation, control is transferred to the designated statement with the error code stored in the system variable !ERROR_STATE. The text of the error message is contained in !ERROR_STATE.MSG.

The effect of ON_IOERROR can be canceled by using the label “NULL” in the call.

Syntax

ON_IOERROR, *Label*

Arguments

Label

Statement to jump to when I/O error is encountered.

Keywords

None.

Examples

The following code segment reads an integer from the keyboard. If an invalid number is entered, the program re-prompts.

```
i = 0 ; Number to read:

valid = 0 ; Valid flag

WHILE valid EQ 0 DO BEGIN
  ON_IOERROR, bad_num
  READ, 'Enter Number: ', i
  ;If we get here, i is good.
  VALID = 1
bad_num: IF ~ valid THEN $
  PRINT, 'You entered an invalid number.'
ENDWHILE
END
```

Version History

Introduced: Original

See Also

[CATCH](#), [MESSAGE](#), [ON_ERROR](#), and Chapter 19, “Controlling Errors” in the *Building IDL Applications* manual.

ONLINE_HELP

The `ONLINE_HELP` procedure invokes IDL's online help system. If called with no arguments, it starts the help viewer with the default IDL help file displayed.

Note

This procedure is intended for use in user-written routines. The `?` command, which is a shorthand for the `ONLINE_HELP` procedure, is intended for use at the IDL command line.

The online help viewer used by `ONLINE_HELP` depends on the operating system in use and the type of help document specified:

UNIX: IDL online help documents are provided in Adobe Acrobat PDF format. To use `ONLINE_HELP` (or the `?` command), you must have version 4.0.5 or later of either the Acrobat Reader or the full Acrobat application installed on your system, and the corresponding `acroread` command must be available in a directory included in your UNIX `PATH` environment variable. See the *Installing and Licensing IDL* manual for details.

Note

The Acrobat Reader application is available free of charge from Adobe at <http://www.adobe.com>. In addition, the most current version of Acrobat Reader available for each platform supported by IDL at the time of IDL's release is included on the IDL CD-ROM.

In addition, if a web browser is installed on the system and available for use by IDL, HTML files can be displayed. See "[Displaying HTML Files under UNIX](#)" on page 1504 for details.

Windows: IDL online help is built around the standard Windows help facilities. Windows HTML Help files (`.chm`) as well as traditional Windows help files (`.hlp`) can be displayed. In addition, if Adobe Acrobat is available on the system, PDF files can be displayed. If a web browser is available on the system, HTML files can be displayed. `ONLINE_HELP` will automatically determine which help format to use based on the file name.

For a more complete description of the Unix and Windows help systems, see "[Providing Online Help For Your Application](#)" in Chapter 20 of the *Building IDL Applications* manual.

Syntax

ONLINE_HELP [, *Value*] [, BOOK=*'filename'*] [, /FULL_PATH] [, /QUIT]

UNIX-Only Keywords: [, /FOLD_CASE] [, PAGE=*pageno*]
[, /SUPPRESS_PLUGIN_ERRORS]

Windows-Only Keywords: [, /CONTEXT] [, /TOPICS]

Arguments

Value

An optional string that contains the name of a topic to be displayed.

Under UNIX, *Value* is the name of a topic to be displayed. If *Value* is omitted, if the specified topic does not exist in the specified or default file, or if the IDL Acrobat plug-in is not present, IDL displays a default topic. (For more information on the IDL Acrobat plug-in, see “About IDL’s Online Help System” in Chapter 20 of the *Building IDL Applications* manual.)

Under Windows, *Value* is loaded into the help viewer’s Index dialog. If *Value* is omitted, the specified or default file is displayed at its beginning. If the CONTEXT keyword is set, *Value* should be an integer value (not a string) that represents the *context number* of the help topic to be displayed; the specified topic will be displayed immediately, bypassing the Index dialog. If the BOOK keyword specifies a PDF file, *Value* is ignored and the specified file is opened to the first page of the file.

Keywords

BOOK

Set this keyword to a string containing the name of the help file to be displayed. If the BOOK keyword is omitted, the default IDL help file is displayed.

- Under UNIX, if the *Value* parameter is specified, the default IDL help file is `reference.pdf`. If the *Value* parameter is not specified, the default IDL help file is `onlguide.pdf`. Both files are located in the `Help` subdirectory of the IDL distribution.
- Under Windows, the default IDL help file is `idl.chm`, located in the `Help` subdirectory of the IDL distribution.

If the `FULL_PATH` keyword is specified, `BOOK` must specify a complete file path, including the directory specification and file extension for the file. If `FULL_PATH` is not specified:

- The file extension is optional and may be omitted.
- `ONLINE_HELP` will search the directories given by the `!HELP_PATH` environment variable to locate the file.

Any file specified by this keyword must be in the appropriate format for the viewer being invoked:

- Under UNIX, the file must be either a PDF file (`.pdf`) or an HTML file (`.html` or `.htm`).
- Under Microsoft Windows, the file must be either an HTML Help file (`.chm`), a WinHelp file (`.hlp`), a PDF file (`.pdf`), or an HTML file (`.html` or `.htm`). If you specify the file extension, IDL will use the appropriate viewer for that type of file.

Note

If no file extension is included in the value of the `BOOK` keyword, IDL will search each directory in `!HELP_PATH` until it finds a matching file with one of the following file extensions, in this order: `.chm` (Windows only), `.hlp` (Windows only), `.pdf`, `.html`, `.htm`. You can override this behavior by explicitly specifying the desired file extension.

CONTEXT

This keyword is only available on Windows platforms.

Set this keyword to indicate that the *Value* argument is an integer value that represents the context number of the help topic to be displayed. This keyword is intended for use with user-compiled help files that contain topics that have been mapped to specific context numbers when they were compiled using the `[MAP]` section of the help project file. Specifying a non-existent context number causes an error dialog to be displayed. For more information on how to create Help files with context numbers, see the documentation for the Help system compiler that you are using.

FOLD_CASE

This keyword is only available on UNIX platforms.

Normally, the string given by the *Value* argument is folded to upper case before being handed to Acrobat for display. Explicitly set `FOLD_CASE=0` to indicate that the string should be handed to Acrobat without modification.

FULL_PATH

Set this keyword to indicate that value of the BOOK keyword is a full and complete path to the help file, including any necessary directory information, and a file extension. If FULL_PATH is not specified, ONLINE_HELP searches the !HELP_PATH system variable to locate the file, and the file extension is optional.

PAGE

This keyword is only available on UNIX platforms.

Set this keyword equal to a page number. Acrobat will open the specified page in the specified PDF file.

QUIT

Set this keyword to close the Help viewer.

Note

The QUIT keyword will close the Windows help viewers. It will close the Adobe Acrobat Reader if the IDL-Acrobat plug-in is installed. For other applications launched by ONLINE_HELP, this keyword has no effect.

SUPPRESS_PLUGIN_ERRORS

Under Unix, if the IDL-Acrobat plug-in is not available for your current platform, ONLINE_HELP will issue warning messages explaining that it is unable to position the document, and that the user will need to manually navigate to the desired information once the Acrobat reader application is running. Set this keyword to prevent these warnings from being issued. On non-Unix platforms, this keyword is quietly ignored.

TOPICS

This keyword is only available on Windows platforms.

Set this keyword to display the Index dialog for the specified help file.

Obsolete Keywords

The following keywords are obsolete:

- HTML_HELP

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Displaying HTML Files under UNIX

To display an HTML file on a Windows system, IDL simply instructs Windows to open the specified file and leaves it to the operating system to choose the correct application based on the file type. UNIX systems have no cross-platform standard facility for associating different types of files with specific applications, so the situation is slightly more complicated.

When the `BOOK` keyword specifies an HTML file (that is, when the file name ends with any of `.html`, `.HTML`, `.htm`, or `.HTM`), IDL calls a shell script located in the IDL distribution. The shell script then launches a web browser with the specified file as its argument. By default, IDL assumes that it should launch the Netscape web browser, and that the `netscape` command is found in one of the directories specified by the `PATH` environment variable. Individual users can override the default behavior by setting environment variables to specify either a different browser or an entirely different shell script.

See the comments in the `online_help_html` script located in the `bin` subdirectory of the IDL distribution for details on setting the relevant environment variables.

Examples

The following example uses the `ONLINE_HELP` procedure to launch the help viewer to display information on the `FFT` function:

```
ONLINE_HELP, 'fft'
```

This example displays the *External Development Guide* in PDF format:

```
ONLINE_HELP, BOOK='edg'
```

This Windows-only example displays the topic corresponding to context number 100 in a traditional Windows help file.

```
ONLINE_HELP, 100, /CONTEXT, /FULL_PATH, $  
BOOK='C:\keith\myfile.hlp'
```

This cross-platform example displays an HTML file in the system's default web browser.

```
ONLINE_HELP, BOOK='myfile.html'
```

Version History

Introduced: 4.0.1

Modified to invoke Adobe Acrobat under UNIX, to invoke the HTML Help viewer by default under Windows, to allow opening of HTML and PDF files on all platforms, and to add the UNIX-only keywords FOLD_CASE and PAGE: 5.6

SUPPRESS_PLUGIN_ERRORS keyword: 6.1

See Also

[MK_HTML_HELP](#), [ONLINE_HELP_PDF_INDEX](#), Chapter 20, “Providing Online Help For Your Application” in the *Building IDL Applications* manual

ONLINE_HELP_PDF_INDEX

The `ONLINE_HELP_PDF_INDEX` procedure displays a searchable index of the IDL PDF documentation set. It is available only on UNIX platforms that support the IDL-Acrobat plug-in. (For more information on the IDL Acrobat plug-in, see [“About IDL’s Online Help System”](#) in Chapter 20 of the *Building IDL Applications* manual.)

Note

`ONLINE_HELP_PDF_INDEX` is called automatically by IDL’s `ONLINE_HELP` procedure or “?” functionality. It is not intended to be called directly by entering the routine name at the IDL command line.

`ONLINE_HELP_PDF_INDEX` is a widget-based graphical application. The interface and its controls are described in [“Using `ONLINE_HELP_PDF_INDEX`”](#) on page 1507.

Warning

The `ONLINE_HELP_PDF_INDEX` procedure relies on the presence of the file `mindex.txt` in the `Help` subdirectory of the IDL distribution. If this file is not present, `ONLINE_HELP_PDF_INDEX` will exit with an error.

This routine is written in the IDL language. Its source code can be found in the file `online_help_pdf_index.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
ONLINE_HELP_PDF_INDEX [, SearchTerm]
```

Arguments

SearchTerm

A scalar string containing a term to be located in the IDL master index. *SearchTerm* will be loaded into the `ONLINE_HELP_PDF_INDEX` widget application’s search field, and the index list will scroll to the top-level index entry that most closely matches *SearchTerm*.

Note

See [“The “Always Show This List” Checkbox”](#) on page 1508 for information on modifying this behavior.

Keywords

None.

Using ONLINE_HELP_PDF_INDEX

The `ONLINE_HELP_PDF_INDEX` utility presents a widget interface with two tabs: one that allows searching in and selecting items from the *IDL Master Index*, and one that allows the user to define topics of interest within the IDL PDF documentation set.

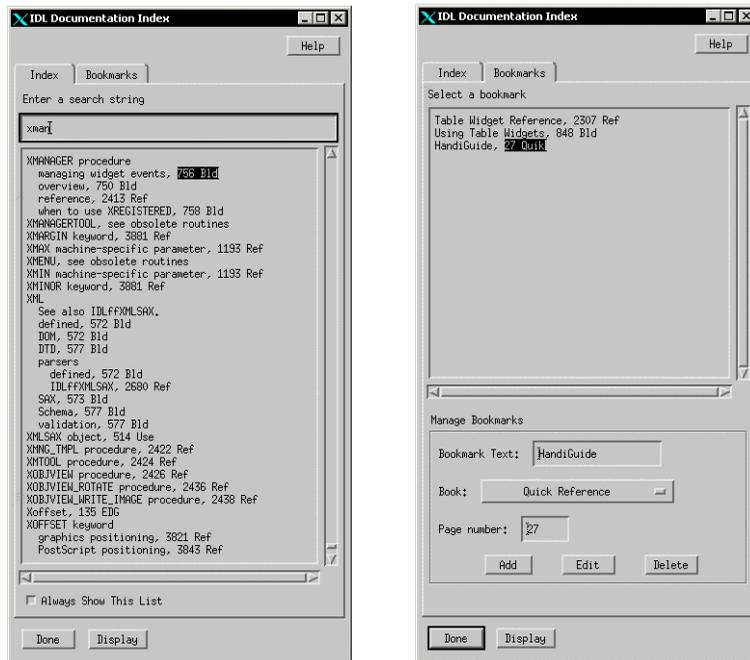


Figure 3-49: The `ONLINE_HELP_PDF_INDEX` Interface

Using the Index Tab

The *IDL Master Index* is a single document that includes index entries for the entire IDL documentation set. It is included in the `help` subdirectory of the IDL distribution in an Adobe Acrobat PDF version (`mindex.pdf`) and a text-only version (`mindex.txt`).

Selecting and Displaying Topics

To use the `ONLINE_HELP_PDF_INDEX` interface to search for a term in the master index, select the **Index** tab and type into the **Enter a search string** field. The index list will scroll automatically to the top-level index entry that most closely matches the string you enter, and the first page number/book abbreviation combination will be highlighted.

To display the selected page in the Adobe Acrobat viewer, click **Display**, press the **Enter** key, or double-click on the highlighted entry using the mouse.

To switch between the search field and the index list, press the **Tab** key. When the index list is selected, change the highlighted item using the arrow keys on your keyboard.

Click **Done** to dismiss the Index widget.

The “Always Show This List” Checkbox

By default, the `ONLINE_HELP_PDF_INDEX` interface is displayed every time the “?” or `ONLINE_HELP` command is used, even if *SearchTerm* is found and displayed in the Adobe Acrobat viewer. Unchecking the **Always Show This List** checkbox on the **Index** tab changes this behavior, and only displays the interface if *SearchTerm* is not found in the PDF documentation set.

Using the Bookmarks Tab

To define personal topics of interest in the IDL documentation set, select the **Bookmarks** tab.

To display the page associated with a bookmark in the Adobe Acrobat viewer:

1. Highlight the bookmark using the mouse or arrow keys.
2. Click **Display**, press the **Enter** key, or double-click on the highlighted entry using the mouse.

To add a new bookmark:

1. Enter a descriptive string in the **Bookmark text** field.
2. Select a manual from the IDL documentation set from the **Book** pulldown menu.
3. Enter the page number in the **Page number** field.
4. Click **Add**.

To modify an existing bookmark:

1. Highlight the bookmark in the list.
2. Make the appropriate changes in the **Bookmark text**, **Book** pulldown list, and **Page number** fields and click **Edit**.

To delete a bookmark, highlight the bookmark in the list and click **Delete**.

Note

There must be at least one bookmark. If you delete the only bookmark in the bookmarks list, a new default bookmark will be created for you.

About Bookmarks

Each IDL user on a UNIX system has a personal bookmarks file that can be used to store index-like references to pages in IDL's PDF documentation set.

Note

Bookmarks into the PDF documentation set will work only for the version of IDL with which they were created.

Like index entries, bookmarks refer to a specific page in one of the IDL manuals. Because page numbers generally change when a new version of an IDL manual is released, bookmarks from one release of IDL will typically not point to the same information in the PDF files provided with a different release. This means that when you install and run a new version of IDL, your existing bookmarks will no longer be valid, and they will not be copied to the new bookmarks file.

Examples

On a UNIX platform that supports the IDL-Acrobat plug-in, entering “?” with no search term at the IDL command prompt displays the `ONLINE_HELP_PDF_INDEX` interface.

Version History

Introduced: 6.1

See Also

[ONLINE_HELP](#), “About IDL’s Online Help System” in Chapter 20 of the *Building IDL Applications* manual

OPEN

The three OPEN procedures open a specified file for input and/or output.

- OPENR (OPEN Read) opens an existing file for input only.
- OPENW (OPEN Write) opens a new file for input and output. If the file exists, it is truncated and its old contents are destroyed.
- OPENU (OPEN Update) opens an existing file for input and output.

Syntax

There are three forms of the OPEN procedure:

```
OPENR, Unit, File
OPENW, Unit, File
OPENU, Unit, File
```

Keywords (all platforms): [, /APPEND | , /COMPRESS] [, BUFSIZE={0 | 1 | value>512}] [, /DELETE] [, ERROR=variable] [, /F77_UNFORMATTED] [, /GET_LUN] [, /MORE] [, /NOEXPAND_PATH] [, /STDIO] [, /SWAP_ENDIAN] [, /SWAP_IF_BIG_ENDIAN] [, /SWAP_IF_LITTLE_ENDIAN] [, /VAX_FLOAT] [, WIDTH=value] [, /XDR]

UNIX-Only Keywords: [, /RAWIO]

Arguments

Unit

The unit number to be associated with the opened file.

File

A string containing the name of the file to be opened. Under UNIX, the filename can contain any wildcard characters recognized by the shell specified by the SHELL environment variable. However, it is faster not to use wildcards because IDL doesn't use the shell to expand file names unless it has to.

Note

The optional *Record_Length* argument is obsolete, and should not be used in new code. See [Appendix I, “Obsolete Features”](#) for details.

Keywords

Note

Platform-specific keywords are listed at the end of this section.

APPEND

Set this keyword to open the file with the file pointer at the end of the file, ready for data to be appended. Normally, the file is opened with the file pointer at the beginning of the file. Under UNIX, use of APPEND prevents OPENW from truncating existing file contents. The APPEND and COMPRESS keywords are mutually exclusive and cannot be specified together.

BUFSIZE

Set this keyword to a value greater than 512 to specify the size of the I/O buffer (in bytes) used when reading and writing files. Setting BUFSIZE=1 (or any other value less than 512) sets the buffer to the default size, which is platform-specific. Set BUFSIZE=0 to disable I/O buffering.

Note that the buffer size is only changeable when reading and writing stream files. Under UNIX, the RAWIO keyword must not be set. Also note that the system stdio may choose to ignore the buffer size setting.

COMPRESS

If COMPRESS is set, IDL reads and writes all data to the file in the standard GZIP format. IDL's GZIP support is based on the freely available ZLIB library by Mark Adler and Jean-loup Gailly (see www.zlib.org for details). This means that IDL's compressed files are 100% compatible with the widely available gzip and gunzip programs. COMPRESS cannot be used with the APPEND keyword.

DELETE

Set this keyword to delete the file when it is closed.

Warning

Setting the DELETE keyword *causes the file to be deleted* even if it was opened for read-only access. In addition, once a file is opened with this keyword, there is no way to cancel its operation.

ERROR

A named variable to place the error status in. If an error occurs in the attempt to open *File*, IDL normally takes the error handling action defined by the `ON_ERROR` and/or `ON_IOERROR` procedures. `OPEN` always returns to the caller without generating an error message when `ERROR` is present. A nonzero error status indicates that an error occurred. The error message can then be found in `!ERROR_STATE.MSG`.

For example, statements similar to the following can be used to detect errors:

```
; Try to open the file demo.dat:
OPENR, 1, 'demo.dat', ERROR = err

; If err is nonzero, something happened. Print the error message to
; the standard error file (logical unit -2):
IF (err NE 0) then PRINTF, -2, !ERROR_STATE.MSG
```

F77_UNFORMATTED

Unformatted variable-length record files produced by UNIX FORTRAN programs contain extra information along with the data in order to allow the data to be properly recovered. This method is necessary because FORTRAN input/output is based on record-oriented files, while UNIX files are simple byte streams that do not impose any record structure. Set the `F77_UNFORMATTED` keyword to read and write this extra information in the same manner as `f77(1)`, so that data to be processed by both IDL and FORTRAN. See “[UNIX-Specific Information](#)” in Chapter 11 of the *Building IDL Applications* manual for further details.

GET_LUN

Set this keyword to use the `GET_LUN` procedure to set the value of *Unit* before the file is opened. Instead of using the two statements:

```
GET_LUN, Unit
OPENR, Unit, 'data.dat'
```

you can use the single statement:

```
OPENR, Unit, 'data.dat', /GET_LUN
```

MORE

If `MORE` is set, and the specified *File* is a terminal, then all output to this unit is formatted in a manner similar to the UNIX `more(1)` command and sent to the standard output stream. Output pauses at the bottom of each screen, at which point the user can press one of the following keys:

- Space: Display the next page of text.

- Return: Display the next line of text.
- ‘q’ or ‘Q’: Suppress all remaining output.
- ‘h’ or ‘H’: Display this list of options.

For example, the following statements show how to output a file named text.dat to the terminal:

```

; Open the text file:
OPENR, inunit, 'text.dat', /GET_LUN

; Open the terminal as a file:
OPENW, outunit, '/dev/tty', /GET_LUN, /MORE

; Read the first line:
line = '' & READF, inunit, line

; While there is text left, output it:
WHILE ~ EOF(inunit) DO BEGIN
    PRINTF, outunit, line
    READF, inunit, line
ENDWHILE

; Close the files and deallocate the units:
FREE_LUN, inunit & FREE_LUN, outunit

```

NOEXPAND_PATH

Set this keyword to specify that the *File* argument be used exactly as supplied, without applying the usual file path expansion.

RAWIO

This keyword is only available on UNIX platforms.

Set this keyword to disable all use of the standard UNIX I/O for the file, in favor of direct calls to the operating system. This allows direct access to devices, such as tape drives, that are difficult or impossible to use effectively through the standard I/O.

Using this keyword has the following implications:

- No formatted or associated (ASSOC) I/O is allowed on the file. Only READU and WRITEU are allowed.
- Normally, attempting to read more data than is available from a file causes the unfilled space to be set to zero and an error to be issued. This does not happen with files opened with RAWIO. When using RAWIO, the programmer must check the transfer count, either via the TRANSFER_COUNT keywords to READU and WRITEU, or the FSTAT function.

- The EOF and POINT_LUN functions cannot be used with a file opened with RAWIO.
- Each call to READU or WRITEU maps directly to UNIX read(2) and write(2) system calls. The programmer must read the UNIX system documentation for these calls and documentation on the target device to determine if there are any special rules for I/O to that device. For example, the size of data that can be transferred to many cartridge tape drives is often forced to be a multiple of 512 bytes.

STDIO

Forces the file to be opened via the standard C I/O library (stdio) rather than any other more native OS API that might usually be used. This is primarily of interest to those who intend to access the file from external code, and is not necessary for most files.

Note

If you intend to use the opened file with the READ_JPEG or WRITE_JPEG procedures using their UNIT keyword, you must specify the STDIO keyword to OPEN to ensure that the file is compatible.

The only exception to this rule is if the filename ends in .jpg or .jpeg and the STDIO keyword is not present in the call to OPEN. In this case OPEN uses stdio by default, covering most uses of jpeg files without requiring you to take special steps.

SWAP_ENDIAN

Set this keyword to swap byte ordering for multi-byte data when performing binary I/O on the specified file. This is useful when accessing files also used by another system with byte ordering different than that of the current host.

SWAP_IF_BIG_ENDIAN

Setting this keyword is equivalent to setting SWAP_ENDIAN; it only takes effect if the current system has big endian byte ordering. This keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

Setting this keyword is equivalent to setting SWAP_ENDIAN; it only takes effect if the current system has little endian byte ordering. This keyword does not refer to the byte ordering of the input data, but to the computer hardware.

VAX_FLOAT

The opened file contains VAX format floating point values. This keyword implies little endian byte ordering for all data contained in the file, and supersedes any setting of the `SWAP_ENDIAN`, `SWAP_IF_BIG_ENDIAN`, or `SWAP_IF_LITTLE_ENDIAN` keywords.

The default setting for this keyword is `FALSE`.

Warning

Please read [“Note on Accessing Data in VAX Floating Point Format”](#) on page 1517 before using this feature.

WIDTH

The desired output width. If no output width is specified, IDL uses the following rules to determine where to break lines:

- If the output file is a terminal, the terminal width is used.
- Otherwise, a default of 80 columns is used.

XDR

Set this keyword to open the file for unformatted XDR (eXternal Data Representation) I/O via the `READU` and `WRITEU` procedures. Use XDR to make binary data portable between different machine architectures by reading and writing all data in a standard format. When a file is open for XDR access, the only I/O data transfer procedures that can be used with it are `READU` and `WRITEU`. XDR is described in [“Portable Unformatted Input/Output”](#) in Chapter 11 of the *Building IDL Applications* manual.

Obsolete Keywords

The following keywords are obsolete:

- `BINARY`
- `BLOCK`
- `DEFAULT`
- `EXTENDSIZE`
- `FIXED`
- `FORTRAN`
- `INITIALSIZE`
- `KEYED`
- `LIST`
- `MACCREATOR`
- `MACTYPE`
- `NONE`
- `NOAUTOMODE`
- `NOSTDIO`
- `PRINT`

- SEGMENTED
- SHARED
- STREAM
- SUBMIT
- SUPERSEDE
- TRUNCATE_ON_CLOSE
- UDF_BLOCK
- VARIABLE

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Note on Accessing Data in VAX Floating Point Format

The floating-point number format used by a program such as IDL is determined entirely by the computer hardware upon which it runs. In the early years of computing it was common for different machines to have incompatible floating-point formats. In the 1970s and 1980s, PDP-11 and VAX minicomputers were widely used for scientific computation, and their floating-point format (known as VAX F and D floating) became the *de facto* standard for science. Early versions of IDL used these formats.

In ensuing years, the computing industry has converged upon a floating-point standard known as IEEE 754, commonly referred to as “IEEE floating” or “IEEE arithmetic.” Other formats (including the VAX formats) have diminished in importance. Now, all common computing hardware uses the IEEE format, which has significant advantages over earlier formats:

- Binary data is portable to almost all current and foreseeable computing hardware and operating systems, requiring at most simple byte-swapping.
- Special *Infinity* and *Not A Number (NaN)* values for undefined computations allow exceptional computations to be carried out in a well-defined manner (and at full speed) by modern pipelined computer architectures.

This convergence gained momentum in the 1980s as workstations and personal computers came into prominence. As a result, almost all versions of IDL since 1987 have used IEEE floating point arithmetic, and all current versions of IDL use this format.

Despite the almost universal current use of the IEEE format, valuable older data stored in the VAX floating-point formats exists at various scientific institutions around the world. In order to allow access to this data, IDL is able to read and write data in these formats. The `VAX_FLOAT` keyword to the `OPEN` procedure is used to enable this feature.

When converting between VAX and IEEE formats, you should be aware of the following basic numerical issues in order to get the best results. Translation of

floating-point values from IDL's native IEEE format to the VAX format and back (that is, VAX to IEEE to VAX) is not a completely reversible operation, and should be avoided when possible. There are many cases where the recovered values will differ from the original values, including:

- The VAX floating-point format lacks support for the IEEE special values (*NaN* and *Infinity*). Hence, their special meaning is lost when they are converted to VAX format and cannot be recovered.
- The IEEE and VAX floating formats have intrinsic differences in precision and range, which can cause information to be lost in both directions. When converting from one format to another, IDL rounds the value to the nearest representable value in the target format.

As a practical matter, an initial conversion of existing VAX format data to IEEE cannot be avoided if the data is to be used on modern machines. However, each format conversion can add a small amount of error to the resulting values, so it is important to minimize the number of such conversions. RSI recommends using IEEE/VAX conversions only to read existing VAX format data, and strongly recommends that all new files be created using the native IEEE format. This introduces only a single unavoidable conversion, and minimizes the resulting conversion error.

Examples

The following example opens the IDL distribution file `people.dat` and reads an image from that file:

```
; Open 'people.dat' on file unit number 1. The FILEPATH
; function is used to return the full path name to this
; distribution file.
OPENR, 1, FILEPATH('people.dat', SUBDIR = ['examples', 'data'])

; Define a variable into which the image will be read:
image=BYTARR(192, 192, /NOZERO)

; Read the data:
READU, 1, image

; Display the image:
TV, image
```

Version History

Introduced: Original

See Also

[CLOSE](#), [GET_LUN](#), [POINT_LUN](#), [PRINT/PRINTF](#), [READ/READF](#), [READU](#),
[WRITEU](#)

O PLOT

The OPLOT procedure plots vector data over a previously-drawn plot. It differs from PLOT only in that it does not generate a new axis. Instead, it uses the scaling established by the most recent call to PLOT and simply overlays a plot of the data on the existing axis.

Syntax

```
O PLOT, [X,] Y [, MAX_VALUE=value] [, MIN_VALUE=value] [, NSUM=value]
  [, /POLAR] [, THICK=value]
```

Graphics Keywords: [, CLIP=[X_0 , Y_0 , X_1 , Y_1]] [, COLOR=value]
 [, LINSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, /NOCLIP] [, PSYM=integer{0 to 10}]
 [, SYMSIZE=value] [, /T3D] [, ZVALUE=value{0 to 1}]

Arguments

X

A vector argument. If X is not specified, Y is plotted as a function of point number (starting at zero). If both arguments are provided, Y is plotted as a function of X.

This argument is converted to double-precision floating-point before plotting. Plots created with OPLOT are limited to the range and precision of double precision floating-point values.

Y

The ordinate data to be plotted. This argument is converted to double-precision floating-point before plotting.

Keywords

MAX_VALUE

The maximum value to be plotted. If this keyword is present, data values greater than the value of MAX_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

MIN_VALUE

The minimum value to be plotted. If this keyword is present, data values less than the value of MIN_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

NSUM

The presence of this keyword indicates the number of data points to average when plotting. If NSUM is larger than 1, every group of NSUM points is averaged to produce one plotted point. If there are m data points, then m/NSUM points are displayed. On logarithmic axes a geometric average is performed.

It is convenient to use NSUM when there is an extremely large number of data points to plot because it plots fewer points, the graph is less cluttered, and it is quicker.

POLAR

Set this keyword to produce polar plots. The X and Y vector parameters, both of which must be present, are first converted from polar to Cartesian coordinates. The first parameter is the radius, and the second is expressed in radians.

For example, to make a polar plot, use the command:

```
O PLOT, /POLAR, R, THETA
```

THICK

Controls the thickness of the lines connecting the points. A thickness of 1.0 is normal, 2.0 is double wide, etc.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above. [CLIP](#), [COLOR](#), [LINestyle](#), [NOCLIP](#), [PSYM](#), [SYMSIZE](#), [T3D](#), [ZVALUE](#).

Examples

```
; Create a simple dataset:  
D = SIN(FINDGEN(100))/EXP(FINDGEN(100)/50))  
  
; Create an X-Y plot of vector D:  
PLOT, D
```

```
    ; Overplot the sine of D as a thick, dashed line:  
    OPLOT, SIN(D), LINESSTYLE = 5, THICK = 2
```

Version History

Introduced: Original

See Also

[IPLOT](#), [OPLOTERR](#), [PLOT](#)

OLOTERR

The OLOTERR procedure plots error bars over a previously drawn plot. A plot of X versus Y with error bars drawn from $Y - Err$ to $Y + Err$ is written to the output device over any plot already there.

This routine is written in the IDL language. Its source code can be found in the file `oploterr.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
OLOTERR, [ X, ] Y, Err [, Psym ]
```

Arguments

X

An optional array of X values. The procedure checks whether or not the third parameter passed is a vector to decide if X was passed. If X is not passed, then `INDGEN(Y)` is assumed for the X values.

Y

The array of Y values. Y cannot be of type string.

Err

The array of error bar values.

Psym

The plotting symbol to use (default = +7).

Keywords

None

Version History

Introduced: Original

See Also

[ERRPLOT](#), [IPLLOT](#), [OPLOT](#), [PLOTERR](#)

P_CORRELATE

The P_CORRELATE function computes the partial correlation coefficient of a dependent variable and one particular independent variable when the effects of all other variables involved are removed.

To compute the partial correlation, the following method is used:

- Let Y and X be the variables of primary interest and let $C_1...C_p$ be the variables held fixed.
- First, calculate the residuals after regressing Y on $C_1...C_p$. (These are the parts of Y that cannot be predicted by $C_1...C_p$.)
- Then, calculate the residuals after regressing X on $C_1...C_p$. (These are the parts of X that cannot be predicted by $C_1...C_p$.)
- The partial correlation coefficient between Y and X adjusted for $C_1...C_p$ is the correlation between these two sets of residuals.

This routine is written in the IDL language. Its source code can be found in the file `p_correlate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = P_CORRELATE( X, Y, C [, /DOUBLE] )
```

Return Value

Returns the correlation coefficient.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector that specifies the independent variable data.

Y

An n -element integer, single-, or double-precision floating-point vector that specifies the dependent variable data.

C

An integer, single-, or double-precision floating-point array that specifies the independent variable data whose effects are to be removed. *C* may either be an *n*-element vector containing the independent variable, or a *p*-by-*n* two-dimensional array in which each column corresponds to a separate independent variable.

Keywords**DOUBLE**

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```

; Define three sample populations:
X0 = [64, 71, 53, 67, 55, 58, 77, 57, 56, 51, 76, 68]
X1 = [57, 59, 49, 62, 51, 50, 55, 48, 52, 42, 61, 57]
X2 = [ 8, 10, 6, 11, 8, 7, 10, 9, 10, 6, 12, 9]

; Compute the partial correlation of X0 and X1 with the effects
; of X2 removed.
result = P_CORRELATE(X0, X1, X2)

PRINT, result

```

IDL prints:

```
0.533469
```

Version History

Introduced: 4.0

See Also

[A_CORRELATE](#), [C_CORRELATE](#), [CORRELATE](#), [M_CORRELATE](#),
[R_CORRELATE](#)

PARTICLE_TRACE

The `PARTICLE_TRACE` procedure traces the path of a massless particle through a vector field. The function allows the user to specify a set of starting points and a vector field. The input seed points can come from any vertex-producing process. The points are tracked by treating the vector field as a velocity field and integrating. Each path is tracked until the path leaves the input volume or a maximum number of steps is reached. The vertices generated along the paths are returned packed into a single array along with a polyline connectivity array. The polyline connectivity array organizes the vertices into separate paths (one per seed). Each path has an orientation. The initial orientation may be set using the `SEED_NORMAL` keyword. As a path is tracked, the change in the normal is also computed and may be returned to the user as an optional argument. Path output can be passed directly to an `IDLgrPolyline` object or passed to the `STREAMLINE` procedure for generation of orientated ribbons. Control over aspects of the integration (e.g. method or stepsize) is also provided.

Syntax

```
PARTICLE_TRACE, Data, Seeds, Verts, Conn [, Normals]
    [, MAX_ITERATIONS=value] [, ANISOTROPY=array]
    [, INTEGRATION={0 | 1}] [, SEED_NORMAL=vector]
    [, TOLERANCE=value] [, MAX_STEPSIZE=value] [, /UNIFORM]
```

Arguments

Data

Input data array. This array can be of dimensions $[2, dx, dy]$ for two-dimensional vector fields or $[3, dx, dy, dz]$ for three-dimensional vector fields.

Seeds

Input array of seed points ($[3, n]$ or $[2, n]$).

Verts

Array of output path vertices ($[3, n]$ or $[2, n]$ array of floats).

Conn

Output path connectivity array in `IDLgrPolyline POLYLINES` keyword format. There is one set of line segments in this array for each input seed point.

Normals

Output normal estimate at each output vertex ($[3, n]$ array of floats).

Keywords

ANISOTROPY

Set this input keyword to a two- or three- element array describing the distance between grid points in each dimension. The default value is $[1.0, 1.0, 1.0]$ for three-dimensional data and $[1.0, 1.0]$ for two-dimensional data.

INTEGRATION

Set this keyword to one of the following values to select the integration method:

- 0 = 2nd order Runge-Kutta (the default)
- 1 = 4th order Runge-Kutta

SEED_NORMAL

Set this keyword to a three-element vector which selects the initial normal for the paths. The default value is $[0.0, 0.0, 1.0]$. This keyword is ignored for 2-D data.

TOLERANCE

This keyword is used with adaptive step-size control in the 4th order Runge-Kutta integration scheme. It is ignored if the UNIFORM keyword is set or the 2nd order Runge-Kutta scheme is selected.

MAX_ITERATIONS

This keyword specifies the maximum number of line segments to return for each path. The default value is 200.

MAX_STEPSIZE

This keyword specifies the maximum path step size. The default value is 1.0.

UNIFORM

If this keyword is set, the step size will be set to a fixed value, set via the MAX_STEPSIZE keyword. If this keyword is not specified, and TOLERANCE is either unspecified or inapplicable, then the step size is computed based on the velocity at the current point on the path according to the formula:

$\text{stepsize} = \text{MIN}(\text{MaxStepSize}, \text{MaxStepSize}/\text{MAX}(\text{ABS}(U), \text{ABS}(V), \text{ABS}(W)))$

where (U,V,W) is the local velocity vector.

Version History

Introduced: 5.5

PATH_CACHE

The `PATH_CACHE` procedure is used to control IDL's use of the *path cache*. By default, as IDL searches directories included in the `!PATH` system variable for `.pro` or `.sav` files to compile, it creates an in-memory list of *all* `.pro` and `.sav` files contained in each directory. When IDL later searches for a `.pro` or `.sav` file, before attempting to open the file in a given directory, IDL checks the path cache to determine whether the directory has already been cached. If the directory is included in the cache, IDL uses the cached information to determine whether the file will be found in that directory, and will only attempt to open the file there if the cache tells it that the file exists. By eliminating unnecessary attempts to open files, the path cache speeds the path searching process.

The path cache is enabled by default, and in almost all cases its operation is transparent to the IDL user, save for the boost in path searching speed it provides. Because the cache automatically adjusts to changes made to IDL's path, use of `PATH_CACHE` should not be necessary in typical IDL operation. It is provided to allow complete control over the details of how and when the caching operation is performed.

- For information on when the path cache is *not* used, see [“Situations in which IDL will not use the Path Cache”](#) on page 1532.
- For information on disabling the path cache, see [“Disabling the Path Cache”](#) on page 1533.

Note

Prior to IDL 6.0, IDL did not use a path cache. Aside from the improvement in performance, the behavior of IDL with the path cache is identical to that without in almost all cases. The rare cases in which it differs, and options for disabling its use, are discussed in [“Options for Avoiding Use of the Path Cache”](#) on page 1534.

About the Path Cache

The first time an IDL session attempts to call a function or procedure written in the IDL language, it must locate and compile the file containing the code for that routine. The file containing the routine must have the same name as the routine, with either a `.pro` or a `.sav` extension. After trying to open the file in the user's current working directory, IDL will attempt to open the file in each of the directories listed in the `!PATH` system variable, in the order specified by `!PATH`. The search stops when a file with the desired name is found or no directories remain in `!PATH`.

By default, IDL maintains an in-memory cache of the locations of `.pro` and `.sav` files stored in directories included in the `!PATH` system variable. The path cache is built automatically during normal operation, as IDL searches the directories specified by `!PATH`. Once a directory is cached, IDL knows whether or not it contains a given file, without the need to actually attempt to open that file. This information allows IDL to bypass directories that do not contain the desired file, providing a significant boost in the speed of path searching. The path cache can significantly improve the startup speed of large, object-oriented applications, because method resolution requires extensive path searching.

The path cache operates on a per-directory basis; if IDL searches a directory for a `.pro` or `.sav` file, the locations of *all* `.pro` and `.sav` files in that directory are added to the cache, and the directory is not searched again until the cache is cleared and rebuilt.

Note

The current contents of the path cache can be viewed using the `PATH_CACHE` keyword to the `HELP` procedure.

Syntax

```
PATH_CACHE[, /CLEAR] [, /ENABLE] [, /REBUILD]
```

Arguments

None.

Keywords

CLEAR

Set this keyword to clear the entire contents of the path cache, leaving it completely empty. If path caching is enabled, IDL will begin rebuilding the cache the next time it needs to locate a `.pro` or `.sav` file. If you wish to prevent the rebuilding of the cache, set the `ENABLE` keyword equal to zero as well.

Note

The `.RESET_SESSION` executive command clears the entire path cache as part of resetting the IDL session.

ENABLE

Set this keyword to a non-zero value to specify that IDL should use the path cache when searching for files and also add new directories to the cache as they are opened. Set this keyword to zero to disable use of the cache when searching for files, and to discontinue adding new directories.

Note

Disabling the cache does not cause the current contents of the cache to be discarded. To discard the cache information, specify the CLEAR keyword.

REBUILD

Set this keyword to discard the current contents of the path cache (as if the CLEAR keyword had been specified), and then immediately rebuild the cache by searching the directories specified by the current value of the !PATH system variable for .pro and .sav files.

Note

If !PATH contains many directories, or if access to those directories is slow, rebuilding the cache using this method may also be slow. In many cases, the CLEAR keyword is sufficient, since IDL will rebuild the empty cache as program execution requires it to search for .pro and .sav files.

Situations in which IDL will not use the Path Cache

By default, IDL uses the path cache whenever it tries to locate .pro or .sav files. However, IDL will never use the path cache in the following situations:

Current Working Directory

The path cache is neither checked nor added to if the file being searched for exists in the current working directory. Before IDL searches !PATH for a file to compile, it always looks in the current working directory without checking the cache.

Relative Paths

The path cache does not cache directories specified relative to the current directory, even though relative paths are allowed in the specification of !PATH.

An absolute (or *fully qualified*) path is a path that completely specifies the location of a file. Under UNIX, an absolute path is specified relative to the root of the filesystem, and therefore starts with a slash (/) character. Under Microsoft Windows, an absolute

path starts with a drive letter (C:, for example) or a double backslash (\\) (if the file is specified using the Universal Naming Convention format). In contrast, a relative path is incomplete, and must be interpreted relative to the current working directory of the IDL process. IDL only caches absolute paths.

Executive Commands

The path cache is neither checked nor added to when a `.COMPILE` or `.RUN` executive command is issued. In such cases, IDL performs a standard directory-by-directory search of the directories included in `!PATH`.

IDL_NOCACHE File Present

IDL will not cache the contents of any directory that contains a file named `IDL_NOCACHE`. See [“Marking Specific Directories as Uncacheable”](#) on page 1534 for additional information on this feature.

Path Cache Disabled

IDL will neither check nor add files to the path cache if it has been disabled. See [“Disabling the Path Cache”](#), below, for additional information.

Disabling the Path Cache

By default, IDL caches the locations of `.pro` and `.sav` files in all directories specified by the `!PATH` system variable. Use of the path cache can be fully disabled in the following ways:

1. By issuing the `PATH_CACHE` command with the `ENABLE` keyword set equal to zero. This will disable the path cache until you manually re-enable it, or for the duration of the current IDL session. See the description of the `ENABLE` keyword, above, for details.
2. By unchecking the “Enable Path Caching” checkbox on the Path tab of the IDLDE Preferences dialog. See “Path Preferences” in Chapter 5 of the *Using IDL* manual for details.
3. By defining an environment variable named `IDL_PATH_CACHE_DISABLE` before starting IDL. See “Environment Variables Used by IDL” in Chapter 1 of the *Using IDL* manual for details.

In addition, you can selectively disable use of the path cache for specific directories by creating a file named `IDL_NOCACHE` in the directory. See [“Marking Specific Directories as Uncacheable”](#), below, for details.

Marking Specific Directories as Uncacheable

You can mark specific directories as being uncacheable even though the directory is included in `!PATH`. To do so, create a file named `IDL_NOCACHE` in that directory.

Note

IDL does not inspect the contents of an `IDL_NOCACHE` file; it can contain anything you wish, or nothing at all. Under Unix operating systems, the `IDL_NOCACHE` file must be named exactly as shown, using all uppercase characters in the name. Under Microsoft Windows, the characters can have any case, but RSI suggests you use upper case for consistency.

When IDL encounters a directory containing an `IDL_NOCACHE` file during normal path searching, it makes a special entry in the path cache telling it that the directory must not be cached. Once this is done, all future attempts to locate files in that directory will be done without using cached information.

Note

If the directory to which you add an `IDL_NOCACHE` file has already been added to the path cache for the current IDL session, you must clear the existing cache (using the `CLEAR` keyword to the `PATH_CACHE` procedure) before the no-cache setting will take effect.

To re-enable path caching for a directory that has been marked as uncacheable, remove the `IDL_NOCACHE` file, and then reset IDL's path cache in one of the following ways:

- Specify the `CLEAR` keyword to the `PATH_CACHE` procedure.
- Issue the `.RESET_SESSION` executive command.
- Exit and restart the IDL session.

Options for Avoiding Use of the Path Cache

In most cases, the files contained in directories included in `!PATH` do not change during an IDL session. In such cases the path cache is completely transparent to the IDL user, and serves only to speed compilation of IDL routines. As a result, there is rarely a reason to globally disable the path cache.

If files are created or deleted in a directory included in `!PATH` during an IDL session, the path cache can become confused and provide bad information to IDL about the contents of that directory. There are several ways to handle this situation. The

following list of alternatives is given in rough order of preference, with the easiest and lowest-impact options given first:

1. Leave the path cache enabled, and change your current working directory to the directory in which files are created or deleted. Since IDL checks the current working directory before checking the directories in !PATH, use of the path cache does not affect IDL's ability to find these files.
2. If the addition or deletion of files in a directory included in !PATH is a rare occurrence, leave the path cache enabled and clear it in one of the following ways after the contents of the directory have changed:
 - Specify the CLEAR keyword to the PATH_CACHE procedure.
 - Issue the .RESET_SESSION executive command.
 - Exit and restart the IDL session.
3. Leave the path cache enabled and use the .COMPILE or .RUN executive commands to force the compilation of any file, regardless of the contents of the path cache.
4. If you have a directory (other than your current working directory) in which files are regularly added or deleted during the execution of IDL sessions, you can leave path caching enabled but explicitly disable caching of that specific directory by creating an IDL_NOCACHE file, as described in [“Marking Specific Directories as Uncacheable”](#) on page 1534. This approach works for all IDL sessions that access the directory, and is therefore convenient in long-term or multi-user situations.
5. You can completely disable operation of the path cache using one of the methods described under [“Disabling the Path Cache”](#) on page 1533. This is not recommended, because most directories are not dynamic, and completely disabling path caching sacrifices the performance advantages of caching directories whose contents *are* static.

Note on Behavior at Startup

Depending on the value of your !PATH system variable, you may notice that some directories are being cached immediately when IDL starts up. This will occur if your path definition string includes the <IDL_DEFAULT> token, or if one or more entries include the “+” symbol. In these cases, in order for IDL to build the !PATH system variable, it must inspect subdirectories of the specified directories for the presence of .pro and .sav files, with the side effect of adding these directories to the path cache. See EXPAND_PATH for a discussion of IDL's path expansion behavior.

Examples

The following statement disables path caching for the current session:

```
PATH_CACHE, ENABLE = 0
```

The following statement disables path caching for the current session and throws away the current contents of the cache:

```
PATH_CACHE, ENABLE = 0, /CLEAR
```

Suppose you want to remove a directory included in `!PATH` from the cache without resetting your IDL session. The following statements cause the specified directory not to be included in future caching by creating a file named `IDL_NOCACHE` in that directory:

```
OPENW, UNIT = u, '/home/idluser/idl_dev_dir/IDL_NOCACHE', /GET_LUN
FREE_LUN, u
```

The `OPENW` and `FREE_LUN` statements create an empty file with the desired name in the target directory. Executing the following statement clears the cache so as to reflect the change in the current IDL session:

```
PATH_CACHE, /CLEAR
```

The next time IDL encounters this directory in a path search, it will see the presence of the `IDL_NOCACHE` and make a note in the path cache that the directory is not cacheable.

Note

You can also create the `IDL_NOCACHE` file outside IDL using any convenient command (text editor, Unix `touch` command, *etc.*). If the file is created outside IDL, only the `PATH_CACHE, /CLEAR` statement is necessary.

Version History

Introduced: 6.0

See Also

[.FULL_RESET_SESSION](#), [.RESET_SESSION](#), “[!PATH](#)” in Appendix D, “[Environment Variables Used by IDL](#)” in Chapter 1 of the *Using IDL* manual, “[Path Preferences](#)” in Chapter 5 of the *Using IDL* manual

PATH_SEP

The `PATH_SEP` function returns the proper file path segment separator character for the current operating system. This is the character used by the host operating system for delimiting subdirectory names in a path specification. Use this function instead of hard-coding separators to make code more portable.

This routine is written in the IDL language. Its source code can be found in the file `path_sep.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = PATH_SEP( [ /PARENT_DIRECTORY ] [, /SEARCH_PATH ] )
```

Return Value

Returns a string containing the specified separator character.

Arguments

None.

Keywords

Specify at most one of the following keywords:

PARENT_DIRECTORY

If set, `PATH_SEP` returns the standard directory notation used by the host operating system to indicate the parent of a directory.

SEARCH_PATH

If set, `PATH_SEP` returns the character used to separate entries in a search path.

Version History

Introduced: 5.5

See Also

[FILE_BASENAME](#), [FILE_DIRNAME](#), [FILE_SEARCH](#), [FILEPATH](#)

PCOMP

The PCOMP function computes the principal components of an m -column, n -row array, where m is the number of variables and n is the number of observations or samples. The principal components of a multivariate data set may be used to restate the data in terms of derived variables or may be used to reduce the dimensionality of the data by reducing the number of variables (columns).

This routine is written in the IDL language. Its source code can be found in the file `pcomp.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = PCOMP( A [, COEFFICIENTS=variable] [, /COVARIANCE] [, /DOUBLE]  
               [, EIGENVALUES=variable] [, NVARIABLES=value] [, /STANDARDIZE]  
               [, VARIANCES=variable] )
```

Return Value

The result is an $nvariables$ -column ($nvariables \leq m$), n -row array of derived variables.

Arguments

A

An m -column, n -row, single- or double-precision floating-point array.

Keywords

COEFFICIENTS

Use this keyword to specify a named variable that will contain the principal components used to compute the derived variables. The principal components are the coefficients of the derived variables and are returned in an m -column, m -row array. The rows of this array correspond to the coefficients of the derived variables. The coefficients are scaled so that the sums of their squares are equal to the eigenvalue from which they are computed.

COVARIANCE

Set this keyword to compute the principal components using the covariances of the original data. The default is to use the correlations of the original data to compute the principal components.

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision result. Set `DOUBLE=0` to use single-precision for computations and to return a single-precision result. The default is `/DOUBLE` if *Array* is double precision, otherwise the default is `DOUBLE=0`.

EIGENVALUES

Use this keyword to specify a named variable that will contain a one-column, *m*-row array of eigenvalues that correspond to the principal components. The eigenvalues are listed in descending order.

NVARIABLES

Use this keyword to specify the number of derived variables. A value of zero, negative values, and values in excess of the input array's column dimension result in a complete set (*m*-columns and *n*-rows) of derived variables.

STANDARDIZE

Set this keyword to convert the variables (the columns) of the input array to standardized variables (variables with a mean of zero and variance of one).

VARIANCES

Use this keyword to specify a named variable that will contain a one-column, *m*-row array of variances. The variances correspond to the percentage of the total variance for each derived variable.

Examples

```
PRO ex_pcomp

;Define an array with 4 variables and 20 observations.
array = [[19.5, 43.1, 29.1, 11.9], $
         [24.7, 49.8, 28.2, 22.8], $
         [30.7, 51.9, 37.0, 18.7], $
         [29.8, 54.3, 31.1, 20.1], $
```

```

[19.1, 42.2, 30.9, 12.9], $
[25.6, 53.9, 23.7, 21.7], $
[31.4, 58.5, 27.6, 27.1], $
[27.9, 52.1, 30.6, 25.4], $
[22.1, 49.9, 23.2, 21.3], $
[25.5, 53.5, 24.8, 19.3], $
[31.1, 56.6, 30.0, 25.4], $
[30.4, 56.7, 28.3, 27.2], $
[18.7, 46.5, 23.0, 11.7], $
[19.7, 44.2, 28.6, 17.8], $
[14.6, 42.7, 21.3, 12.8], $
[29.5, 54.4, 30.1, 23.9], $
[27.7, 55.3, 25.7, 22.6], $
[30.2, 58.6, 24.6, 25.4], $
[22.7, 48.2, 27.1, 14.8], $
[25.2, 51.0, 27.5, 21.1]]

;Remove the mean from each variable.
m = 4      ; number of variables
n = 20     ; number of observations
means = TOTAL(array, 2)/n
array = array - REBIN(means, m, n)

;Compute derived variables based upon the principal components.
result = PCOMP(array, COEFFICIENTS = coefficients, $
              EIGENVALUES=eigenvalues, VARIANCES=variances, /COVARIANCE)
PRINT, 'Result: '
PRINT, result, FORMAT = '(4(F8.2))'
PRINT
PRINT, 'Coefficients: '
FOR mode=0,3 DO PRINT, $
    mode+1, coefficients[* ,mode], $
    FORMAT='("Mode#", I1, 4(F10.4))'
eigenvectors = coefficients/REBIN(eigenvalues, m, m)
PRINT
PRINT, 'Eigenvectors: '
FOR mode=0,3 DO PRINT, $
    mode+1, eigenvectors[* ,mode], $
    FORMAT='("Mode#", I1, 4(F10.4))'
array_reconstruct = result ## eigenvectors
PRINT
PRINT, 'Reconstruction error: ', $
    TOTAL((array_reconstruct - array)^2)
PRINT
PRINT, 'Energy conservation: ', TOTAL(array^2),
TOTAL(eigenvalues)*(n-1)
PRINT
PRINT, '      Mode   Eigenvalue   PercentVariance'
FOR mode=0,3 DO PRINT, $

```

```
mode+1, eigenvalues[mode], variances[mode]*100
```

```
END
```

When the above program is compiled and executed, the following output is produced:

```
Result:
```

```
-107.38  13.40  -1.41  -0.03
   3.20   0.70   5.95  -0.02
  32.50  38.66  -3.87   0.01
  40.89  13.79  -4.98  -0.01
-107.24  19.36   1.77   0.02
  18.43 -17.15  -1.47  -0.00
  99.89  -6.23   0.13   0.02
  45.38   8.11   6.53  -0.01
 -21.31 -18.31   3.75  -0.01
   5.54 -11.17  -4.52   0.02
  83.14   4.97   0.09   0.01
  87.11  -3.16   2.81   0.00
-101.32 -11.78  -6.12   0.01
 -73.07   6.24   6.61   0.02
-137.02 -19.10   1.33   0.01
  57.11   6.96   0.84  -0.01
  42.13 -10.07  -2.14   0.01
  83.30 -16.69  -2.72  -0.01
 -54.13   2.56  -4.21  -0.03
   2.84  -1.06   1.62  -0.01
```

```
Coefficients:
```

```
Mode#1   4.8799   5.0568   1.0282   4.7936
Mode#2   1.0147  -0.9545   3.4885  -0.7743
Mode#3  -0.6183  -0.9554   0.2690   1.5796
Mode#4  -0.0900   0.0752   0.0472   0.0022
```

```
Eigenvectors:
```

```
Mode#1   0.0665   0.0689   0.0140   0.0653
Mode#2   0.0690  -0.0649   0.2372  -0.0526
Mode#3  -0.1601  -0.2473   0.0697   0.4089
Mode#4  -5.6290   4.7013   2.9540   0.1372
```

```
Reconstruction error:  1.44876e-010
```

```
Energy conservation:      1748.17      1748.17
```

Mode	Eigenvalue	PercentVariance
1	73.4205	79.7970
2	14.7099	15.9875
3	3.86271	4.19818
4	0.0159915	0.0173803

The first two derived variables account for 96% of the total variance of the original data.

Version History

Introduced: 5.0

See Also

[CORRELATE](#), [EIGENQL](#)

PLOT

The PLOT procedure draws graphs of vector arguments. If one parameter is used, the vector parameter is plotted on the ordinate versus the point number on the abscissa. To plot one vector as a function of another, use two parameters. PLOT can also be used to create polar plots by setting the POLAR keyword.

Syntax

```
PLOT, [X,] Y [, /ISOTROPIC] [, MAX_VALUE=value] [, MIN_VALUE=value]
  [, NSUM=value] [, /POLAR] [, THICK=value] [, /XLOG] [, /YLOG]
  [, /YNOZERO]
```

```
Graphics Keywords: [, BACKGROUND=color_index] [, CHARSIZE=value]
  [, CHARTHICK=integer] [, CLIP=[X0, Y0, X1, Y1]] [, COLOR=value] [, /DATA |
  /DEVICE | /NORMAL] [, FONT=integer] [, LINESSTYLE={0 | 1 | 2 | 3 | 4 | 5}]
  [, /NOCLIP] [, /NODATA] [, /NOERASE] [, POSITION=[X0, Y0, X1, Y1]]
  [, PSYM=integer{0 to 10}] [, SUBTITLE=string] [, SYMSIZE=value] [, /T3D]
  [, THICK=value] [, TICKLEN=value] [, TITLE=string]
  [, {X | Y | Z}CHARSIZE=value]
  [, {X | Y | Z}GRIDSTYLE=integer{0 to 5}]
  [, {X | Y | Z}MARGIN=[left, right]]
  [, {X | Y | Z}MINOR=integer]
  [, {X | Y | Z}RANGE=[min, max]]
  [, {X | Y | Z}STYLE=value]
  [, {X | Y | Z}THICK=value]
  [, {X | Y | Z}TICK_GET=variable]
  [, {X | Y | Z}TICKFORMAT=string]
  [, {X | Y | Z}TICKINTERVAL= value]
  [, {X | Y | Z}TICKLAYOUT=scalar]
  [, {X | Y | Z}TICKLEN=value]
  [, {X | Y | Z}TICKNAME=string_array]
  [, {X | Y | Z}TICKS=integer]
  [, {X | Y | Z}TICKUNITS=string]
  [, {X | Y | Z}TICKV=array]
  [, {X | Y | Z}TITLE=string]
  [, ZVALUE=value{0 to 1}]
```

Arguments

X

A vector argument. If X is not specified, Y is plotted as a function of point number (starting at zero). If both arguments are provided, Y is plotted as a function of X.

This argument is converted to double precision floating-point before plotting. Plots created with PLOT are limited to the range and precision of double-precision floating-point values.

Y

The ordinate data to be plotted. This argument is converted to double-precision floating-point before plotting.

Keywords

ISOTROPIC

Set this keyword to force the scaling of the X and Y axes to be equal.

Note

The X and Y axes will be scaled isotropically and then fit within the rectangle defined by the POSITION keyword; one of the axes may be shortened. See [“POSITION”](#) on page 4339 for more information.

MAX_VALUE

The maximum value to be plotted. If this keyword is present, data values greater than the value of MAX_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

MIN_VALUE

The minimum value to be plotted. If this keyword is present, data values less than the value of MIN_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

NSUM

The presence of this keyword indicates the number of data points to average when plotting. If NSUM is larger than 1, every group of NSUM points is averaged to produce one plotted point. If there are m data points, then m/NSUM points are displayed. On logarithmic axes a geometric average is performed.

It is convenient to use NSUM when there is an extremely large number of data points to plot because it plots fewer points, the graph is less cluttered, and it is quicker.

POLAR

Set this keyword to produce polar plots. The X and Y vector parameters, both of which must be present, are first converted from polar to Cartesian coordinates. The first parameter is the radius, and the second is the angle (expressed in radians). For example, to make a polar plot, you would use a command such as:

```
PLOT, /POLAR, R, THETA
```

THICK

Controls the thickness of the lines connecting the points. A thickness of 1.0 is normal, 2 is double wide, etc.

XLOG

Set this keyword to specify a logarithmic X axis, producing a log-linear plot. Set both XLOG and YLOG to produce a log-log plot. Note that logarithmic axes that have ranges of less than a decade are not labeled.

YNOZERO

Set this keyword to inhibit setting the minimum Y axis value to zero when the Y data are all positive and nonzero, and no explicit minimum Y value is specified (using YRANGE, or !Y.RANGE). By default, the Y axis spans the range of 0 to the maximum value of Y , in the case of positive Y data. Set bit 4 in !Y.STYLE to make this option the default.

YLOG

Set this keyword to specify a logarithmic Y axis, producing a linear-log plot. Set both XLOG and YLOG to produce a log-log plot. Note that logarithmic axes that have ranges of less than a decade are not labeled.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above. [BACKGROUND](#), [CHARSIZE](#), [CHARTHICK](#), [CLIP](#), [COLOR](#), [DATA](#), [DEVICE](#), [FONT](#), [LINSTYLE](#), [NOCLIP](#), [NODATA](#), [NOERASE](#), [NORMAL](#), [POSITION](#), [PSYM](#), [SUBTITLE](#), [SYMSIZE](#), [T3D](#), [THICK](#), [TICKLEN](#), [TITLE](#), [\[XYZ\]CHARSIZE](#), [\[XYZ\]GRIDSTYLE](#), [\[XYZ\]MARGIN](#), [\[XYZ\]MINOR](#), [\[XYZ\]RANGE](#), [\[XYZ\]STYLE](#), [\[XYZ\]THICK](#), [\[XYZ\]TICKFORMAT](#), [\[XYZ\]TICKINTERVAL](#), [\[XYZ\]TICKLAYOUT](#), [\[XYZ\]TICKLEN](#), [\[XYZ\]TICKNAME](#), [\[XYZ\]TICKS](#), [\[XYZ\]TICKUNITS](#), [\[XYZ\]TICKV](#), [\[XYZ\]TICK_GET](#), [\[XYZ\]TITLE](#), [ZVALUE](#).

Examples

The PLOT procedure has many keywords that allow you to create a vast variety of plots. Here are a few simple examples using the PLOT command.

```

; Create a simple dataset:
D = FINDGEN(100)

; Create a simple plot with the title "Simple Plot":
PLOT, D, TITLE = 'Simple Plot'

; Plot one argument versus another:
PLOT, SIN(D/3), COS(D/6)

; Create a polar plot:
PLOT, D, D, /POLAR, TITLE = 'Polar Plot'

; Use plotting symbols instead of connecting lines by including the
; PSYM keyword. Label the X and Y axes with XTITLE and YTITLE:
PLOT, SIN(D/10), PSYM=4, XTITLE='X Axis', YTITLE='Y Axis'

```

Version History

Introduced: Original

See Also

[IPLLOT](#), [OPLLOT](#), [PLOTS](#)

PLOT_3DBOX

The PLOT_3DBOX procedure plots a function of two variables (e.g., $Z=f(X, Y)$) inside a 3-D box. Optionally, the data can be projected onto the “walls” surrounding the plot area.

This routine is written in the IDL language. Its source code can be found in the file `plot_3dbox.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
PLOT_3DBOX, X, Y, Z [, GRIDSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, PSYM=integer{1 to 10}] [, /SOLID_WALLS] [, /XY_PLANE] [, XYSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, /XZ_PLANE] [, XZSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, /YZ_PLANE] [, YZSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, AX=degrees] [, AZ=degrees] [, ZAXIS={1 | 2 | 3 | 4}]
```

Graphics Keywords: Accepts all graphics keywords accepted by PLOT except for: FONT, PSYM, SYMSIZE, {XYZ}TICK_GET, and ZVALUE.

Arguments

X

A vector (i.e., a one-dimensional array) of X coordinates.

Y

A vector of Y coordinates.

Z

A vector of Z coordinates. $Z[i]$ is a function of $X[i]$ and $Y[i]$.

Keywords

GRIDSTYLE

Set this keyword to the linestyle index for the type of line to be used when drawing the gridlines. Linestyles are described in the following table:

Index	Linestyle
0	Solid
1	Dotted
2	Dashed
3	Dash Dot
4	Dash Dot Dot
5	Long Dashes

Table 3-93: IDL Linestyles

PSYM

Set this keyword to a plotting symbol index to be used in plotting the data. For more information, see “[PSYM](#)” on page 4340.

SOLID_WALLS

Set this keyword to cause the boundary “walls” of the plot to be filled with the color index specified by the `COLOR` keyword.

XY_PLANE

Set this keyword to plot the X and Y values on the $Z=0$ axis plane.

XYSTYLE

Set this keyword to the linestyle used to draw the XY plane plot. See the table above for a list of linestyles.

XZ_PLANE

Set this keyword to plot the Y and Z values on the $Y=\text{MAX}(Y)$ axis plane.

XZSTYLE

Set this keyword to the linestyle used to draw the XZ plane plot. See the table above for a list of linestyles.

YZ_PLANE

Set this keyword to plot the Y and Z values on the X=MAX(X) axis plane.

YZSTYLE

Set this keyword to the linestyle used to draw the YZ plane plot. See the table above for a list of linestyles.

SURFACE Keywords

In addition to the keywords described above, the AX, AZ, and ZAXIS keywords to the SURFACE procedure are accepted by PLOT_3DBOX. See “SURFACE” on page 2058.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above. [BACKGROUND](#), [CHARSIZE](#), [CHARTHICK](#), [CLIP](#), [COLOR](#), [DATA](#), [DEVICE](#), [LINestyle](#), [NOCLIP](#), [NOERASE](#), [NORMAL](#), [POSITION](#), [SUBTITLE](#), [T3D](#), [THICK](#), [TICKLEN](#), [TITLE](#), [\[XYZ\]CHARSIZE](#), [\[XYZ\]GRIDSTYLE](#), [\[XYZ\]MARGIN](#), [\[XYZ\]MINOR](#), [\[XYZ\]RANGE](#), [\[XYZ\]STYLE](#), [\[XYZ\]THICK](#), [\[XYZ\]TICKFORMAT](#), [\[XYZ\]TICKLEN](#), [\[XYZ\]TICKNAME](#), [\[XYZ\]TICKS](#), [\[XYZ\]TICKV](#), [\[XYZ\]TITLE](#).

Examples

```
; Create some data to be plotted:
X = REPLICATE(5., 10.)
X1 = COS(FINDGEN(36)*10.*!DTOR)*2.+5.
X = [X, X1, X]
Y = FINDGEN(56)
Z = REPLICATE(5., 10)
Z1 = SIN(FINDGEN(36)*10.*!DTOR)*2.+5.
Z = [Z, Z1, Z]
```

```
; Create the box plot with data projected on all of the walls. The
; PSYM value of -4 plots the data as diamonds connected by lines:
PLOT_3DBOX, X, Y, Z, /XY_PLANE, /YZ_PLANE, /XZ_PLANE, $
  /SOLID_WALLS, GRIDSTYLE=1, XYSTYLE=3, XZSTYLE=4, $
  YZSTYLE=5, AZ=40, TITLE='Example Plot Box', $
  XTITLE='X Coordinate', YTITLE='Y Coodinate', $
  ZTITLE='Z Coordinate', SUBTITLE='Sub Title', $
  /YSTYLE, ZRANGE=[0,10], XRANGE=[0,10], $
  PSYM=-4, CHARSIZE=1.6
```

Version History

Introduced: Pre 4.0

See Also

[IPLOT](#), [PLOTS](#), [SURFACE](#)

PLOT_FIELD

The PLOT_FIELD procedure plots a 2-D field. N random points are picked, and from each point a path is traced along the field. The length of the path is proportional to the field vector magnitude.

This routine is written in the IDL language. Its source code can be found in the file `plot_field.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
PLOT_FIELD, U, V [, ASPECT=ratio] [, LENGTH=value] [, N=num_arrows]  
[, TITLE=string]
```

Arguments

U

A 2-D array giving the field vector at each point in the U(X) direction.

V

A 2-D array giving the field vector at each point in the V(Y) direction.

Keywords

ASPECT

Set this keyword to the aspect ratio of the plot (i.e., the ratio of the X size to Y size). The default is 1.0.

LENGTH

Set this keyword to the length of the longest field vector expressed as a fraction of the plotting area. The default is 0.1.

N

Set this keyword to the number of arrows to draw. The default is 200.

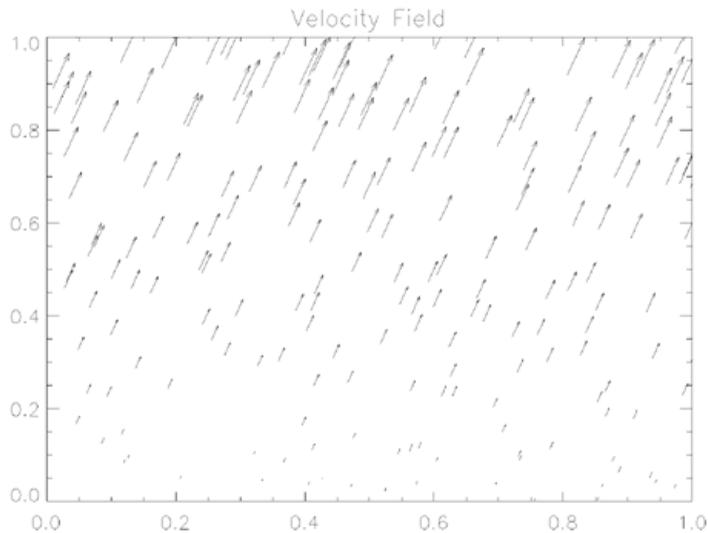
TITLE

Set this keyword to the title of plot. The default is “Velocity Field”.

Examples

```
; Create array X:  
X = FINDGEN(20, 20)  
  
; Create array Y:  
Y = FINDGEN(20, 20)*3  
  
; Plot X vs. Y:  
PLOT_FIELD, X, Y
```

The above commands produce the following plot:



Version History

Introduced: Original

See Also

[FLOW3](#), [VEL](#), [VELOVECT](#)

PLOTERR

The PLOTERR procedure plots individual data points with error bars.

This routine is written in the IDL language. Its source code can be found in the file `ploterr.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
PLOTERR, [ X,] Y, Err [, TYPE={1 | 2 | 3 | 4}] [, PSYM=integer{1 to 10}]
```

Arguments

X

An optional array of X values. The procedure checks the number of arguments passed to decide if X was passed. If X is not passed, `INDGEN(Y)` is assumed for X values.

Y

The array of Y values. Y cannot be of type string.

Err

The array of error-bar values.

Keywords

TYPE

The type of plot to be produced. The possible types are:

- 1 = X Linear - Y Linear (default)
- 2 = X Linear - Y Log
- 3 = X Log - Y Linear
- 4 = X Log - Y Log

PSYM

The plotting symbol to use. The default is +7.

Version History

Introduced: Original

See Also

[ERRPLOT](#), [IPLLOT](#), [OPLOTERR](#), [PLOT](#)

PLOTS

The PLOTS procedure plots vectors or points on the current graphics device in either two or three dimensions. The coordinates can be given in data, device, or normalized form using the DATA (the default), DEVICE, or NORMAL keywords.

The COLOR keyword can be set to a scalar or vector value. If it is set to a vector value, the line segment connecting (X_i, Y_i) to (X_{i+1}, Y_{i+1}) is drawn with a color index of $COLOR_{i+1}$. In this case, COLOR must have the same number of elements as X and Y.

Syntax

PLOTS, X [, Y [, Z]] [, /CONTINUE]

Graphics Keywords: [, CLIP=[X_0, Y_0, X_1, Y_1]] [, COLOR=*value*] [, /DATA | , /DEVICE | , /NORMAL] [, LINSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, /NOCLIP] [, PSYM=*integer*{0 to 10}] [, SYMSIZE=*value*] [, /T3D] [, THICK=*value*] [, Z=*value*]

Arguments

X

A vector or scalar argument providing the X components of the points to be connected. If only one argument is specified, X must be an array of either two or three vectors (i.e., (2, *) or (3, *)). In this special case, $x[0, *]$ are taken as the X values, $x[1, *]$ are taken as the Y values, and $x[2, *]$ are taken as the Z values.

Y

An optional argument providing the Y coordinate(s) of the points to be connected.

Z

An optional argument providing the Z coordinates of the points to be connected. If Z is not provided, X and Y are used to draw lines in two dimensions.

Z has no effect if the keyword T3D is not specified and the system variable !P.T3D=0.

Keywords

CONTINUE

Set this keyword to continue drawing a line from the last point of the most recent call to PLOTS.

For example:

```

; Position at (0,0):
PLOTS, 0, 0

; Draws vector from (0,0) to (1,1):
PLOTS, 1, 1, /CONTINUE

; Draws two vectors from (1,1) to (2,2) to (3,3):
PLOTS, [2,3], [2,3], /CONTINUE

```

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above. [CLIP](#), [COLOR](#), [DATA](#), [DEVICE](#), [LINestyle](#), [NOCLIP](#), [NORMAL](#), [PSYM](#), [SYMsize](#), [T3D](#), [THICK](#), [Z](#).

Examples

```

; Draw a line from (100, 200) to (600, 700), in device coordinates,
; using color index 12:
PLOTS, [100,600], [200,700], COLOR=12, /DEVICE

; Draw a polyline where the line color is proportional to the
; ordinate that ends each line segment.
; First create datasets X and Y:
X = SIN(FINDGEN(100)) & Y = COS(FINDGEN(100))

; Now plot X and Y in normalized coordinates with colors as
; described above:
PLOTS, X, Y, COLOR = BYTSCL(Y, TOP=!D.N COLORS-1), /NORMAL

; Load a good colortable to better show the result:
LOADCT, 13

; Draw 3-D vectors over an established SURFACE plot.
; The SAVE keyword tells IDL to save the 3-D transformation
; established by SURFACE.
SURFACE, DIST(5), /SAVE

```

```
; Draw a line between (0,0,0) and (3,3,3). The T3D keyword makes
; PLOTS use the previously established 3-D transformation:
PLOTS, [0,3], [0,3], [0,3], /T3D

; Draw a line between (3,0,0) and (3,3,3):
PLOTS, [3,3], [0,3], [0,3], /T3D

; Draw a line between (0,3,0) and (3,3,3):
PLOTS, [0,3], [3,3], [0,3], /T3D
```

Version History

Introduced: Original

See Also

[ANNOTATE](#), [IPLOT](#), [XYOUTS](#)

PNT_LINE

The PNT_LINE function computes the perpendicular distance between a point $P0$ and a line between points $L0$ and $L1$. This function is limited by the machine accuracy of single precision floating point.

This routine is written in the IDL language. Its source code can be found in the file `pnt_line.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$Result = \text{PNT_LINE}(P0, L0, L1 [, PI] [, /INTERVAL])$

Return Value

Returns the perpendicular distance.

Arguments

P0

The location of the point. $P0$ may have 2 to n elements, for n dimensions.

L0

One end-point of the line. $L0$ must have same number of elements as $P0$.

L1

The other end-point of the line. $L1$ must have the same number of elements as $L0$.

PI

A named variable that will contain the location of the point on the line between $L0$ and $L1$ that is closest to $P0$. PI is not necessarily in the interval $(L0, L1)$.

Keywords

INTERVAL

If set, and if the point on the line between $L0$ and $L1$ that is closest to $P0$ is not within the interval $(L0, L1)$, PNT_LINE will return the distance from $P0$ to the closer of the two endpoints $L0$ and $L1$.

Examples

To print the distance between the point (2,3) and the line from (-3,3) to (5,12), and also the location of the point on the line closest to (2,3), enter the following command:

```
PRINT, PNT_LINE([2,3], [-3,3], [5,12], P1), P1
```

IDL prints:

```
3.73705      -0.793104      5.48276
```

Version History

Introduced: Pre 4.0

See Also

[CIR_3PNT](#), [SPH_4PNT](#)

POINT_LUN

The POINT_LUN procedure sets or obtains the current position of the file pointer for the specified file.

Note

POINT_LUN cannot be used with files opened with the RAWIO keyword to the OPEN routines. Depending upon the device in question, the IOCTL function might be used instead for files of this type.

Use Of POINT_LUN On Compressed Files

In general, it is not possible to arbitrarily move the file pointer within a compressed file (files opened with the COMPRESS keyword to OPEN) because the file compression code needs to maintain a compression state for the file that includes all the data that has already been passed in the stream. This limitation results in the following constraints on the use of POINT_LUN with compressed files:

- POINT_LUN is not allowed on compressed files open for output, except to positions beyond the current file position. The compression code emulates such motion by outputting enough zero bytes to move the pointer to the new position.
- POINT_LUN is allowed to arbitrary positions on compressed files opened for input. However, this feature is emulated by positioning the file to the beginning of the file and then reading and discarding enough data to move the file pointer to the desired position. This can be extremely slow.

For these reasons, use of POINT_LUN on compressed files, although possible under some circumstances, is best avoided.

Syntax

POINT_LUN, *Unit*, *Position*

Arguments

Unit

The file unit for the file in question. If *Unit* is positive, POINT_LUN sets the file position to the position given by *Position*. If negative, POINT_LUN gets the current

file position and assigns it to the variable given by *Position*. Note that `POINT_LUN` cannot be used with the 3 standard file units (0, -1, and -2).

Position

If *Unit* is positive, *Position* gives the byte offset into the file at which the file pointer should be set. For example, to rewind the file to the beginning, specify 0.

If *Unit* is negative, *Position* must be a named variable into which the current file position will be stored. The returned type will be a longword signed integer if the position is small enough to fit, and a 64-bit longword integer otherwise.

Keywords

None.

Examples

To move the file pointer 2048 bytes into the file associated with file unit number 1, enter:

```
POINT_LUN, 1, 2048
```

To return the file pointer for file unit number 2, enter:

```
POINT_LUN, -2, pos
```

Version History

Introduced: Original

See Also

[GET_LUN](#), [OPEN](#), [TRUNCATE_LUN](#)

POLAR_CONTOUR

The POLAR_CONTOUR procedure draws a contour plot from data in polar coordinates. Data can be regularly- or irregularly-gridded. All of the keyword options supported by CONTOUR are available to POLAR_CONTOUR.

This routine is written in the IDL language. Its source code can be found in the file `polar_contour.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
POLAR_CONTOUR, Z, Theta, R [, C_ANNOTATION=vector_of_strings]
  [, C_CHARSIZE=value] [, C_CHARTHICK=integer] [, C_COLORS=vector]
  [, C_LINestyle=vector] [, /FILL | , CELL_FILL
  [, C_ORIENTATION=degrees] [, C_SPACING=value]] [, C_THICK=vector]
  [, /CLOSED] [, /IRREGULAR] [, LEVELS=vector / NLEVELS=integer{1 to
  29}] [, MAX_VALUE=value] [, MIN_VALUE=value] [, /OVERPLOT]
  [, /PATH_DATA_COORDS | ,TRIANGULATION=variable] [, /XLOG]
  [, /YLOG] [, /ZAXIS] [, SHOW_TRIANGULATION=color_index]
```

Arguments

Z

The data values to be contoured. If the data is regularly gridded, *Z* must have the dimensions (N_ELEMENTS(*Theta*), N_ELEMENTS(*R*)). Note that the ordering of the elements in the array *Z* is opposite that used by the POLAR_SURFACE routine.

Theta

A vector of angles in radians. For regularly-gridded data, *Theta* must have the same number of elements as the first dimension of *Z*. For a scattered grid, *Theta* must have the same number of elements as *Z*.

R

A vector of radius values. For regularly-gridded data, *R* must have the same number of elements as the second dimension of *Z*. For a scattered grid, *R* must have the same number of elements as *Z*.

Keywords

POLAR_CONTOUR accepts all of the keywords accepted by the CONTOUR routine except C_LABELS, DOWNHILL, FOLLOW, PATH_FILENAME, PATH_INFO, and PATH_XY. See “CONTOUR” on page 315. In addition, there is one unique keyword:

SHOW_TRIANGULATION

Set this keyword to a color index to be used in overplotting the triangulation between datapoints.

Examples

This example uses POLAR_CONTOUR with regularly-gridded data:

```
;Handle TrueColor displays:
DEVICE, DECOMPOSED=0

;Load color table
TEK_COLOR

nr = 12 ; number of radii
nt = 18 ; number of Thetas

; Create a vector of radii:
r = FINDGEN(nr)/(nr-1)

; Create a vector of Thetas:
theta = 2*!PI * FINDGEN(nt)/(nt-1)

; Create some data values to be contoured:
z = COS(theta*3) # (r-0.5)^2

; Create the polar contour plot:
POLAR_CONTOUR, z, theta, r, /FILL, c_color=[2, 3, 4, 5]
```

Version History

Introduced: 4.0

See Also

[CONTOUR](#)

POLAR_SURFACE

The POLAR_SURFACE function interpolates a surface from polar coordinates (R , Θ , Z) to rectangular coordinates (X , Y , Z).

This routine is written in the IDL language. Its source code can be found in the file `polar_surface.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = POLAR_SURFACE( Z, R, Theta [, /GRID] [, SPACING=[xspacing,
    yspacing]] [, BOUNDS=[x0, y0, x1, y1]] [, /QUINTIC] [, MISSING=value] )
```

Return Value

The function returns a two-dimensional array of the same type as Z .

Arguments

Z

An array containing the surface value at each point. If the data are regularly gridded in R and Θ , Z is a two dimensional array, where $Z_{i,j}$ has a radius of R_i and an azimuth of Θ_{tj} . If the data are irregularly-gridded, R_i and Θ_{tj} contain the radius and azimuth of each Z_i . Note that the ordering of the elements in the array Z is opposite that used by the POLAR_CONTOUR routine.

R

The radius. If the data are regularly gridded in R and Θ , $Z_{i,j}$ has a radius of R_i . If the data are irregularly-gridded, R must have the same number of elements as Z , and contains the radius of each point.

Theta

The azimuth, in radians. If the data are regularly gridded in R and Θ , $Z_{i,j}$ has an azimuth of Θ_{tj} . If the data are irregularly-gridded, Θ must have the same number of elements as Z , and contains the azimuth of each point.

Keywords

GRID

Set this keyword to indicate that Z is regularly gridded in R and Θ .

SPACING

A two element vector containing the desired grid spacing of the resulting array in x and y . If omitted, the grid will be approximately 51 by 51.

BOUNDS

A four element vector, $[x_0, y_0, x_1, y_1]$, containing the limits of the xy grid of the resulting array. If omitted, the extent of input data sets the limits of the grid.

QUINTIC

Set this keyword to use quintic interpolation, which is slower but smoother than the default linear interpolation.

MISSING

Use this keyword to specify a value to use for areas within the grid but not within the convex hull of the data points. The default is 0.0.

Examples

```
; Define radius and Theta:
R = FINDGEN(50) / 50.0
THETA = FINDGEN(50) * (2 * !PI / 50.0)

; Make a function (tilted circle):
Z = R # SIN(THETA)

; Show it:
SURFACE, POLAR_SURFACE(Z, R, THETA, /GRID)
```

Version History

Introduced: Pre 4.0

See Also

[POLAR](#) keyword to [PLOT](#)

POLY

The POLY function evaluates a polynomial function of a variable.

This routine is written in the IDL language. Its source code can be found in the file `poly.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = POLY(*X*, *C*)

Return Value

The result is equal to:

$$C_0 + C_1x + C_2x^2 + \dots$$

Arguments

X

The variable. This value can be a scalar, vector or array.

C

The vector of polynomial coefficients. The degree of the polynomial is `N_ELEMENTS(C) - 1`.

Keywords

None.

Version History

Introduced: Original

See Also

[FZ_ROOTS](#)

POLY_2D

The `POLY_2D` function performs polynomial warping of images. This function performs a geometrical transformation in which the resulting array is defined by:

$$g[x, y] = f[x', y'] = f[a[x, y], b[x, y]]$$

where $g[x, y]$ represents the pixel in the output image at coordinate (x, y) , and $f[x', y']$ is the pixel at (x', y') in the input image that is used to derive $g[x, y]$. The functions $a(x, y)$ and $b(x, y)$ are polynomials in x and y of degree N , whose coefficients are given by P and Q , and specify the spatial transformation:

$$x' = a(x, y) = \sum_{i=0}^N \sum_{j=0}^N P_{i,j} x^i y^j$$

$$y' = b(x, y) = \sum_{i=0}^N \sum_{j=0}^N Q_{i,j} x^i y^j$$

Either the nearest neighbor or bilinear interpolation methods can be selected.

Syntax

`Result = POLY_2D(Array, P, Q [, Interp [, Dim_x, Dim_y]] [, CUBIC={-1 to 0}] [, MISSING=value])`

Arguments

Array

A two-dimensional array of any basic type except string. The result has the same type as `Array`.

P and Q

P and Q are arrays containing the polynomial coefficients. Each array must contain $(N+1)^2$ elements (where N is the degree of the polynomial). For example, for a linear transformation, P and Q contain four elements and can be a 2×2 array or a 4-element vector. $P_{i,j}$ contains the coefficient used to determine x' , and is the weight of the term $x^i y^j$. The `POLYWARP` procedure can be used to fit (x', y') as a function of (x, y) and determines the coefficient arrays P and Q .

Interp

Set this argument to 1 to perform bilinear interpolation. Set this argument to 2 to perform cubic convolution interpolation (as described under the CUBIC keyword, below). Otherwise, the nearest neighbor method is used. For the linear case, ($N=1$), bilinear interpolation requires approximately twice as much time as does the nearest neighbor method.

Dim_x

If present, *Dim_x* specifies the number of columns in the output. If omitted, the output has the same number of columns as *Array*.

Dim_y

If present, *Dim_y* specifies the number of rows in the output. If omitted, the output has the same number of rows as *Array*.

Keywords

CUBIC

Set this keyword to a value between -1 and 0 to use the cubic convolution interpolation method with the specified value as the interpolation parameter. Setting this keyword equal to a value greater than zero specifies a value of -1 for the interpolation parameter. Park and Schowengerdt (see reference below) suggest that a value of -0.5 significantly improves the reconstruction properties of this algorithm. Note that cubic convolution interpolation works only with one- and two-dimensional arrays.

Cubic convolution is an interpolation method that closely approximates the theoretically optimum sinc interpolation function using cubic polynomials. According to sampling theory, details of which are beyond the scope of this document, if the original signal, f , is a band-limited signal, with no frequency component larger than ω_0 , and f is sampled with spacing less than or equal to $1/2\omega_0$, then f can be reconstructed by convolving with a sinc function: $\text{sinc}(x) = \sin(\pi x) / (\pi x)$.

In the one-dimensional case, four neighboring points are used, while in the two-dimensional case 16 points are used. Note that cubic convolution interpolation is significantly slower than bilinear interpolation.

For further details see:

Rifman, S.S. and McKinnon, D.M., “Evaluation of Digital Correction Techniques for ERTS Images; Final Report”, Report 20634-6003-TU-00, TRW Systems, Redondo Beach, CA, July 1974.

S. Park and R. Schowengerdt, 1983 “Image Reconstruction by Parametric Cubic Convolution”, *Computer Vision, Graphics & Image Processing* 23, 256.

MISSING

Specifies the output value for points whose x' , y' is outside the bounds of *Array*. If MISSING is not specified, the resulting output value is extrapolated from the nearest pixel of *Array*.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Some simple linear (degree one) transformations are:

$P_{0,0}$	$P_{1,0}$	$P_{0,1}$	$P_{1,1}$	$Q_{0,0}$	$Q_{1,0}$	$Q_{0,1}$	$Q_{1,1}$	Effect
0	0	1	0	0	1	0	0	Identity
0	0	0.5	0	0	1	0	0	Stretch X by a factor of 2
0	0	1	0	0	2.0	0	0	Shrink Y by a factor of 2
z	0	1	0	0	1	0	0	Shift left by z pixels
0	1	0	0	0	0	1	0	Transpose

Table 3-94: Simple Transformations for Use with POLY_2D

POLY_2D is often used in conjunction with the POLYWARP procedure to warp images.

```
; Create and display a simple image:
A = BYTSCL(SIN(DIST(250)), TOP=!D.TABLE_SIZE) & TV, A

; Set up the arrays of original points to be warped:
XO = [61, 62, 143, 133]
YO = [89, 34, 38, 105]

; Set up the arrays of points to be fit:
XI = [24, 35, 102, 92]
YI = [81, 24, 25, 92]

; Use POLYWARP to generate the P and Q inputs to POLY_2D:
POLYWARP, XI, YI, XO, YO, 1, P, Q

; Perform an image warping based on P and Q:
B = POLY_2D(A, P, Q)

; Display the new image:
TV, B, 250, 250
```

Images can also be warped over irregularly gridded control points using the `WARP_TRI` procedure.

Version History

Introduced: Original

See Also

[POLYWARP](#)

POLY_AREA

The POLY_AREA function returns the area of a polygon given the coordinates of its vertices.

It is assumed that the polygon has n vertices with n sides and the edges connect the vertices in the order:

$$[(x_1,y_1), (x_2,y_2), \dots, (x_n,y_n), (x_1,y_1)]$$

such that the last vertex is connected to the first vertex.

This routine is written in the IDL language. Its source code can be found in the file `poly_area.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = POLY_AREA(X, Y [, /DOUBLE] [, /SIGNED])

Return Value

If either of the input arguments is double-precision or if the DOUBLE keyword is set, the result is a double-precision value, otherwise, the result is single-precision.

Arguments

X

An n -element vector of X coordinate locations for the vertices.

Y

An n -element vector of Y coordinate locations for the vertices.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision result. Explicitly set DOUBLE=0 to use single-precision for computations and to return a single-precision result. By default, if either of the arguments to POLY_AREA is double-precision, computations are done in double-precision; if both arguments are single-precision, computations are done in single-precision.

SIGNED

If set, returns a signed area. Polygons with edges traversed in counterclockwise order have a positive area; polygons traversed in the clockwise order have a negative area.

Version History

Introduced: Original

See Also

[DEFROI](#), [POLYFILLV](#)

POLY_FIT

The POLY_FIT function performs a least-square polynomial fit with optional weighting and returns a vector of coefficients.

The POLY_FIT routine uses matrix inversion to determine the coefficients. A different version of this routine, SVDFIT, uses singular value decomposition (SVD). The SVD technique is more flexible and robust, but may be slower.

This routine is written in the IDL language. Its source code can be found in the file `poly_fit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = POLY_FIT( X, Y, Degree [, CHISQ=variable] [, COVAR=variable]
  [, /DOUBLE] [, MEASURE_ERRORS=vector] [, SIGMA=variable]
  [, STATUS=variable] [, YBAND=variable] [, YERROR=variable]
  [, YFIT=variable] )
```

Return Value

POLY_FIT returns a vector of coefficients of length $Degree+1$. If the `DOUBLE` keyword is set, or if `X` or `Y` are double precision, then the result will be double precision, otherwise the result will be single precision.

Arguments

X

An n -element vector of independent variables.

Y

A vector of dependent variables, the same length as `X`.

Degree

The degree of the polynomial to fit.

Yfit, Yband, Sigma, Corrm

The Yfit, Yband, Sigma, and Corrm arguments are obsolete, and have been replaced by the YFIT, YBAND, YERROR, and COVAR keywords, respectively. Code using these

arguments will continue to work as before, but new code should use the keywords instead.

Keywords

CHISQ

Set this keyword to a named variable that will contain the value of the unreduced chi-square goodness-of-fit statistic.

COVAR

Set this keyword to a named variable that will contain the Covariance matrix of the coefficients.

Note

The COVAR matrix depends only upon the independent variable X and (optionally) the MEASURE_ERRORS. The values do not depend upon Y . See section 15.4 of *Numerical Recipes in C* (Second Edition) for details.

DOUBLE

Set this keyword to force computations to be done in double-precision arithmetic. All computations are performed using double-precision arithmetic.

MEASURE_ERRORS

Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y .

Note

For Gaussian errors (e.g., instrumental uncertainties), MEASURE_ERRORS should be set to the standard deviations of each point in Y . For Poisson or statistical weighting, MEASURE_ERRORS should be set to $\text{SQRT}(Y)$.

SIGMA

Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters.

Note

If MEASURE_ERRORS is omitted, then you are assuming that a polynomial is the correct model for your data, and therefore, no independent goodness-of-fit test is

possible. In this case, the values returned in SIGMA are multiplied by $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number of coefficients. See section 15.2 of *Numerical Recipes in C* (Second Edition) for details.

STATUS

Set this keyword to a named variable to receive the status of the operation. Possible status values are:

- 0 = Successful completion.
- 1 = Singular array (which indicates that the inversion is invalid). *Result* is NaN.
- 2 = Warning that a small pivot element was used and that significant accuracy was probably lost.
- 3 = Undefined (NaN) error estimate was encountered.

Note

If STATUS is not specified, any error messages will be output to the screen.

Tip

Status values of 2 or 3 can often be resolved by setting the DOUBLE keyword.

YBAND

Set this keyword to a named variable that will contain the 1 standard deviation error estimate for each point.

YERROR

Set this keyword to a named variable that will contain the standard error between YFIT and Y.

YFIT

Set this keyword to a named variable that will contain the vector of calculated Y values. These values have an error of + or - YBAND.

Examples

In this example, we use X and Y data corresponding to the known polynomial $f(x) = 0.25 - x + x^2$. Using POLY_FIT to compute a second degree polynomial fit returns the exact coefficients (to within machine accuracy).

```
; Define an 11-element vector of independent variable data:
X = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

; Define an 11-element vector of dependent variable data:
Y = [0.25, 0.16, 0.09, 0.04, 0.01, 0.00, 0.01, 0.04, 0.09, $
     0.16, 0.25]

; Define a vector of measurement errors:
measure_errors = REPLICATE(0.01, 11)

; Compute the second degree polynomial fit to the data:
result = POLY_FIT(X, Y, 2, MEASURE_ERRORS=measure_errors, $
                 SIGMA=sigma)

; Print the coefficients:
PRINT, 'Coefficients: ', result
PRINT, 'Standard errors: ', sigma
```

IDL prints:

```
Coefficients:      0.250000      -1.00000      1.00000
Standard errors:   0.00761853      0.0354459      0.0341395
```

Version History

Introduced: Original

See Also

[COMFIT](#), [CURVEFIT](#), [GAUSSFIT](#), [LINFIT](#), [REGRESS](#), [SFIT](#), [SVDFIT](#)

POLYFILL

The POLYFILL procedure fills the interior of a region of the display enclosed by an arbitrary two or three-dimensional polygon. The available filling methods are: solid fill, parallel lines, or a pattern contained in an array. Not all methods are available on every hardware output device. See “Fill Methods” below.

Note

POLYFILL uses the current graphics device’s own polygon filling methodology when possible. For some devices, polygon filling is designed to avoid filling a given pixel more than once when neighboring polygons (that is, polygons with shared edges) are drawn. If the resulting pixel fill from POLYFILL is unsatisfactory, consider using [DRAW_ROI](#) instead.

The polygon is defined by a list of connected vertices stored in X, Y, and Z. The coordinates can be given in data, device, or normalized form using the DATA, DEVICE, or NORMAL keywords.

Fill Methods

Line-fill method: Filling using parallel lines is device-independent and works on all devices that can draw lines. Crosshatching can be simulated by performing multiple fills with different orientations. The spacing, linestyle, orientation, and thickness of the filling lines can be specified using the corresponding keyword parameters. The LINE_FILL keyword selects this filling style, but is not required if either the ORIENTATION or SPACING parameters are present.

Solid fill method: By default, POLYFILL fills the polygon with a solid color. For devices that do not directly support filling with a solid color, the solid fill is automatically emulated using the line-fill method.

Patterned fill: Some output devices support filling with a pattern. For these devices, the fill pattern array can be explicitly specified with the PATTERN keyword. Refer to the description of that keyword for a list of devices that support patterned fill.

Syntax

```
POLYFILL, X [, Y [, Z]] [, IMAGE_COORD=array] [, /IMAGE_INTERP]
      [, /LINE_FILL] [, PATTERN=array] [, SPACING=centimeters]
      [, TRANSPARENT=value]
```

Graphics Keywords: [, CLIP=[X₀, Y₀, X₁, Y₁]] [, COLOR=value] [, /DATA | ,
/DEVICE | , /NORMAL] [, LINSTYLE={0 | 1 | 2 | 3 | 4 | 5}] [, /NOCLIP]

[, ORIENTATION=*ccw_degrees_from_horiz*] [, /T3D] [, THICK=*value*]
 [, Z=*value*]

Arguments

X

A vector argument providing the X coordinates of the points to be connected. The vector must contain at least three elements. If only one argument is specified, X must be an array of either two or three vectors (i.e., (2 , *) or (3 , *)). In this special case, the vector `X[0 , *]` specifies the X values, `X[1 , *]` specifies Y, and `X[2 , *]` contain the Z values.

Y

A vector argument providing the Y coordinates of the points to be connected. Y must contain at least three elements.

Z

An optional vector argument providing the Z coordinates of the points to be connected. If Z is not provided, X and Y are used to draw lines in two dimensions. Z must contain at least three elements. Z has no effect if the keyword T3D is not specified and the system variable !P.T3D= 0.

Keywords

IMAGE_COORD

This keyword is only available when creating output in the Z-Buffer.

A $2 \times n$ array containing the fill pattern array subscripts of each of the n polygon vertices. Use this keyword in conjunction with the PATTERN keyword to warp images over 2-D and 3-D polygons.

IMAGE_INTERP

This keyword is only available when creating output in the Z-Buffer.

Specifies the method of sampling the PATTERN array when the IMAGE_COORD keyword is present. The default method is to use nearest-neighbor sampling. Bilinear interpolation sampling is performed if IMAGE_INTERP is set.

LINE_FILL

Set this keyword to indicate that polygons are to be filled with parallel lines, rather than using solid or patterned filling methods. When using the line-drawing method of filling, the thickness, linestyle, orientation, and spacing of the lines may be specified with keywords.

PATTERN

A rectangular array of pixels giving the fill pattern. If this keyword parameter is omitted, POLYFILL fills the area with a solid color. The pattern array may be of any size; if it is smaller than the filled area the pattern array is cyclically repeated.

Note

This keyword is supported for the following devices:

METAFILE, PRINTER, PS, WIN, X, and Z.

For the PostScript device, fill patterns are only supported with language level 2. Use the LANGUAGE_LEVEL keyword to DEVICE to set the PostScript language level to 2 if filled patterns are to be used. On Windows98, the pattern size is limited to 8x8.

For example, to fill the current plot window with a grid of dots, enter the following commands:

```
; Define pattern array as 10 by 10:
PAT = BYTARR(10,10)

; Set center pixel to bright:
PAT[5,5] = 255

; Fill the rectangle defined by the four corners of the window with
; the pattern:
POLYFILL, !X.WINDOW([0,1,1,0]), $
          !Y.WINDOW([0,0,1,1]), /NORM, PAT = PAT
```

SPACING

The spacing, in centimeters, between the parallel lines used to fill polygons.

TRANSPARENT

This keyword is only available when creating output in the Z-Buffer.

Specifies the minimum pixel value to draw in conjunction with the PATTERN and IMAGE_COORD keywords. Pixels less than this value are not drawn and the Z-buffer is not updated.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above. CLIP, COLOR, DATA, DEVICE, LINSTYLE, NOCLIP, NORMAL, ORIENTATION, T3D, THICK, Z.

Z-Buffer-Specific Keywords

Certain keyword parameters are only active when the Z-buffer is the currently selected graphics device: IMAGE_COORD, IMAGE_INTERP, TRANSPARENT and COLOR. These parameters allow images to be warped over 2-D or 3-D polygons, and the output of shaded polygons. See [“The Z-Buffer Device”](#) in Appendix A.

For shaded polygons, the COLOR keyword can specify an array that contains the color index at each vertex. Color indices are linearly interpolated between vertices. If COLOR contains a scalar, the entire polygon is drawn with the given color index, just as with the other graphics output devices.

Images can be warped over polygons by passing in the image with the PATTERN parameter, and a $(2, n)$ array containing the image space coordinates that correspond to each of the N vertices with the IMAGE_COORD keyword.

The IMAGE_INTERP keyword indicates that bilinear interpolation is to be used, rather than the default nearest-neighbor sampling. Pixels less than the value of TRANSPARENT are not drawn, simulating transparency.

Examples

Fill a rectangular polygon that has the vertices (30,30), (100, 30), (100, 100), and (30, 100) in device coordinates:

```
; Create the vectors of X and Y values:
X = [30, 100, 100, 30] & Y = [30, 30, 100, 100]

; Fill the polygon with color index 175:
POLYFILL, X, Y, COLOR = 175, /DEVICE
```

Version History

Introduced: Original

See Also

[POLYFILLV](#)

POLYFILLV

The POLYFILLV function returns a vector containing the subscripts of the array elements contained inside a polygon defined by vectors.

The X and Y parameters are vectors that contain the subscripts of the vertices that define the polygon in the coordinate system of the two-dimensional S_x by S_y array. The S_x and S_y parameters define the number of columns and rows in the array enclosing the polygon. At least three points must be specified, and all points should lie within the limits: $0 \leq X_i < S_x$ and $0 \leq Y_i < S_y$ for all i .

As with the POLYFILL procedure, the polygon is defined by connecting each point with its successor and the last point with the first. This function is useful for defining, analyzing, and displaying regions of interest within a two-dimensional array.

The scan line coordinate system defined by Rogers in *Procedural Elements for Computer Graphics*, McGraw-Hill, 1985, page 71, is used. In this system, the scan lines are considered to pass through the center of each row of pixels. Pixels are activated if the center of the pixel is to the right of the intersection of the scan line and the polygon edge within the interval.

Syntax

$Result = \text{POLYFILLV}(X, Y, S_x, S_y [, Run_Length])$

Return Value

Returns a vector containing the one-dimensional subscripts of the array elements contained inside a polygon defined by vectors X and Y . If no points are contained within the polygon, a -1 is returned and an informational message is printed.

Arguments

X

A vector containing the X subscripts of the vertices that define the polygon.

Y

A vector containing the Y subscripts of the vertices that define the polygon.

S_x

The number of columns in the array surrounding the polygon.

S_y

The number of rows in the array surrounding the polygon.

Run_Length

Set this optional parameter to a nonzero value to make POLYFILLV return a vector of run lengths, rather than subscripts. For large polygons, a considerable savings in space results. When run-length encoded, each element with an even subscript result contains the length of the run, and the following element contains the starting index of the run.

Examples

To determine the mean and standard deviation of the elements within a triangular region defined by the vertices at pixel coordinates (100, 100), (200, 300), and (300, 100), inside a 512 x 512 array called DATA, enter the commands:

```
; Get the subscripts of the elements inside the triangle:  
P = DATA[POLYFILLV([100,200,300], [100,300,100], 512, 512)]  
  
; Use the STDEV function to obtain the mean and standard deviation  
; of the selected elements:  
STD = STDEV(P,MEAN)
```

Version History

Introduced: Original

See Also

[POLYFILL](#)

POLYSHADE

The POLYSHADE function creates a shaded-surface representation of one or more solids described by a set of polygons. This function accepts, as arguments, an array of three-dimensional vertices and a list of the indices of the vertices that describe each polygon.

Shading values are determined from one of three sources: a light source model, a user-specified array containing vertex shade values, or a user-specified array containing polygon shade values.

The shaded surface is constructed using the scan line algorithm. The default shading model is a combination of diffuse reflection and depth cueing. With this shading model, polygons are shaded using either constant shading, in which each polygon is given a constant intensity, or with Gouraud shading where the intensity is computed at each vertex and then interpolated over the polygon. Use the SET_SHADING procedure to control the direction of the light source and other shading parameters.

User-specified shading arrays allow “4-dimensional” displays that consist of a surface defined by a set of polygons, shaded with values from another variable.

Syntax

Result = POLYSHADE(*Vertices*, *Polygons*)

or

Result = POLYSHADE(*X*, *Y*, *Z*, *Polygons*)

Keywords: [, /DATA | , /NORMAL] [, POLY_SHADES=*array*] [, SHADES=*array*] [, /T3D] [, TOP=*value*] [, XSIZE=*columns*] [, YSIZE=*rows*]

Return Value

Returns a shaded-surface representation. Output is a two-dimensional byte array containing the shaded image unless the current graphics output device is the Z-buffer. If the current output device is the Z-buffer, the results are merged with the Z-buffer’s contents and the function result contains a dummy value.

Arguments

Vertices

A $(3, n)$ array containing the X, Y, and Z coordinates of each vertex. Coordinates can be in either data or normalized coordinates, depending on which keywords are present.

X, Y, Z

The X, Y, and Z coordinates of each vertex can, alternatively, be specified as three array expressions of the same dimensions.

Polygons

An integer or longword array containing the indices of the vertices for each polygon. The vertices of each polygon should be listed in counterclockwise order when observed from outside the surface. The vertex description of each polygon is a vector of the form: $[n, i_0, i_1, \dots, i_{n-1}]$ and the array *Polygons* is the concatenation of the lists of each polygon. For example, to render a pyramid consisting of four triangles, *Polygons* would contain 16 elements, made by concatenating four, four-element vectors of the form $[3, V_0, V_1, V_2]$. V_0 , V_1 , and V_2 are the indices of the vertices describing each triangle.

Keywords

DATA

Set this keyword to indicate that the vertex coordinates are in data units, the default coordinate system.

NORMAL

Set this keyword to indicate that coordinates are in normalized units, within the three dimensional (0,1) cube.

POLY_SHADES

An array expression, with the same number of elements as there are polygons defined in the *Polygons* array, containing the color index used to render each polygon. No interpolation is performed if all pixels within a given polygon have the same shade value. For most displays, this parameter should be scaled into the range of bytes.

SHADES

An array expression, with the same number of elements as *Vertices*, containing the color index at each vertex. The shading of each pixel is interpolated from the surrounding SHADE values. For most displays, this parameter should be scaled into the range of bytes.

Warning

When using the SHADES keyword on TrueColor devices, we recommend that decomposed color support be turned off by setting DECOMPOSED=0 for [DEVICE](#).

T3D

Set this keyword to use the three-dimensional to two-dimensional transformation contained in the homogeneous 4 by 4 matrix !P.T. Note that if T3D is set, !P.T must contain a valid transformation matrix. The SURFACE, SCALE3, and T3D procedures (and others) can all be used to set up transformations.

TOP

The maximum shading value when light source shading is in effect. The default value is one less than the number of colors available in the currently selected graphics device.

XSIZE

The number of columns in the output image array. If this parameter is omitted, the number of columns is equal to the X size of the currently selected display device.

Warning: The size parameters should be explicitly specified when the current graphics device is PostScript or any other high-resolution device. Making the output image the default full device size is likely to cause an insufficient memory error.

YSIZE

The number of rows in the output image array. If this parameter is omitted, the number of rows is equal to the Y resolution of the currently selected display device.

Examples

POLYSHADE is often used in conjunction with SHADE_VOLUME for volume visualization. The following example creates a spherical volume dataset and renders an isosurface from that dataset:

```
; Create an empty, 3-D array:
SPHERE = FLTARR(20, 20, 20)

; Create the spherical dataset:
FOR X=0,19 DO FOR Y=0,19 DO FOR Z=0,19 DO $
    SPHERE(X, Y, Z) = SQRT((X-10)^2 + (Y-10)^2 + (Z-10)^2)

; Find the vertices and polygons for a density level of 8:
SHADE_VOLUME, SPHERE, 8, V, P

; Set up an appropriate 3-D transformation so we can see the
; sphere. This step is very important:
SCALE3, XRANGE=[0,20], YRANGE=[0,20], ZRANGE=[0,20]

; Render the image. Note that the T3D keyword has been set so that
; the previously-established 3-D transformation is used:
image = POLYSHADE(V, P, /T3D)

; Display the image:
TV, image
```

Version History

Introduced: Original

See Also

[IVOLUME](#), [PROJECT_VOL](#), [RECON3](#), [SET_SHADING](#), [SHADE_SURF](#),
[SHADE_VOLUME](#), [VOXEL_PROJ](#)

POLYWARP

The POLYWARP procedure performs polynomial spatial warping.

Using least squares estimation, POLYWARP determines the coefficients $Kx_{i,j}$ and $Ky_{(i,j)}$ of the polynomial functions:

$$X_i = \sum_{i,j} Kx_{i,j} \cdot Xo^j \cdot Yo^i$$

$$Y_i = \sum_{i,j} Ky_{i,j} \cdot Xo^j \cdot Yo^i$$

Kx and Ky can be used as inputs P and Q to the built-in function `POLY_2D`. This coordinate transformation may be then used to map from Xo , Yo coordinates into Xi , Yi coordinates.

This routine is written in the IDL language. Its source code can be found in the file `polywarp.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

POLYWARP, Xi , Yi , Xo , Yo , $Degree$, Kx , Ky [, /DOUBLE] [, STATUS=*variable*]

Arguments

Xi , Yi

Vectors of X and Y coordinates to be fit as a function of Xo and Yo .

Xo , Yo

Vectors of X and Y independent coordinates. These vectors must have the same number of elements as Xi and Yi .

Degree

The degree of the fit. The number of coordinate pairs must be greater than or equal to $(Degree+1)^2$.

Kx

A named variable that will contain the array of coefficients for X_i as a function of (X_o, Y_o) . This parameter is returned as a $(Degree+1)$ by $(Degree+1)$ element array.

Ky

A named variable that will contain the array of coefficients for Y_i . This parameter is returned as a $(Degree+1)$ by $(Degree+1)$ element array.

Keywords**DOUBLE**

Set this keyword to use double-precision for computations and to return a double-precision result. Explicitly set `DOUBLE=0` to ensure the use of single-precision for computations and to return a single-precision result. By default, `POLYWARP` performs computations in double precision and returns a double-precision result if any of the inputs are double-precision; computations are performed in single precision and the result returned as single-precision if all of the inputs are single-precision.

STATUS

Set this keyword to a named variable to receive the status of the operation. Possible status values are:

Value	Description
0	Successful completion.
1	Singular array (which indicates that the inversion is invalid).
2	Warning that a small pivot element was used and that significant accuracy was probably lost.

Table 3-95: STATUS Keyword Values

Note

If `STATUS` is not specified, any warning messages will be output to the screen.

Examples

The following example shows how to display an image and warp it using the POLYWARP and POLY_2D routines.

```
; Create and display the original image:
A = BYTSCL(SIN(DIST(250)))
TVSCL, A

; Now set up the Xi's and Yi's:
XI = [24, 35, 102, 92]
YI = [81, 24, 25, 92]

; Enter the Xo's and Yo's:
XO = [61, 62, 143, 133]
YO = [89, 34, 38, 105]

; Run POLYWARP to obtain a Kx and Ky:
POLYWARP, XI, YI, XO, YO, 1, KX, KY

; Create a warped image based on Kx and Ky with POLY_2D:
B = POLY_2D(A, KX, KY)

; Display the new image:
TV, B
```

Version History

Introduced: Original

See Also

[POLY_2D](#), [WARP_TRI](#)

POPD

The POPD procedure changes the current working directory to the directory saved on the top of the directory stack maintained by the PUSHHD and POPD procedures. This top entry is then removed from the stack.

Attempting to pop a directory when the stack is empty causes a warning message to be printed. The current directory is not changed in this case. The common block DIR_STACK is used to store the directory stack.

This routine is written in the IDL language. Its source code can be found in the file `popd.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

POPD

Arguments

None.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[CD](#), [PRINTD](#), [PUSHHD](#)

POWELL

The POWELL procedure minimizes a user-written function *Func* of two or more independent variables using the Powell method. POWELL does not require a user-supplied analytic gradient.

POWELL is based on the routine `powell` described in section 10.5 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
POWELL, P, Xi, Ftol, Fmin, Func [, /DOUBLE] [, ITER=variable]  
[, ITMAX=value]
```

Arguments

P

On input, *P* is an *n*-element vector specifying the starting point. On output, it is replaced with the location of the minimum.

Xi

On input, *Xi* is an initial *n* by *n* element array whose columns contain the initial set of directions (usually the *n* unit vectors). On output, it is replaced with the then-current directions.

Ftol

An input value specifying the fractional tolerance in the function value. Failure to decrease by more than *Ftol* in one iteration signals completeness. For single-precision computations, a value of 1.0×10^{-4} is recommended; for double-precision computations, a value of 1.0×10^{-8} is recommended.

Fmin

On output, *Fmin* contains the value at the minimum-point *P* of the user-supplied function specified by *Func*.

Func

A scalar string specifying the name of a user-supplied IDL function of two or more independent variables to be minimized. This function must accept a vector argument X and return a scalar result.

For example, suppose we wish to minimize the function

$$f(x, y) = (x + 2y)e^{-x^2 - y^2}$$

To evaluate this expression, we define an IDL function named POWFUNC:

```
FUNCTION powfunc, X
  RETURN, (X[0] + 2.0*X[1]) * EXP(-X[0]^2 -X[1]^2)
END
```

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ITER

Use this keyword to specify an output variable that will be set to the number of iterations performed.

ITMAX

Use this keyword to specify the maximum allowed number of iterations. The default is 200.

Warning

POWELL halts once the value specified with ITMAX has been reached.

Examples

We can use POWELL to minimize the function POWFUNC given above.

```
PRO TEST_POWELL

  ; Define the fractional tolerance:
  ftol = 1.0e-4
```

```

; Define the starting point:
P = [.5d, -.25d]

; Define the starting directional vectors in column format:
xi = TRANSPOSE([[1.0, 0.0],[0.0, 1.0]])

; Minimize the function:
POWELL, P, xi, ftol, fmin, 'powfunc'

; Print the solution point:
PRINT, 'Solution point: ', P

; Print the value at the solution point:
PRINT, 'Value at solution point: ', fmin

END

FUNCTION powfunc, X
  RETURN, (X[0] + 2.0*X[1]) * EXP(-X[0]^2 -X[1]^2)
END

```

IDL prints:

```

Solution point:      -0.31622777      -0.63245552
Value at solution point:      -0.95900918

```

The exact solution point is [-0.31622777, -0.63245553].

The exact minimum function value is -0.95900918.

Version History

Introduced: 4.0

See Also

[AMOEB](#), [DFPMIN](#), [SIMPLEX](#)

PRIMES

The PRIMES function computes the first K prime numbers.

This routine is written in the IDL language. Its source code can be found in the file `primes.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = PRIMES(*K*)

Return Value

The result is a K -element long integer vector.

Arguments

K

An integer or long integer scalar that specifies the number of primes to be computed.

Examples

To compute the first 25 prime numbers:

```
PRINT, PRIMES(25)
```

IDL prints:

```
  2    3    5    7   11   13
 17   19   23   29   31   37
 41   43   47   53   59   61
 67   71   73   79   83   89
 97
```

Version History

Introduced: 4.0

PRINT/PRINTF

The two PRINT procedures perform formatted output. PRINT performs output to the standard output stream (IDL file unit -1), while PRINTF requires a file unit to be explicitly specified.

Format Compatibility

If the FORMAT keyword is not present and PRINT is called with more than one argument, and the first argument is a scalar string starting with the characters “\$(”, this initial argument is taken to be the format specification, just as if it had been specified via the FORMAT keyword. This feature is maintained for compatibility with version 1 of VMS IDL.

Syntax

```
PRINT [, Expr1, ..., Exprn]
```

```
PRINTF [, Unit, Expr1, ..., Exprn]
```

Keywords: [, [AM_PM](#)=[string, string]] [, [DAYS_OF_WEEK](#)=string_array{7 names}] [, [FORMAT](#)=value] [, [MONTHS](#)=string_array{12 names}] [, [/STDIO_NON_FINITE](#)]

Arguments

Unit

For PRINTF, *Unit* specifies the file unit to which the output is sent.

Expr_i

The expressions to be output.

Keywords

AM_PM

Supplies a string array of 2 names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the FORMAT keyword.

DAYS_OF_WEEK

Supplies a string array of 7 names to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the FORMAT keyword.

FORMAT

If FORMAT is not specified, IDL uses its default rules for formatting the output. FORMAT allows the format of the output to be specified in precise detail, using a FORTRAN-style specification. See “[Using Explicitly Formatted Input/Output](#)” in Chapter 11 of the *Building IDL Applications* manual.

MONTHS

Supplies a string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the FORMAT keyword.

STDIO_NON_FINITE

Set this keyword to allow the writing of data files readable by C or FORTRAN programs on a given platform; it is otherwise unnecessary. The various systems supported by IDL differ widely in the representation used for non-finite floating point values (i.e., NaN and Infinity). Consider that the following are all possible representations for NaN on at least one IDL platform:

```
NaN, NanQ, ?.0000, nan0x2, nan0x7, 1.#QNAN, -1.#IND0.
```

And the following are considered to be Infinity:

```
Inf, Infinity, ++.0000, ----.0000, 1.#INF
```

On input, IDL can recognize any of these, but on output, it uses the same standard representation on all platforms. This promotes cross-platform consistency. To cause IDL to use the system C library `sprintf()` function to format such values, yielding the native representation for that platform, set the `STDIO_NON_FINITE` keyword.

Obsolete Keywords

The following keywords are obsolete:

- REWRITE

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

To print the string “IDL is fun.” enter the command:

```
PRINT, 'IDL is fun.'
```

To print the same message to the open file associated with file unit number 2, use the command:

```
PRINTF, 2, 'IDL is fun.'
```

Version History

Introduced: Original

See Also

[ANNOTATE](#), [MESSAGE](#), [WRITEU](#), [XYOUTS](#)

PRINTD

The PRINTD procedure prints the contents of the directory stack maintained by the PUSHHD and POPD procedures. The contents of the directory stack are listed on the default output device. The common block DIR_STACK is used to store the directory stack.

This routine is written in the IDL language. Its source code can be found in the file `printd.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

PRINTD

Arguments

None.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[CD](#), [POPD](#), [PUSHHD](#)

PRO

The PRO statement defines an IDL procedure.

Note

For information on using the PRO statement, see [Chapter 4, “Procedures and Functions”](#) in the *Building IDL Applications* manual.

Syntax

```
PRO Procedure_Name, argument_1, ..., argument_n
    ...
END
```

Arguments

argument_n

A parameter that is passed to the procedure.

Keywords

None.

Examples

The following example demonstrates the use of arguments in a PRO statement:

```
PRO MYPROCEDURE
    X = 5
    ; Call the ADD procedure:
    ADD, 3, X
END

PRO ADD, A, B
    PRINT, 'A = ', A
    PRINT, 'B = ', B
    A = A + B
    PRINT, 'A = ', A
END
```

After running `myprocedure.pro`, IDL returns:

```
A = 3  
B = 5  
A = 8
```

Version History

Introduced: Original

PRODUCT

The `PRODUCT` function returns the product of elements within an array. The product of the array elements over a given dimension is returned if the *Dimension* argument is present. Because the product can easily overflow, the product is computed using double-precision arithmetic and the *Result* is double precision.

Tip

If your array has a mix of very large and very small values, the product may underflow or overflow during the computation, even though the final result would be within double-precision limits. In this case, you should not use `PRODUCT`, but instead compute the product by taking the logarithm, using the `TOTAL` function, and then taking the exponential: $Result = EXP(TOTAL(ALOG(Array)))$.

Syntax

```
Result = PRODUCT(Array [, Dimension] [, /CUMULATIVE] [, /INTEGER]
                [, /NAN] [, /PRESERVE_TYPE] )
```

Return Value

Returns the product of the elements of *Array*.

Arguments

Array

The array for which to compute the product. This array can be of any basic type except string.

Dimension

An optional argument specifying the dimension over which to compute the product, starting at one. If this argument is not present or zero, the product of all the array elements is returned. If this argument is present, the result is an array with one less dimension than *Array*. For example, if the dimensions of *Array* are N_1, N_2, N_3 , and *Dimension* is 2, the dimensions of the result are (N_1, N_3) , and element (i,j) of the result contains the product:

$$R_{i,j} = \prod_{k=0}^{N_2-1} A_{i,k,j}$$

Keywords

CUMULATIVE

If this keyword is set, the result is an array of the same size as the input, with each element, i , containing the product of the input array elements 0 to i . This keyword also works with the *Dimension* parameter, in which case the cumulative product is performed over the given dimension.

Tip

If the input array is a temporary variable or an expression, and the result type matches the input type (for example by using the `PRESERVE_TYPE` keyword), then `PRODUCT` will perform the cumulative product in place and no additional memory will be used.

INTEGER

Set this keyword to perform the `PRODUCT` using integer arithmetic, and to return an integer result. If *Array* is of type `ULONG64` then unsigned 64-bit integers are used for the computation and the Result is of type `ULONG64`, otherwise signed 64-bit integers are used and the Result is of type `LONG64`. If *Array* is complex and `INTEGER` is set, then only the real part of each value is used for the computation.

Note

If *Array* has a large number of values or the values themselves are large, then the `PRODUCT` with the `INTEGER` keyword may easily overflow the largest 64-bit integer and return an incorrect result. In this case you may want to avoid using the `INTEGER` keyword.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. Elements with the value *NaN* are treated as missing data

with the value 1. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

PRESERVE_TYPE

Set this keyword to perform the `PRODUCT` using the input type, and to return a result of the same type. The `INTEGER` keyword is ignored if `PRESERVE_TYPE` is set.

Note

For byte or integer data the `PRODUCT` with `PRESERVE_TYPE` may easily overflow the largest value for that type. In these cases you may want to use the `INTEGER` keyword instead.

Thread Pool Keywords

This routine is written to make use of IDL's thread pool, which can increase execution speed on systems with multiple CPUs. The values stored in the `!CPU` system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#).

Examples

To find the product of all elements in a one-dimensional array:

```
; Define a one-dimensional array:
array = [20, 10, 5, 5, 3]

; Find the product of the array elements:
prod = PRODUCT(array)

; Print the results:
PRINT, 'Product of Array = ', prod
```

IDL prints:

```
Product of Array =          15000.000
```

Now find the product of elements in a two-dimensional array:

```

; Define a two-dimensional array:
array = FINDGEN(4,4) + 1

; Find the product of all array elements:
prodAll = PRODUCT(array)

; Find the product along the first dimension:
prod1 = PRODUCT(array, 1)

; Find the product along the second dimension:
prod2 = PRODUCT(array, 2)

; Print the results:
PRINT, 'Product of all elements = ', prodAll
PRINT, 'Product along first dimension: '
PRINT, prod1
PRINT, 'Product along second dimension: '
PRINT, prod2

```

IDL prints:

```

Product of all elements    2.0922790e+013
Product along first dimension:
 24.000000      1680.0000      11880.000      43680.000
Product along second dimension:
 585.00000     1680.0000     3465.0000     6144.0000

```

Version History

Introduced: 5.6

INTEGER and PRESERVE_TYPE keywords: 6.1

See Also

[FACTORIAL](#), [TOTAL](#)

PROFILE

The PROFILE function extracts a profile from an image and returns the values of the image along the profile line marked by the user.

This routine is written in the IDL language. Its source code can be found in the file `profile.pro` in the `lib` subdirectory of the IDL distribution.

Using PROFILE

To mark a profile line after calling PROFILE, click in the image with the left mouse button to mark the beginning and ending points. The pixel coordinates of the selected points are displayed in the IDL command log.

Syntax

```
Result = PROFILE( Image [, XX, YY] [, /NOMARK] [, XSTART=value]  
                [, YSTART=value] )
```

Return Value

Returns a floating-point vector containing the values along the profile line.

Arguments

Image

The data array representing the image. This array can be of any type except complex.

XX

A named variable that will contain the X coordinates of the points along the selected profile.

YY

A named variable that will contain the Y coordinates of the points along the selected profile.

Keywords

NOMARK

Set this keyword to inhibit marking the image with the profile line.

XSTART

The starting X location of the lower-left corner of *Image*. If this keyword is not specified, 0 is assumed.

YSTART

The starting Y location of the lower-left corner of *Image*. If this keyword is not specified, 0 is assumed.

Examples

This example displays an image, selects a profile, and plots that profile in a new window:

```
; Create an image:  
A = BYTSCL(DIST(256))  
  
; Display the image:  
TV, A  
  
; Extract a profile from the image:  
R = PROFILE(A)
```

Mark two points on the image with the mouse.

```
; Create a new plotting window:  
WINDOW, /FREE  
  
; Plot the profile:  
PLOT, R
```

Note

The PROFILES procedure is an interactive version of this routine.

Version History

Introduced: Original

See Also

[PROFILES](#)

PROFILER

The PROFILER procedure allows you to access the IDL Code Profiler. The IDL Code Profiler helps you analyze the performance of your applications. You can easily monitor the calling frequency and execution time for procedures and functions.

Syntax

```
PROFILER [, Module] [, /CLEAR] [, DATA=variable] [, OUTPUT=variable]  
[, /REPORT] [, /RESET] [, /SYSTEM]
```

Arguments

Module

The program to which changes in profiling will be applied. If *Module* is not specified, profiling changes will be applied to all currently-compiled programs.

Note

The *Module* is often named with respect to the file in which it is stored. For example, the file `build_it.pro` may contain the module, `build_it`. If you specify the file name, you will incur a syntax error.

Keywords

CLEAR

Set this keyword to disable profiling of *Module* or of all compiled modules if *Module* is not specified.

DATA

Set this keyword to a specified variable containing output of the report as an unnamed structure with the following tags and data types:

```
{NAME:char, COUNT:long, ONLY_TIME:double, TIME:double,  
SYSTEM:byte}
```

OUTPUT

Set this keyword to a specified variable in which to store the results of the REPORT keyword.

REPORT

Set this keyword to report the results of profiling. If you enter a program at the command line, the PROFILER procedure will report the status of all the specified modules used either since the beginning of the IDL session, or since the PROFILER was reset.

RESET

Set this keyword to clear the results of profiling.

SYSTEM

Set this keyword to profile IDL system procedures and functions. By default, only user-written or library files, which have been compiled, are profiled.

Examples

```
; Include IDL system procedures and functions when profiling:
PROFILER, /SYSTEM

; Create a dataset using the library function DIST. Note that DIST
; is immediately compiled:
A= DIST(500)

; Display the image:
TV, A

; Retrieve the profiling results:
PROFILER, /REPORT
```

IDL prints:

Module	Type	Count	Only(s)	Avg.(s)	Time(s)	Avg.(s)
FINDGEN	(S)	1	0.000239	0.000239	0.000239	0.000239
FLTARR	(S)	1	0.010171	0.010171	0.010171	0.010171
N_ELEMENTS	(S)	1	0.000104	0.000104	0.000104	0.000104
ON_ERROR	(S)	1	0.000178	0.000178	0.000178	0.000178
SQRT	(S)	251	0.099001	0.000394	0.099001	0.000394
TV	(S)	1	2.030000	2.030000	2.030000	2.030000

Version History

Introduced: 5.1

See Also

[Chapter 18, “Debugging an IDL Program”](#) in the *Building IDL Applications* manual.

PROFILES

The `PROFILES` procedure interactively draws row or column profiles of an image in a separate window. A new window is created and the mouse location in the original window is used to plot profiles in the new window.

This routine is written in the IDL language. Its source code can be found in the file `profiles.pro` in the `lib` subdirectory of the IDL distribution.

Using PROFILES

Moving the mouse within the original image interactively creates profile plots in the newly-created profile window. Pressing the left mouse button toggles between row and column profiles. The right mouse button exits.

Syntax

```
PROFILES, Image [, /ORDER] [, SX=value] [, SY=value] [, WSIZE=value]
```

Arguments

Image

The variable that represents the image displayed in the current window. This data need not be scaled into bytes. The profile graphs are made from this array, even if it is not currently displayed.

Keywords

ORDER

Set this keyword to 1 for images written top down or 0 for bottom up. Default is the current value of `!ORDER`.

SX

Starting X position of the image in the window. If this keyword is omitted, 0 is assumed.

SY

Starting Y position of the image in the window. If this keyword is omitted, 0 is assumed.

WSIZE

The size of the PROFILES window as a fraction or multiple of 640 by 512.

Examples

Create and display an image and use the PROFILES routine on it.

```
; Create an image:  
A = BYTSCL(DIST(256))  
  
; Display the image:  
TV, A  
  
; Run the PROFILES routine:  
PROFILES, A, WSIZE = .5
```

A 320 x 256 pixel PROFILES window should appear. Move the cursor over the original image to see the profile at the cursor position. Press the left mouse button to toggle between row and column profiles. Press the right mouse button (with the cursor over the original image) to exit the routine.

Version History

Introduced: Pre 4.0

See Also

[PROFILE](#)

PROJECT_VOL

The `PROJECT_VOL` function returns a two-dimensional image that is the projection of a 3-D volume of data onto a plane (similar to an X-ray). The returned image is a translucent rendering of the volume (the highest data values within the volume show up as the brightest regions in the returned image). Depth queuing and opacity may be used to affect the image. The volume is projected using a 4x4 matrix, so any type of projection may be used including perspective. Typically the system viewing matrix (!P.T) is used as the 4x4 matrix.

Note

The `VOXEL_PROJ` procedure performs many of the same functions as this routine, and is faster.

This routine is written in the IDL language. Its source code can be found in the file `project_vol.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = PROJECT_VOL( Vol, X_Sample, Y_Sample, Z_Sample
    [, /AVG_INTENSITY] [, /CUBIC] [, DEPTH_Q=value] [, OPAQUE=3D_array]
    [, TRANS=array] [, XSIZE=longword integer] [, YSIZE=longword integer]
    [, /Z_BUFFER] )
```

Return Value

Returns a projection of the volumetric data.

Arguments

Vol

A 3-D array of any type except string or structure containing the three-dimensional volume of data to project.

X_Sample

A long integer specifying the number of rays to project along the X dimension of the image. The returned image will have the dimensions `X_sample` by `Y_sample`.

Y_Sample

A long integer specifying the number of rays to project along the Y dimension of the image. To preserve the correct aspect ratio of the data, *Y_sample* should equal *X_sample*.

Z_Sample

A long integer specifying the number of samples to take along each ray. Higher values for *X_sample*, *Y_sample*, and *Z_sample* increase the image resolution as well as execution time.

Keywords

AVG_INTENSITY

If this keyword is set, the average intensity method of projection is used. The default is a maximum intensity projection. This keyword is ignored if the *Z_BUFFER* keyword is set.

CUBIC

If this keyword is set, the cubic method of interpolation is used. The default is bilinear interpolation.

DEPTH_Q

Set this keyword to indicate that the image should be created using depth queuing. The depth queuing should be a single floating-point value between 0.0 and 1.0. This value specifies the brightness of the farthest regions of the volume relative to the closest regions of the volume. A value of 0.0 will cause the back side of the volume to be completely blacked out, while a value of 1.0 indicates that the back side will show up just as bright as the front side. The default is 1.0 (indicating no depth queuing).

OPAQUE

A 3-D array of any type except string or structure, with the same size and dimensions as *Vol*. This array specifies the opacity of each cell in the volume. *OPAQUE* values of 0 allow all light to pass through. *OPAQUE* values are cumulative. For example, if a ray emanates from a data value of 50, and then passes through 10 opaque cells (each with a data value of 0 and an opacity value of 5) then that ray would be completely blocked out (the cell with the data value of 50 would be invisible on the returned image). The default is no opacity.

TRANS

A 4x4 floating-point array to use as the transformation matrix when projecting the volume. The default is to use the system viewing matrix (!P.T).

XSIZE

The x size of the returned image to return. CONGRID is used to resize the final image to be XSIZE by YSIZE. The default is the x size of the current window (or the x size of the Z-buffer). If no current window exists, then the default is X_sample .

YSIZE

The y size of the returned image. CONGRID is used to resize the final image to be XSIZE by YSIZE. The default is the y size of the current window (or the y size of the Z-buffer). If no current window exists, then the default is Y_sample .

Z_BUFFER

If this keyword is set, the projection is combined with the contents of the Z-buffer. The default is to not use the Z-buffer contents.

Examples

Use the T3D routine to set up a viewing projection and render a volume of data using PROJECT_VOL.

```

; First, create some data:
vol = RANDOMU(S, 40, 40, 40)
FOR I=0, 10 DO vol = SMOOTH(vol, 3)
vol = BYTSCL(vol(3:37, 3:37, 3:37))
opaque = RANDOMU(S, 40, 40, 40)
FOR I=0, 10 DO opaque = SMOOTH(opaque, 3)
opaque = BYTSCL(opaque(3:37, 3:37, 3:37), TOP=25B)

; Set up the view:
xmin = 0 & ymin = 0 & zmin = 0
xmax = 34 & ymax = 34 & zmax = 34
!X.S = [-xmin, 1.0] / (xmax - xmin)
!Y.S = [-ymin, 1.0] / (ymax - ymin)
!Z.S = [-zmin, 1.0] / (zmax - zmin)
T3D, /RESET
T3D, TRANSLATE=[-0.5, -0.5, -0.5]
T3D, SCALE=[0.7, 0.7, 0.7]
T3D, ROTATE=[30, -30, 60]
T3D, TRANSLATE=[0.5, 0.5, 0.5]
WINDOW, 0, XSIZE=512, YSIZE=512

```

```
; Generate and display the image:  
img = PROJECT_VOL(vol, 64, 64, 64, DEPTH_Q=0.7, $  
    OPAQUE=opaque, TRANS=(!P.T))  
TVSCL, img
```

Version History

Introduced: Pre 4.0

See Also

[POLYSHADE](#), [VOXEL_PROJ](#)

PS_SHOW_FONTS

The PS_SHOW_FONTS procedure displays all the PostScript fonts that IDL knows about, with both the StandardAdobe and ISOLatin1 encodings. Each display takes a separate page, and each character in each font is shown with its character index.

A PostScript file is produced, one page per font/mapping combination. The output file contains almost 70 pages of output. A PostScript previewer is recommended rather than sending it to a printer.

This routine is written in the IDL language. Its source code can be found in the file `ps_show_fonts.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
PS_SHOW_FONTS [, /NOLATIN]
```

Arguments

None.

Keywords

NOLATIN

Set this keyword to prevent output of ISOLatin1 encodings.

Version History

Introduced: Pre 4.0

See Also

[PSAFM](#)

PSAFM

The PSAFM procedure takes an Adobe Font Metrics file as input and generates a new AFM file in the format that IDL likes. This new file differs from the original in the following ways:

- Information not used by IDL is removed.
- AFM files with the `AdobeStandardEncoding` are supplemented with an `ISOLatin1Encoding`.

This routine is written in the IDL language. Its source code can be found in the file `psafm.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

`PSAFM, Input_Filename, Output_Filename`

Arguments

Input_Filename

A string that contains the name of existing AFM file from Adobe.

Output_Filename

A string that specifies the name of new IDL-format AFM file to be created.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[PS_SHOW_FONTS](#)

PSEUDO

The PSEUDO procedure creates a pseudo-color table based on the LHB (Lightness, Hue, and Brightness) system and loads it.

The pseudo-color mapping used is generated by first translating from the LHB coordinate system to the LAB coordinate system, finding N colors spread out along a helix that spans this LAB space (supposedly a near maximal entropy mapping for the eye, given a particular N) and remapping back into the RGB (Red, Green, and Blue) colorspace. The result is loaded as the current colortable.

This routine is written in the IDL language. Its source code can be found in the file `pseudo.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

PSEUDO, *Litlo*, *Lithi*, *Satlo*, *Sathi*, *Hue*, *Loops* [, *Colr*]

Arguments

Litlo

Starting lightness, from 0 to 100%.

Lithi

Ending lightness, from 0 to 100%.

Satlo

Starting saturation, from 0 to 100%.

Sathi

Ending saturation, from 0 to 100%.

Hue

Starting hue, in degrees, from 0 to 360.

Loops

The number of loops of hue to make in the color helix. This value can range from 0 to around 3 to 5 and need not be an integer.

Colr

An optional (256,3) integer array in which the new R, G, and B values are returned.
Red = *Colr*[:,0], green = *Colr*[:,1], blue = *Colr*[:,2].

Keywords

None.

Version History

Introduced: Original

See Also

[COLOR_CONVERT](#), [COLOR_QUAN](#)

PTR_FREE

The PTR_FREE procedure destroys the heap variables pointed at by its pointer arguments. Any memory used by the heap variable is released, and the variable ceases to exist. No change is made to the arguments themselves and all pointers to the destroyed variables continue to exist. Such pointers are known as *dangling references*. PTR_FREE is the only way that pointer heap variables can be destroyed. If PTR_FREE is not called on a heap variable, it continues to exist until the IDL session ends, even if no pointers remain that can be used to reference it.

Note

PTR_FREE is not recursive. That is, if the heap variable pointed at by `pointer1` contains `pointer2`, destroying `pointer1` will *not* destroy the heap variable pointed at by `pointer2`. Take care not to lose the only pointer to a heap variable by destroying a pointer to a heap variable that contains that pointer.

Syntax

PTR_FREE, P_1 ,, P_n

Arguments

P_i

Scalar or array arguments of pointer type. If the NULL pointer is passed, PTR_FREE ignores it quietly.

Keywords

None.

Version History

Introduced: 5.0

PTR_NEW

The PTR_NEW function provides the primary mechanism for creating heap variables.

Syntax

```
Result = PTR_NEW( [InitExpr] [, /ALLOCATE_HEAP] [, /NO_COPY] )
```

Return Value

It returns a pointer to the created variable.

Arguments

InitExpr

If *InitExpr* is provided, PTR_NEW uses it to initialize the newly created heap variable. Note that the new heap variable does not point at the *InitExpr* variable in any sense—the new heap variable simply contains a copy of its value.

If *InitExpr* is not provided, PTR_NEW does not create a new heap variable, and returns the *Null Pointer*, a special pointer with a fixed known value that can never point at a heap variable. The null pointer is useful as a terminator in dynamic data structures, or as a placeholder in structure definitions.

Keywords

ALLOCATE_HEAP

Set this keyword to cause PTR_NEW to allocate an undefined heap variable rather than return a null pointer when *InitExpr* is not specified.

NO_COPY

Usually, when the *InitExpr* argument is provided, PTR_NEW allocates additional memory to make a copy. If the NO_COPY keyword is set, the value data is taken away from the *InitExpr* variable and attached directly to the heap variable. This feature can be used to move data very efficiently. However, it has the side effect of causing the *InitExpr* variable to become undefined. Using the NO_COPY keyword is completely equivalent to the statement:

```
Result = PTR_NEW(TEMPORARY(InitExpr))
```

and is provided as a syntactic convenience.

Version History

Introduced: 5.0

PTR_VALID

The PTR_VALID function verifies the validity of its pointer arguments, or alternatively returns a vector of pointers to all the existing valid pointer heap variables.

Syntax

Result = PTR_VALID([Arg] [, /CAST] [, COUNT=*variable*])

Return Value

If called with an pointer or array of pointers as its argument, PTR_VALID returns a byte array of the same size as the argument. Each element of the result is set to True (1) if the corresponding pointer in the argument refers to an existing valid heap variable, or to False (0) otherwise.

If called with an integer or array of integers as its argument and the CAST keyword is set, PTR_VALID returns an array of pointers. Each element of the result is a pointer to the heap variable indexed by the integer value. Integers used to index heap variables are shown in the output of the HELP and PRINT commands. This is useful primarily in programming/debugging when you have lost a reference but see it with HELP and need to get a reference to it interactively in order to determine what it is and take steps to fix the code. See the “Examples” section below for an example.

If no argument is specified, PTR_VALID returns a vector of pointers to all existing valid pointer heap variables—even if there are currently no pointers to the heap variable. This usage allows you to “reclaim” pointer heap variables to which all pointers have been lost. If no valid pointer heap variables exist, a scalar null pointer is returned.

Arguments

Arg

Arg can be one of the following:

1. A scalar or array argument of pointer type.
2. If the CAST keyword is set, an integer index or array of integer indices to heap variables. Integers used to index heap variables are shown in the output of the HELP and PRINT commands.

Keywords

CAST

Set this keyword to create a new pointer to each heap variable index specified in *Arg*.

COUNT

Set this keyword equal to a named variable that will contain the number of currently valid heap variables. This value is returned as a longword integer.

Examples

To determine if a given pointer refers to a valid heap variable:

```
IF (PTR_VALID(p)) THEN ...
```

To destroy all existing pointer heap variables:

```
PTR_FREE, PTR_VALID()
```

You can use the **CAST** keyword to “reclaim” lost heap variables. For example:

```
A = PTR_NEW(10)
PRINT, A
```

IDL prints:

```
<PtrHeapVar2>
```

In this case, the integer index to the heap variable is 2. If we reassign the variable **A**, we will “lose” the pointer, but the heap variable will still exist:

```
A=0
PRINT, A, PTR_VALID()
```

IDL prints:

```
0 <PtrHeapVar2>
```

We can reclaim the lost heap variable using the **CAST** keyword:

```
A = PTR_VALID(2, /CAST)
PRINT, A
```

IDL prints:

```
<PtrHeapVar2>
```

Version History

Introduced: 5.0

PTRARR

The PTRARR function returns a pointer vector or array. The individual elements of the array are set to the Null Pointer.

Syntax

$$Result = PTRARR(D_1, \dots, D_8 [, /ALLOCATE_HEAP | , /NOZERO])$$

Return Value

Returns a vector or array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

ALLOCATE_HEAP

Normally, PTRARR sets every element of the result to the null pointer. If you wish IDL to allocate heap variables for every element of the array instead, set the ALLOCATE_HEAP keyword. In this case, every element of the array will be initialized to point at an undefined heap variable.

NOZERO

If ALLOCATE_HEAP is not specified, PTRARR sets every element of the result to the null pointer. If NOZERO is nonzero, this initialization is not performed and PTRARR executes faster. NOZERO is ignored if ALLOCATE_HEAP is specified.

Warning

If you specify NOZERO, the resulting array will have whatever value happens to exist at the system memory location that the array is allocated from. You should be careful to initialize such an array to valid pointer values.

Example

Create P, a 3 element by 3 element pointer array with each element containing the Null Pointer by entering:

```
P = PTRARR(3, 3)
```

Version History

Introduced: 5.0

PUSHD

The PUSHD procedure pushes a directory onto the top of the directory stack maintained by the PUSHD and POPD procedures. The name of the current directory is pushed onto the directory stack. This directory becomes the next directory used by POPD. IDL changes directories to the one specified by the *Dir* argument. The common block DIR_STACK is used to store the directory stack.

This routine is written in the IDL language. Its source code can be found in the file `pushd.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

PUSHD, *Dir*

Arguments

Dir

A string containing the name of the directory to change to. The current directory is pushed onto the top of the directory stack.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[CD](#), [POPD](#), [PRINTD](#)

QGRID3

The QGRID3 function linearly interpolates the dependent variable values to points in a regularly sampled volume. Its inputs are a triangulation of scattered data points in three dimensions, and the value of a dependent variable for each point.

Note

QGRID3 accepts complex input but only utilizes the real part of any complex number.

Syntax

```
Result = QGRID3( XYZ, F, Tetrahedra [, DELTA=vector ] [, DIMENSION=vector ]
               [, MISSING=value ] [, START=vector ] )
```

or

```
Result = QGRID3( X, Y, Z, F, Tetrahedra [, DELTA=array ] [, DIMENSION=array ]
               [, MISSING=value ] [, START=array ] )
```

Return Value

Result is a 3-dimensional array of either single or double precision floating type, of the specified dimensions.

Arguments

XYZ

This is a 3-by-*n* array containing the scattered points.

X, Y, Z

One-dimensional vectors containing the *X*, *Y*, and *Z* point coordinates.

F

The function value at each location in the form of an *n*-element vector.

Tetrahedra

A longword array containing the point indices of each tetrahedron, as created by QHULL.

Keywords

Note

Any of the keywords may be set to a scalar if all elements are the same.

DELTA

A scalar or three element array specifying the grid spacing in X, Y, and Z. If this keyword is not specified, it is set to create a grid of DIMENSION cells, enclosing the volume from START to [max(x), max(y), max(z)].

DIMENSION

A three element array specifying the grid dimensions in X, Y, and Z. Default value is 25 for each dimension.

MISSING

The value to be used for grid points that lie outside the convex hull of the scattered points. The default is 0.

START

A three element array specifying the start of the grid in X, Y, and Z. Default value is [min(x), min(y), min(z)].

Examples

Example 1

This example interpolates a data set measured on an irregular grid.

```

; Create a dataset of N points.
n = 200
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)
z = RANDOMU(seed, n)

; Create dependent variable.
f = x^2 - x*y + z^2 + 1

; Obtain a tetrahedra using the QHULL procedure.
QHULL, x, y, z, tet, /DELAUNAY

; Create a volume with dimensions [51, 51, 51]
```

```

; over the unit cube.
volume = QGRID3(x, y, z, f, tet, START=0, DIMENSION=51, $
    DELTA=0.02)

; Display the volume.
XVOLUME, BYTSCL(volume)

```

Example 2

This example is similar to the previous one, however in this example we use a [3, *n*] array of points.

```

; Create a dataset of N points.
n = 200
p = RANDOMU(seed, 3, n)

; Create dependent variable.
f = p[0,*]^2 - p[0,*]*p[1,*] + p[2,*]^2 + 1

; Obtain a tetrahedra.
QHULL, p, tet, /DELAUNAY

; Create a volume with dimensions [51, 51, 51] over the unit cube.
volume = QGRID3(p, f, tet, START=0, DIMENSION=51, DELTA=0.02)

; Display the volume.
XVOLUME, BYTSCL(volume)

```

Example 3

The following example uses the data from the `irreg_grid2.txt` ASCII file. This file contains scattered three-dimensional data. This file contains bore hole data for a square mile of land. The QHULL procedure is used to triangulate the three-dimensional locations. The QGRID3 function uses the results from QHULL to grid the data into a volume. The scattered data is displayed as symbol polyline objects in the XOBJVIEW utility. The resulting gridded volume is displayed in the XVOLUME utility:

```

; Import the Data:

; Determine the path to the file. This file contains bore hole
; data for a square mile of land. The bore hole samples were
; roughly taken diagonally from the upper left corner of the
; square to the lower right corner.
file = FILEPATH('irreg_grid2.txt', $
    SUBDIRECTORY = ['examples', 'data'])

; Import the data from the file into a structure.

```

```

dataStructure = READ_ASCII(file)

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
z = dataArray[*, 2]
data = dataArray[*, 3]

; Determine number of data points.
nPoints = N_ELEMENTS(data)

; Triangulate the Data with QHULL:

; Construct the convex hulls of the volume.
QHULL, x, y, z, tetrahedra, /DELAUNAY

; Grid the Data and Display the Results:

; Initialize volume parameters.
cubeSize = [51, 51, 51]
; Grid the data into a volume.
volume = QGRID3(x, y, z, data, tetrahedra, START = 0, $
    DIMENSION = cubeSize, DELTA = 0.02)
; Scale the volume to be able to view the full data value range
; with the color tables provided in the XVOLUME utility.
scaledVolume = BYTSCL(volume)

; Display the results in the XVOLUME utility.
XVOLUME, scaledVolume

; Derive the isosurface for mineral deposits with the data value
; of 2.5.
ISOSURFACE, volume, 2.5, vertices, connectivity

; Initialize a model to contain the isosurface.
oModel = OBJ_NEW('IDLgrModel')

; Initialize the polygon object of the isosurface.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, COLOR = [0, 0, 255])

; Determine the range in each direction.
xRange = [0, cubeSize[0]]
yRange = [0, cubeSize[1]]

```

```

zRange = [0, cubeSize[2]]

; Initialize an axis for each direction.
oAxes = OBJARR(3)
oAxes[0] = OBJ_NEW('IDLgrAxis', 0, RANGE = xRange, $
    LOCATION = [xRange[0], yRange[0], zRange[0]], /EXACT, $
    TICKLEN = (0.02*(yRange[1] - yRange[0])))
oAxes[1] = OBJ_NEW('IDLgrAxis', 1, RANGE = yRange, $
    LOCATION = [xRange[0], yRange[0], zRange[0]], /EXACT, $
    TICKLEN = (0.02*(xRange[1] - xRange[0])))
oAxes[2] = OBJ_NEW('IDLgrAxis', 2, RANGE = zRange, $
    LOCATION = [xRange[0], yRange[1], zRange[0]], /EXACT, $
    TICKLEN = (0.02*(xRange[1] - xRange[0])))

; Add the polygon and axes object to the model.
oModel->Add, oPolygon
oModel->Add, oAxes

; Rotate the model for a better perspective.
oModel->Rotate, [0, 0, 1], 30.
oModel->Rotate, [1, 0, 0], -45.

; Display the model, which contains the isosurface.
XOBJVIEW, oModel, /BLOCK, SCALE = 0.75, $
    TITLE = 'Isosurface at the Value of 2.5'

; Cleanup object references.
OBJ_DESTROY, [oModel]

```

Version History

Introduced: 5.5

See Also

[QHULL](#)

QHULL

The QHULL procedure constructs convex hulls, Delaunay triangulations, and Voronoi diagrams of a set of points of 2-dimensions or higher. It uses and is based on the program QHULL, which is described in Barber, Dobkin and Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No 4, December 1996, Pages 469-483.

Note

QHULL accepts complex input but only utilizes the real part of any complex number.

For more information about QHULL see <http://www.geom.umn.edu/software/qhull/>.

Syntax

QHULL, *V*, *Tr*

or,

QHULL, *V0*, *V1*, [, *V2* ... [, *V6*]], *Tr*

Keywords: [, **BOUNDS**=*variable*] [, **CONNECTIVITY**=*variable*] [, **/DELAUNAY**] [, **SPHERE**=*variable*] [, **VDIAGRAM**=*array*] [, **VNORMALS**=*array*] [, **VVERTICES**=*array*]

Arguments

V

An input argument providing an *nd*-by-*np* array containing the locations of *np* points, in *nd* dimensions. The number of dimensions, *nd*, must be greater than or equal to 2.

V0, V1, V2, ..., V(N-1)

Input vectors of dimension *np*-by-1 elements each containing the *i*-th coordinate of *np* points in *nd* dimensions. A maximum of seven input vectors may be specified.

Tr

An *nd1*-by-*nt* array containing the indices of either the convex hull (*nd1* is equal to *nd*), or the Delaunay triangulation (*nd1* is equal to *nd*+1) of the input points.

Keywords

BOUNDS

Set this keyword equal to a named variable that will contain the indices of the vertices of the convex hull.

Note

The order of the vertices returned in this variable is unspecified.

CONNECTIVITY

Set this keyword to a named variable in which the adjacency list for each of the np nodes is returned. The list has the following form:

Each element i , $0 \leq i < np$, contains the starting index of the connectivity list (*list*) for node i within the list array. To obtain the adjacency list for node i , extract the list elements from $list[i]$ to $list[i+1] - 1$. The adjacency list is not ordered. To obtain the connectivity list, either the DELAUNAY or SPHERE keywords must also be specified.

For example, to perform a spherical triangulation, use the following procedure call:

```
QHULL, lon, lat, CONNECTIVITY = list, SHPERE = sphere
```

which returns the adjacency list in the variable *list*. The subscripts of the nodes adjacent to $lon[i]$ and $lat[i]$ are contained in the array: $list[list[i] : list[i+1] - 1]$.

DELAUNAY

Performs a Delaunay triangulation and returns the vertex indices of the resulting polyhedra; otherwise, the convex hull of the data are returned.

SPHERE

Computes the Delaunay triangulation of the points which lie on the surface of a sphere. The *V0* argument contains the longitude, in degrees, and *V1* contains the latitude, in degrees, of each point.

VDIAGRAM

When specified, this keyword returns the connectivity of the Voronoi diagram.

For two-dimensional arrays, VDIAGRAM is a 4-by- nv integer array. For each Voronoi ridge, i , VDIAGRAM[0:1, i] contains the index of the two input points the ridge bisects. VDIAGRAM [2:3, i] contains the indices within VVERTICES of the

Voronoi vertices. In the case of an unbounded half-space, `VDIAGRAM[2, i]` is set to a negative index, j , indicating that the corresponding Voronoi ridge is unbounded, and that the equation for the ridge is contained in `VNORMAL[* , -j-1]`, and starts at Voronoi vertex `[3, i]`.

For three-dimensional or higher dimensional arrays, `VDIAGRAM` is returned as a connectivity vector. This vector has the form `[n, v0, v1, i0, i1, ..., in-3]`, where n is the number of points needed to describe that particular Voronoi ridge, `v0` and `v1` contain the indices for the two input points that the ridge bisects, and `i0...in-3` contain the indices within `VVERTICES` of the Voronoi vertices. In the case of an unbounded half-space, `VDIAGRAM[i]` is set to a negative index, j , indicating that the corresponding Voronoi ridge is unbounded, and that the equation for the ridge is contained in `VNORMAL[* , -j-1]`.

VNORMALS

When specified, this keyword returns the normals of each Voronoi ridge that is unbounded. The normals consist of a $(nd+1)$ -by- nu array, where nd is the number of dimensions, and nu is the number of unbounded vertices. Each row contains the equation for the unbounded ridge in the form:

$$V_0X_0 + V_1X_1 + V_2X_2 + \dots + V_ndX_nd + V_nd+1 = 0$$

where V_* are the elements of the row within `VNORMALS`, and X_* are the multidimensional coordinates. See the preceding description of `VDIAGRAM` for details.

VVERTICES

When specified, this keyword returns the Voronoi vertices.

Examples

```
PRO ex_qhull

; Create a collection of random points.
n = 20
seed = 15
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Construct the Delaunay triangulation
; and the Voronoi diagram.
QHULL, x, y, triangle, /DELAUNAY, $
      VDIAGRAM=vdiagram, VVERTICES=vvert, VNORM=vnorm
```

```

; Plot our input points.
PLOT, [-0.1, 1.1], [-0.1, 1.1], /NODATA, $
    XSTYLE=4, YSTYLE=4
PLOTS, x, y, PSYM=4

; Plot the Voronoi diagram.
FOR i=0,N_ELEMENTS(vdiagram[2,*])-1 DO BEGIN
    vdiag = vdiagram[*, i]
    j = vdiag[2] + 1
    ; Bounded or unbounded?
    IF (j gt 0) THEN BEGIN ; Bounded.
        PLOTS, vvert[*, vdiag[2:3]], PSYM=-5
        CONTINUE
    ENDIF

    ; Unbounded, retrieve starting vertex.
    xystart = vvert[*, vdiag[3]]
    ; Determine the line equation.
    ; Vnorm[0]*x + Vnorm[1]*y + Vnorm[2] = 0
    slope = -vnorm[0,-j]/vnorm[1,-j]
    intercept = -vnorm[2,-j]/vnorm[1,-j]

    ; Need to determine the line direction.
    ; Pick a point on one side along the line.
    xunbound = xystart[0] + 5
    yunbound = slope*xunbound + intercept

    ; Find the closest original vertex.
    void = MIN( (x-xunbound)^2 + (y-yunbound)^2, idx)
    ; By definition of Voronoi diagram, the line should
    ; be closest to one of the bisected points. If not,
    ; our line went in the wrong direction.
    IF (idx ne vdiag[0] && idx ne vdiag[1]) THEN BEGIN
        xunbound = xystart[0] - 5
        yunbound = slope*xunbound + intercept
    ENDIF

    PLOTS, [[xystart], [xunbound, yunbound]]

ENDFOR

END

```

For some other examples using the QHULL procedure, see the [QGRID3](#) function.

Version History

Introduced: 5.5

See Also

[QGRID3](#)

QROMB

The QROMB function evaluates the integral of a function over the closed interval $[A, B]$ using Romberg integration.

QROMB is based on the routine `qromb` described in section 4.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = QROMB( Func, A, B [, /DOUBLE] [, EPS=value] [, JMAX=value]
              [, K=value] )
```

Return Value

The result will have the same structure as the smaller of A and B , and the resulting type will be single- or double-precision floating, depending on the input types.

Arguments

Func

A scalar string specifying the name of a user-supplied IDL function to be integrated. This function must accept a single scalar argument X and return a scalar result. It must be defined over the closed interval $[A, B]$.

For example, if we wish to integrate the cubic polynomial

$$y = x^3 + (x - 1)^2 + 3$$

we define a function CUBIC to express this relationship in the IDL language:

```
FUNCTION cubic, X
  RETURN, X^3 + (X - 1.0)^2 + 3.0
END
```

A

The lower limit of the integration. A can be either a scalar or an array.

B

The upper limit of the integration. B can be either a scalar or an array.

Note

If arrays are specified for A and B , then QROMB integrates the user-supplied function over the interval $[A_i, B_i]$ for each i . If either A or B is a scalar and the other an array, the scalar is paired with each array element in turn.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EPS

The desired fractional accuracy. For single-precision calculations, the default value is 1.0×10^{-6} . For double-precision calculations, the default value is 1.0×10^{-12} .

JMAX

$2^{(JMAX - 1)}$ is the maximum allowed number of steps. If this keyword is not specified, a default of 20 is used.

K

Integration is performed by Romberg's method of order $2K$. If not specified, the default is $K=5$. ($K=2$ is Simpson's rule).

Examples

Example 1

To integrate the CUBIC function (listed above) over the interval $[0, 3]$ and print the result:

```
PRINT, QROMB('cubic', 0.0, 3.0)
```

IDL prints:

```
32.2500
```

This is the exact solution.

Example 2

This example evaluates the volume under a surface using the following double integration:

$$\text{volume} = \int_0^1 \int_0^1 (9x^2y^2 + 4xy + 1) dx dy$$

The exact solution to this equation is 3.

The example consists of four routines: the main routine, the integration in the y direction, the second integration of the x coefficient, and the second integration of the x^2 coefficient. The main routine is the last routine in the program. To run this example, copy the text of all four routines, paste them into an IDL editor window, and save the window's contents as `DoubleIntegration.pro`.

```

FUNCTION XSquaredCoef, x

    ; Integration of the x squared coefficient.
    secondIntegration = 9.*x^2
    RETURN, secondIntegration

END

FUNCTION XCoef, x

    ; Integration of the linear x coefficient.
    secondIntegration = x
    RETURN, secondIntegration

END

FUNCTION YDirection, y

    ; Re-write equation to consider both x coefficients.
    firstIntegration = QROMB('XSquaredCoef', 0., 1.)*y^2 $
    + 4.*(QROMB('XCoef', 0., 1.))*y + 1.
    RETURN, firstIntegration

END

PRO DoubleIntegration

    ; Determine the volume under the surface represented
    ; by 9x^2y^2 + 4xy + 1 over a specific region.
    volume = QROMB('YDirection', 0., 1. )

```

```

; Output results.
PRINT, 'Resulting Volume: ', volume

END

```

Example 3

This example evaluates the mass of a volume using the following triple integration on a three-dimensional equation representing its density:

$$\text{mass} = \int_0^1 \int_0^1 \int_0^1 (9x^2y^2 + 8xyz + 1) dx dy dz$$

The exact solution to this equation is 3.

The example consists of six routines: the main routine, the integration in the z-direction, the second integration of the xy coefficient, the second integration of the second x^2y^2 coefficient, the third integration in the x coefficient, and the third integration in the x^2 coefficient. The main routine is the last routine in the program. To run this example, copy the text of all six routines, paste them into an IDL editor window, and save the window's contents as `TripleIntegration.pro`.

```

FUNCTION XSquaredCoef, x

; Integration of the x squared coefficient.
thirdIntegration = 9.*x^2
RETURN, thirdIntegration

END

FUNCTION XCoef, x

; Integration of the linear x coefficient.
thirdIntegration = x
RETURN, thirdIntegration

END

FUNCTION XSquaredYSquaredCoef, y

; Integration of the y squared coefficient.
secondIntegration = QROMB('XSquaredCoef', 0., 1.)*y^2
RETURN, secondIntegration

END

```

```

FUNCTION XYCoef, y
    ; Integration of the linear y coefficient.
    secondIntegration = QROMB('XCoef', 0., 1.)*y
    RETURN, secondIntegration

END

FUNCTION ZDirection, z
    ; Re-write equation to consider all the x and y
    ; coefficients.
    firstIntegration = QROMB('XSquaredYSquaredCoef', 0., 1.) + $
    8.*(QROMB('XYCoef', 0., 1.))*z + 1.
    RETURN, firstIntegration

END

PRO TripleIntegration
    ; Determine the mass of the density represented
    ; by  $9x^2y^2 + 8xyz + 1$  over a specific region.
    mass = QROMB('ZDirection', 0., 1. )

    ; Output results.
    PRINT, 'Resulting Mass: ', mass

END

```

Version History

Introduced: 4.0

See Also

[INT_2D](#), [INT_3D](#), [INT_TABULATED](#), [QROMO](#), [QSIMP](#)

QROMO

The QROMO function evaluates the integral of a function over the open interval (A , B) using a modified Romberg's method.

QROMO is based on the routine `qromo` described in section 4.4 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = QROMO(Func, A [, B] [, /DOUBLE] [, EPS=value] [, JMAX=value]
           [, K=value] [, /MIDEXP | /MIDINF | /MIDPNT | /MIDSQL | /MIDSQU ] )
```

Return Value

Returns the integral of the function.

Arguments

Func

A scalar string specifying the name of a user-supplied IDL function to be integrated. This function must accept a single scalar argument X and return a scalar result. It must be defined over the open interval (A , B).

For example, if we wish to integrate the fourth-order polynomial

$$y = 1 / x^4$$

we define a function `HYPHER` to express this relationship in the IDL language:

```
FUNCTION hyper, X
    RETURN, 1.0 / X^4
END
```

A

The lower limit of the integration. A can be either a scalar or an array.

B

The upper limit of the integration. B can be either a scalar or an array. If the `MIDEXP` keyword is specified, B is assumed to be infinite, and should not be supplied by the user.

Note: If arrays are specified for A and B , then QROMO integrates the user-supplied function over the interval $[A_i, B_i]$ for each i . If either A or B is a scalar and the other an array, the scalar is paired with each array element in turn.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EPS

The fractional accuracy desired, as determined by the extrapolation error estimate. For single-precision calculations, the default value is 1.0×10^{-6} . For double-precision calculations, the default value is 1.0×10^{-12} .

JMAX

Set to specify the maximum allowed number of mid quadrature points to be $3^{(JMAX-1)}$. The default value is 14.

K

Integration is performed by Romberg's method of order $2K$. If not specified, the default is $K=5$.

MIDEXP

Use the `midexp()` function (see *Numerical Recipes*, section 4.4) as the integrating function. If the MIDEXP keyword is specified, argument B is assumed to be infinite, and should not be supplied by the user.

MIDINF

Use the `midinf()` function (see *Numerical Recipes*, section 4.4) as the integrating function.

MIDPNT

Use the `midpnt()` function (see *Numerical Recipes*, section 4.4) as the integrating function. This is the default if no other integrating function keyword is specified.

MIDSQL

Use the `midsql()` function (see *Numerical Recipes*, section 4.4) as the integrating function.

MIDSQU

Use the `midsqu()` function (see *Numerical Recipes*, section 4.4) as the integrating function.

Examples

Consider the following function:

```
FUNCTION hyper, X
    RETURN, 1.0 / X^4
END
```

This example integrates the `HYPER` function over the open interval $(2, \infty)$ and prints the result:

```
PRINT, QROMO('hyper', 2.0, /MIDEXP)
```

IDL prints:

```
0.0412050
```

Warning

When using the `MIDEXP` keyword, the upper integration limit is assumed to be infinity and is not supplied.

Version History

Introduced: 4.0

See Also

[INT_2D](#), [INT_3D](#), [INT_TABULATED](#), [QROMB](#), [QSIMP](#)

QSIMP

The QSIMP function performs numerical integration of a function over the closed interval $[A, B]$ using Simpson's rule.

QSIMP is based on the routine `qsimp` described in section 4.2 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

Result = QSIMP(*Func*, *A*, *B* [, /DOUBLE] [, EPS=*value*] [, JMAX=*value*])

Return Value

The result will have the same structure as the smaller of *A* and *B*, and the resulting type will be single- or double-precision floating, depending on the input types.

Arguments

Func

A scalar string specifying the name of a user-supplied IDL function to be integrated. This function must accept a single scalar argument *X* and return a scalar result. It must be defined over the closed interval $[A, B]$.

For example, if we wish to integrate the fourth-order polynomial

$$y = (x^4 - 2x^2) \sin(x)$$

we define a function SIMPSON to express this relationship in the IDL language:

```
FUNCTION simpson, X
    RETURN, (X^4 - 2.0 * X^2) * SIN(X)
END
```

A

The lower limit of the integration. *A* can be either a scalar or an array.

B

The upper limit of the integration. *B* can be either a scalar or an array.

Note

If arrays are specified for A and B , then QSIMP integrates the user-supplied function over the interval $[A_i, B_i]$ for each i . If either A or B is a scalar and the other an array, the scalar is paired with each array element in turn.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

EPS

The desired fractional accuracy. For single-precision calculations, the default value is 1.0×10^{-6} . For double-precision calculations, the default value is 1.0×10^{-12} .

JMAX

$2^{(JMAX - 1)}$ is the maximum allowed number of steps. If not specified, a default of 20 is used.

Examples

To integrate the SIMPSON function (listed above) over the interval $[0, \pi/2]$ and print the result:

```
; Define lower limit of integration:
A = 0.0

; Define upper limit of integration:
B = !PI/2.0

PRINT, QSIMP('simpson', A, B)
```

IDL prints:

```
-0.479158
```

The exact solution can be found using the integration-by-parts formula:

```
FB = 4.*B*(B^2-7.)*SIN(B) - (B^4-14.*B^2+28.)*COS(B)
FA = 4.*A*(A^2-7.)*SIN(A) - (A^4-14.*A^2+28.)*COS(A)
exact = FB - FA
PRINT, exact
```

IDL prints:

-0.479156

Version History

Introduced: 4.0

See Also

[INT_2D](#), [INT_3D](#), [INT_TABULATED](#), [QROMB](#), [QROMO](#)

QUERY_* Routines

Query routines allow users to obtain information about an image file without having to read the file. The following QUERY_* routines are available in IDL:

- [QUERY_BMP](#)
- [QUERY_DICOM](#)
- [QUERY_GIF](#)
- [QUERY_IMAGE](#)
- [QUERY_JPEG](#)
- [QUERY_MRSID](#)
- [QUERY_PICT](#)
- [QUERY_PNG](#)
- [QUERY_PPM](#)
- [QUERY_SRF](#)
- [QUERY_TIFF](#)
- [QUERY_WAV](#)

All of the QUERY_* routines return a result, which is a long with the value of 1 if the query was successful (and the file type was correct) or 0 on failure. If the query was successful, the return argument will be an anonymous structure containing all of the available information for that image format.

The status is intended to be used to determine if it is appropriate to use the corresponding READ_ routine for a given file. The return status of the QUERY_* will indicate success if the corresponding READ_ routine is likely to be able to read the file. The return status will indicate failure for cases that contain formats that are not supported by the READ_ routines, even though the files may be valid outside of the IDL environment. For example, IDL's READ_BMP does not support 1-bit-deep images and so the QUERY_BMP function would return failure in the case of a monochrome BMP file.

The returned anonymous structure will have (minimally) the following fields for all file formats. If the file does not support multiple images in a single file, the NUM_IMAGES field will always be 1 and the IMAGE_INDEX field will always be 0. Individual routines document additional fields which are returned for a specific format.

General Query * Routine Info Structures

Field	IDL data type	Description
CHANNELS	Long	Number of samples per pixel
DIMENSIONS	2-D long array	Size of the image in pixels
HAS_PALETTE	Integer	True if a palette is present
NUM_IMAGES	Long	Number of images in the file
IMAGE_INDEX	Long	Image number for which this structure is valid
PIXEL_TYPE	Integer	IDL basic type code for a pixel sample
TYPE	String	String identifying the file format

Table 3-96: Query Routines Info Structure

All the routines accept the `IMAGE_INDEX` keyword although formats which do not support multiple images in a single file will ignore this keyword.

QUERY_TIFF-Specific Routine Info Structures

Field	IDL data type	Description
<code>BITS_PER_SAMPLE</code>	Long	Number of bits per channel. Possible values are 1, 4, 8, 16, or 32.
<code>DESCRIPTION</code>	String	The contents of the TIFF ImageDescription tag, or a null string if undefined.
<code>DOCUMENT_NAME</code>	String	The contents of the TIFF DocumentName tag, or a null string if undefined.
<code>DATE_TIME</code>	String	The contents of the TIFF DateTime tag, or a null string if undefined.

Table 3-97: QUERY_TIFF Routine Info Structure

Field	IDL data type	Description
ORIENTATION	Long	Image orientation (columns, rows): <ul style="list-style-type: none"> • 1 = Left to right, top to bottom (default) • 2 = Right to left, top to bottom • 3 = Right to left, bottom to top • 4 = Left to right, bottom to top • 5 = Top to bottom, left to right • 6 = Top to bottom, right to left • 7 = Bottom to top, right to left • 8 = Bottom to top, left to right
PLANAR_CONFIG	Long	How the components of each pixel are stored. Possible values are: <ul style="list-style-type: none"> • 1 = Pixel interleaved RGB image or a two-dimensional image (no interleaving exists). Pixel components (such as RGB) are stored contiguously. • 2 = Image interleaved. Pixel components are stored in separate planes.
POSITION	Float array	Two-element vector [X position, Y position] containing the X and Y offsets in the same units as reported in the UNITS field. The X position is relative to the left side of the image. The Y position is relative to the top of the image, with positive Y increasing downwards. If the XPosition or YPosition tag is not defined in the file then a value of 0.0 is returned for that element.

Table 3-97: QUERY_TIFF Routine Info Structure (Continued)

Field	IDL data type	Description
PHOTOMETRIC	Long	Color mode used for the image data. Possible values are: <ul style="list-style-type: none"> • 0 = White is zero • 1 = Black is zero • 2 = RGB color model • 3 = Palette color model • 4 = Transparency mask • 5 = Separated (usually CMYK)
RESOLUTION	Float array	Two-element vector [<i>X resolution</i> , <i>Y resolution</i>] containing the number of pixels per <i>unit</i> (as reported in the UNITS field).
UNITS	Long	Units of measurement for RESOLUTION: <ul style="list-style-type: none"> • 1 = No units • 2 = Inches (the default) • 3 = Centimeters For example, if the UNITS field contains the value 2, then the values in the RESOLUTION field represent pixels per inch.
TILE_SIZE	Long array	Two-element vector [<i>Tile width</i> , <i>Tile height</i>]. Non-tiled images will contain [<i>Image width</i> , 1]

Table 3-97: QUERY_TIFF Routine Info Structure (Continued)

QUERY_BMP

QUERY_BMP is a method of obtaining information about a BMP image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Syntax

```
Result = QUERY_BMP ( Filename [, Info] )
```

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the BMP file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value ‘BMP’.

Note

See “[General Query * Routine Info Structures](#)” on page 1652 for detailed structure information.

Keywords

There are no keywords for this routine.

Version History

Introduced: 5.2

See Also

[QUERY_* Routines](#), [READ_BMP](#), [WRITE_BMP](#)

QUERY_DICOM

The QUERY_DICOM function tests a file for compatibility with READ_DICOM and returns an optional structure containing information about images in the DICOM file. This function supports cases in which a blank DICOM tag is supplied.

This routine is written in the IDL language. Its source code can be found in the file `query_dicom.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = QUERY_DICOM( Filename [, Info] [, IMAGE_INDEX=index] )
```

Return Value

This routine returns a long with the value of 1 (one) if the query was successful or 0 (zero) on failure. A result of 1 means it is likely that the file can be read by READ_DICOM.

Arguments

Filename

A scalar string containing the full pathname of the file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value 'DICOM'.

Note

See [“General Query * Routine Info Structures”](#) on page 1652 for detailed structure information.

Keywords

IMAGE_INDEX

Set this keyword to the index (zero based) of the image being queried in the file. This keyword has no effect on files containing a single image.

Examples

DICOM palette vectors are 16 bit quantities and may not cover the entire dynamic range of the image. To view a paletted DICOM image use the following:

```
IF (QUERY_DICOM('file.dcm',info)) THEN BEGIN
  IF (info.has_palette) THEN BEGIN
    TV, READ_IMAGE('file.dcm',r, g, b), /ORDER
    TVLCT,r/256, g/256, b/256
  ENDIF
ENDIF
```

Version History

Introduced: 5.2

See Also

[READ_DICOM](#)

QUERY_GIF

QUERY_GIF is a method of obtaining information about a GIF image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Note

The GIF format uses compression technologies that may be subject to license restrictions. If you receive a warning when attempting to use this routine, contact RSI for information about licensing options.

Syntax

Result = QUERY_GIF (*Filename* [, *Info*])

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the GIF file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value ‘GIF’.

Note

See “[QUERY_* Routines](#)” on page 1651 for detailed structure info.

Keywords

None

See Also

[QUERY_* Routines](#), [READ_GIF](#), [WRITE_GIF](#)

QUERY_IMAGE

The QUERY_IMAGE function determines whether a file is recognized as a supported image file. QUERY_IMAGE first checks the filename suffix, and if found, calls the corresponding QUERY_ routine. For example, if the specified file is image.bmp, QUERY_BMP is called to determine if the file is a valid .bmp file. If the file does not contain a filename suffix, or if the query fails on the specified filename suffix, QUERY_IMAGE checks against all supported file types.

Syntax

```
Result = QUERY_IMAGE ( Filename[, Info] [, CHANNELS=variable]
  [, DIMENSIONS=variable] [, HAS_PALETTE=variable]
  [, IMAGE_INDEX=index] [, NUM_IMAGES=variable]
  [, PIXEL_TYPE=variable] [, SUPPORTED_READ=variable]
  [, SUPPORTED_WRITE=variable] [, TYPE=variable] )
```

Return Value

Result is a long with the value of 1 if the query was successful (the file was recognized as an image file) or 0 on failure. The return status will indicate failure for files that contain formats that are not supported by the corresponding READ_* routine, even though the file may be valid outside the IDL environment.

If the file is a supported image file, an optional structure containing information about the image is returned. If the file is not a supported image file, QUERY_IMAGE returns 0.

Arguments

Filename

A scalar string containing the name of the file to query.

Info

An optional anonymous structure containing information about the image. This structure is valid only when the return value of the function is 1. The Info structure for all image types has the following fields:

Tag	Type
CHANNELS	Long
DIMENSIONS	Two-dimensional long array
FILENAME	Scalar string
HAS_PALETTE	Integer
IMAGE_INDEX	Long
NUM_IMAGES	Long
PIXEL_TYPE	Integer
TYPE	Scalar string

Table 3-98: The Info Structure for All Image Types

Keywords

CHANNELS

Set this keyword to a named variable to retrieve the number of channels in the image.

DIMENSIONS

Set this keyword to a named variable to retrieve the image dimensions as a two-dimensional array.

HAS_PALETTE

Set this keyword to a named variable to equal to 1 if a palette is present, else 0.

IMAGE_INDEX

Set this keyword to the index of the image to query from the file. The default is 0, the first image.

NUM_IMAGES

Set this keyword to a named variable to retrieve the number of images in the file.

PIXEL_TYPE

Set this keyword to a named variable to retrieve the IDL Type Code of the image pixel format. See the documentation for the `SIZE` routine for a complete list of IDL Type Codes.

The valid types for `PIXEL_TYPE` are:

- 1 = Byte
- 2 = Integer
- 3 = Longword Integer
- 4 = Floating Point
- 5 = Double-precision Floating Point
- 12 = Unsigned Integer
- 13 = Unsigned Longword Integer
- 14 = 64-bit Integer
- 15 = Unsigned 64-bit Integer

SUPPORTED_READ

Set this keyword to a named variable to retrieve a string array of image types recognized by `READ_IMAGE`. If the `SUPPORTED_READ` keyword is used the filename and info arguments are optional.

SUPPORTED_WRITE

Set this keyword to a named variable to retrieve a string array of image types recognized by `WRITE_IMAGE`. If the `SUPPORTED_WRITE` keyword is used the filename and info arguments are optional.

TYPE

Set this keyword to a named variable that will contain the image type as a scalar string. Possible return values are `BMP`, `GIF`, `JPEG`, `JPEG2000`, `PNG`, `PPM`, `SRF`, `TIFF`, or `DICOM`.

Version History

Introduced: 5.3

QUERY_JPEG

QUERY_JPG is a method of obtaining information about a JPEG image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Syntax

Result = QUERY_JPEG (*Filename* [, *Info*])

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the JPEG file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value ‘JPEG’.

Note

See “[General Query * Routine Info Structures](#)” on page 1652 for detailed structure information.

Keywords

None.

Version History

Introduced: 5.2

See Also

[QUERY_* Routines](#), [READ_JPEG](#), [WRITE_JPEG](#)

QUERY_JPEG2000

The QUERY_JPEG2000 function allows you to obtain information about a JPEG2000 image file without having to read the file. It is a wrapper around the IDLffJPEG2000 object interface that presents JPEG2000 image loading in a way familiar to users of the QUERY_* image routines. (See “[QUERY_* Routines](#)” on page 1651 for more information.) However this function is not as efficient as the object interface and the object interface should be used whenever possible. See “[IDLffJPEG2000](#)” in Chapter 6 of the *IDL Reference Guide* manual for information about the object interface.

Syntax

Result = QUERY_JPEG2000(*Filename* [, *Info*])

Return Value

This routine returns a long integer with the value of 1 if the query was successful (and the file type was correct) or 0 on failure.

Arguments

Filename

A scalar string containing the full path and filename of the JPEG2000 file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value 'JPEG2000'.

The anonymous structure is detailed in the QUERY_* Routines documentation. In addition to the standard structure fields, the JPEG2000 query structure contains the following additional fields:

Field	IDL Data Type	Description
N_LAYERS	Long	The number of quality layers.

Table 6: JPEG2000 Query Structure

Field	IDL Data Type	Description
N_LEVELS	Long	The number of wavelet decomposition levels.
OFFSET	Two-element vector	The X and Y offsets for all image components.

Table 6: JPEG2000 Query Structure

Keywords

None.

Example

For a relevant example, see [READ_JPEG2000](#).

Version History

Introduced: 6.1

See Also

[QUERY_* Routines](#), [READ_JPEG2000](#), [WRITE_JPEG2000](#)

QUERY_MRSID

The QUERY_MRSID function allows you to obtain information about a MrSID image file without having to read the file. It is a wrapper around the object interface that presents MrSID image loading in a familiar way to users of the QUERY_* image routines. (See “[QUERY_* Routines](#)” on page 1651 for more information.) However this function is not as efficient as the object interface and the object interface should be used whenever possible. See “[IDLffMrSID](#)” on page 2835 for information about the object interface.

Note

This routine is only available under Microsoft Windows.

Syntax

```
Result = QUERY_MRSID( Filename [, Info] [, LEVEL=lvl] )
```

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string or one element by one element array containing the full path and filename of the MrSID file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value 'MrSID'.

The anonymous structure is detailed in the QUERY_* Routines documentation. However, the info structure filled in by QUERY_MRSID has additional members appended to the end:

- info.LEVELS – a named variable that will contain a two-element integer vector of the form [minlvl, maxlvl] that specifies the range of levels within the current image. Higher levels are lower resolution. A level of 0 equals full resolution. Negative values specify higher levels of resolution.

- `Info.GEO_VALID` – a long integer with a value of 1 if the file contains valid georeferencing data, or 0 if the georeferencing data is nonexistent or unsupported.

Note

Always verify that this keyword returns 1 before using the data returned by any other `GEO_*` keyword.

- `Info.GEO_PROJTYPE` – unsigned integer.
- `Info.GEO_ORIGIN` – 2-element double precision array.
- `Info.GEO_RESOLUTION` – 2-element double precision array.

See [“IDLffMrSID::GetProperty”](#) on page 2844 for more information on `GEO_*` values.

Keywords

LEVEL

Set this keyword to an integer that specifies the level to which the `DIMENSIONS` field of the info structure corresponds. This can be used, for example, to determine what level is required to fit the image into a certain area. If this keyword is not specified, the dimensions at level 0 are returned.

Examples

```
; Select the image file.
file = QUERY_MRSID(FILEPATH('test_gs.sid', $
    SUBDIRECTORY=['examples', 'data']), info, LEVEL = -2)

HELP, file
; IDL returns 1 indicating the correct file type
; and successful query.

; Print the range of levels of resolution available within
; the file.
PRINT, 'Range of image levels = ', info.LEVELS

; Print the image dimensions when the image level is set to -2
; as specified by LEVEL = -2 in the QUERY_MRSID statement.
PRINT, 'dimensions of image at LEVEL is -2 =', info.DIMENSIONS
; IDL returns 2048 by 2048

; Check for valid georeferencing data.
```

```
PRINT, 'Result of georeferencing query', info.GEO_VALID  
; IDL returns 0 indicating that the file does not contain  
; georeferencing data.
```

Version History

Introduced: 5.5

QUERY_PICT

QUERY_PICT is a method of obtaining information about a PICT image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Syntax

Result = QUERY_PICT (*Filename* [, *Info*])

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the PICT file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value 'PICT'.

Note

See “[General Query * Routine Info Structures](#)” on page 1652 for detailed structure information.

Keywords

None

Version History

Introduced: 5.2

See Also

[QUERY_* Routines](#), [READ_PICT](#), [WRITE_PICT](#)

QUERY_PNG

QUERY_PNG is a method of obtaining information about a PNG image file without having to read the file. See [“QUERY_* Routines”](#) on page 1651 for more information.

Syntax

```
Result = QUERY_PNG ( Filename [, Info] )
```

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the PNG file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value 'PNG'.

Note

See [“General Query * Routine Info Structures”](#) on page 1652 for detailed structure information.

Keywords

None

Examples

Query included in creating RGBA (16-bit/channel) and Color Indexed (8-bits/channel) image.

```
rgbdata = UINDGEN(4,320,240)
cidata = BYTSCL(DIST(256))
red = indgen(256)
```

```

green = indgen(256)
blue = indgen(256)
WRITE_PNG, 'rgb_image.png', rgbdata
WRITE_PNG, 'ci_image.png', cidata, red, green, blue

; Query and Read the data:
names = ['rgb_image.png', 'ci_image.png', 'unknown.png']

FOR i=0,N_ELEMENTS(names)-1 DO BEGIN
  ok = QUERY_PNG(names[i],s)
  IF (ok) THEN BEGIN
    HELP,s,/STRUCTURE
    IF (s.HAS_PALETTE) THEN BEGIN
      img = READ_PNG(names[i],rpal,gpal,bpal)
      HELP,img,rpal,gpal,bpal
    ENDIF ELSE BEGIN
      img = READ_PNG(names[i])
      HELP,img
    ENDELSE
  ENDIF ELSE BEGIN
    PRINT,names[i],' is not a PNG file'
  ENDELSE
ENDFOR
END

```

Version History

Introduced: 5.2

See Also

[QUERY_* Routines](#), [READ_PNG](#), [WRITE_PNG](#)

QUERY_PPM

QUERY_PPM is a method of obtaining information about a PPM image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Syntax

```
Result = QUERY_PPM ( Filename [, Info] [, MAXVAL=variable] )
```

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the PPM file to query.

Info

Returns an anonymous structure containing information about the image. The Info.TYPE field will return the value ‘PPM’.

Additional field in the Info structure: MAXVAL - maximum pixel value in the image.

Note

See “[General Query * Routine Info Structures](#)” on page 1652 for detailed structure information.

Keywords

MAXVAL

Set this keyword to a named variable to retrieve the maximum pixel value in the image.

Version History

Introduced: 5.2

See Also

[QUERY_* Routines](#), [READ_PPM](#), [WRITE_PPM](#)

QUERY_SRF

QUERY_SRF is a method of obtaining information about an SRF image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Syntax

Result = QUERY_SRF (*Filename* [, *Info*])

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the SRF file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value ‘SRF’.

Note

See “[General Query * Routine Info Structures](#)” on page 1652 for detailed structure information.

Keywords

None.

Version History

Introduced: 5.2

See Also

[QUERY_* Routines](#), [READ_SRF](#), [WRITE_SRF](#)

QUERY_TIFF

QUERY_TIFF is a method of obtaining information about a TIFF image file without having to read the file. See “[QUERY_* Routines](#)” on page 1651 for more information.

Syntax

```
Result = QUERY_TIFF ( Filename [, Info] [, GEOTIFF=variable]
                    [, IMAGE_INDEX=index] )
```

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the pathname of the TIFF file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value ‘TIFF’.

Note

In addition to the fields returned in the Info structure for all image types, there are a number of extra fields in the Info structure for TIFF images. See “[QUERY_TIFF-Specific Routine Info Structures](#)” on page 1652 for detailed structure info.

Keywords

GEOTIFF

Set this keyword to a named variable in which to return an anonymous structure containing the GeoTIFF tags and keys found in the file. If no GeoTIFF information is present in the file, the returned variable is set to a scalar zero. For a description of the possible GeoTIFF tags see the GEOTIFF keyword in READ_TIFF.

Tip

This keyword returns the same information as the GEOTIFF keyword to READ_TIFF, but avoids the overhead of reading in the entire image.

IMAGE_INDEX

Image number index. If this value is larger than the number of images in the file, the function returns 0 (failure).

Examples

This is an example of using QUERY_TIFF to write and read a multi-image TIFF file. The first image is a 16-bit, single channel image stored using compression. The second image is an RGB image stored using 32-bits/channel uncompressed.

```

; Write the image data:
data = FIX(DIST(256))
rgbdata = LONARR(3,320,240)
WRITE_TIFF, 'multi.tif', data, COMPRESSION=1, /SHORT
WRITE_TIFF, 'multi.tif', rgbdata, /LONG, /APPEND

; Read the image data back:
ok = QUERY_TIFF('multi.tif', s)
IF (ok) THEN BEGIN
    FOR i=0, s.NUM_IMAGES-1 DO BEGIN
        imp = QUERY_TIFF('multi.tif', t, IMAGE_INDEX=i)
        img = READ_TIFF('multi.tif', IMAGE_INDEX=i)
        HELP, t, /STRUCTURE
        HELP, img
    ENDFOR
ENDIF

```

Version History

Introduced: 5.2

DESCRIPTION, DOCUMENT_NAME, DATE_TIME, and POSITION *Info* structures added: 6.1

GEOTIFF keyword added: 6.1

See Also

[QUERY_* Routines](#), [READ_TIFF](#), [WRITE_TIFF](#)

QUERY_WAV

The QUERY_WAV function checks that the file is actually a .WAV file and that the READ_WAV function can read the data in the file. Optionally, it can return additional information about the data in the file.

Syntax

Result = QUERY_WAV (*Filename* [, *Info*])

Return Value

This routine returns a long with the value of 1 (one) if the query was successful (and the file type was correct) or 0 (zero) on failure.

Arguments

Filename

A scalar string containing the full pathname of the .WAV file to read.

Info

An anonymous structure containing information about the data in the file. The fields are defined as:

Tag	Type	Definition
CHANNELS	INT	Number of data channels in the file.
SAMPLES_PER_SEC	LONG	Data sampling rate in samples per second.
BITS_PER_SAMPLE	INT	Number of valid bits in the data.

Table 3-99: The Info Structure for Info Fields

Keywords

None.

Version History

Introduced: 5.3

R_CORRELATE

The R_CORRELATE function computes Spearman's (rho) or Kendall's (tau) rank correlation of two sample populations X and Y . The result is a two-element vector containing the rank correlation coefficient and the two-sided significance of its deviation from zero. The significance is a value in the interval [0.0, 1.0]; a small value indicates a significant correlation.

$$\text{rho} = \frac{\sum_{i=0}^{N-1} (R_{x_i} - \bar{R}_x)(R_{y_i} - \bar{R}_y)}{\sqrt{\sum_{i=0}^{N-1} (R_{x_i} - \bar{R}_x)^2} \sqrt{\sum_{i=0}^{N-1} (R_{y_i} - \bar{R}_y)^2}}$$

where R_{x_i} and R_{y_i} are the magnitude-based ranks among X and Y , respectively. Elements of identical magnitude are ranked using a rank equal to the mean of the ranks that would otherwise be assigned.

This routine is written in the IDL language. Its source code can be found in the file `r_correlate.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = R_CORRELATE( X, Y [, D=variable] [, /KENDALL] [, PROBD=variable]
                    [, ZD=variable] )
```

Return Value

Returns a two-element vector indicating the rank correlation coefficient and the significance of its deviation from zero.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An n -element integer, single-, or double-precision floating-point vector.

Keywords

D

Set this keyword to a named variable that will contain the sum-squared difference of ranks. If the KENDALL keyword is set, this parameter is returned as zero.

KENDALL

Set this keyword to compute Kendall's (tau) rank correlation. By default, Spearman's (rho) rank correlation is computed.

PROBD

Set this keyword to a named variable that will contain the two-sided significance level of ZD. If the KENDALL keyword is set, this parameter is returned as zero.

ZD

Set this keyword to a named variable that will contain the number of standard deviations by which D deviates from its null-hypothesis expected value. If the KENDALL keyword is set, this parameter is returned as zero.

Examples

```

; Define two n-element sample populations:
X = [257, 208, 296, 324, 240, 246, 267, 311, 324, 323, 263, $
     305, 270, 260, 251, 275, 288, 242, 304, 267]
Y = [201, 56, 185, 221, 165, 161, 182, 239, 278, 243, 197, $
     271, 214, 216, 175, 192, 208, 150, 281, 196]

; Compute Spearman's (rho) rank correlation of X and Y.
result = R_CORRELATE(X, Y)
PRINT, 'Spearman's (rho) rank correlation: ', result

; Compute Kendall's (tau) rank correlation of X and Y:
result = R_CORRELATE(X, Y, /KENDALL)
PRINT, 'Kendall's (tau) rank correlation: ', result

```

IDL prints:

```

Spearman's (rho) rank correlation:    0.835967    4.42899e-006
Kendall's (tau) rank correlation:    0.624347    0.000118729

```

Version History

Introduced: 4.0

See Also

[A_CORRELATE](#), [C_CORRELATE](#), [CORRELATE](#), [M_CORRELATE](#),
[P_CORRELATE](#)

R_TEST

The `R_TEST` function tests the hypothesis that a binary population (a sequence of 1s and 0s) represents a “random sampling”.

This routine is written in the IDL language. Its source code can be found in the file `r_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = R_TEST( X [, NO=variable] [, N1=variable] [, R=variable] )
```

Return Value

The result is a two-element vector containing the nearly-normal test statistic Z and its associated probability. This two-tailed test is based on the “theory of runs” and is often referred to as the “Runs Test for Randomness.”

Arguments

X

An n -element integer, single-, or double-precision floating-point vector. Elements not equal to 0 or 1 are removed and the length of X is correspondingly reduced.

Keywords

NO

Set this keyword to a named variable that will contain the number of 0s in X .

N1

Set this keyword to a named variable that will contain the number of 1s in X .

R

Set this keyword to a named variable that will contain the number of runs (clusters of 0s and 1s) in X .

Examples

```

; Define a binary population:
X = [0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, $
     1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1]

; Test the hypothesis that X represents a random sampling against
; the hypothesis that it does not represent a random sampling at
; the 0.05 significance level:
result = R_TEST(X, R = r, N0 = n0, N1 = n1)
PRINT, result

```

IDL prints:

```
[2.26487, 0.0117604]
```

Print the values of the keyword parameters:

```

PRINT, 'Runs: ', r & PRINT, 'Zeros: ', n0 & PRINT, 'Ones: ', n1
Runs:      22
Zeros:    16
Ones:     14

```

The computed probability (0.0117604) is less than the 0.05 significance level and therefore we reject the hypothesis that X represents a random sampling. The results show that there are too many runs, indicating a non-random cyclical pattern.

Version History

Introduced: 4.0

See Also

[CTI_TEST](#), [FV_TEST](#), [KW_TEST](#), [LNP_TEST](#), [MD_TEST](#), [RS_TEST](#), [S_TEST](#), [TM_TEST](#), [XSQ_TEST](#)

RADON

The RADON function implements the Radon transform, used to detect features within a two-dimensional image. This function can be used to return either the Radon transform, which transforms lines through an image to points in the Radon domain, or the Radon backprojection, where each point in the Radon domain is transformed to a straight line in the image.

Radon Transform Theory

The Radon transform is used to detect features within an image. Given a function $A(x, y)$, the Radon transform is defined as:

$$R(\theta, \rho) = \int_{-\infty}^{\infty} A(\rho \cos \theta - s \sin \theta, \rho \sin \theta + s \cos \theta) ds$$

This equation describes the integral along a line s through the image, where ρ is the distance of the line from the origin and θ is the angle from the horizontal.

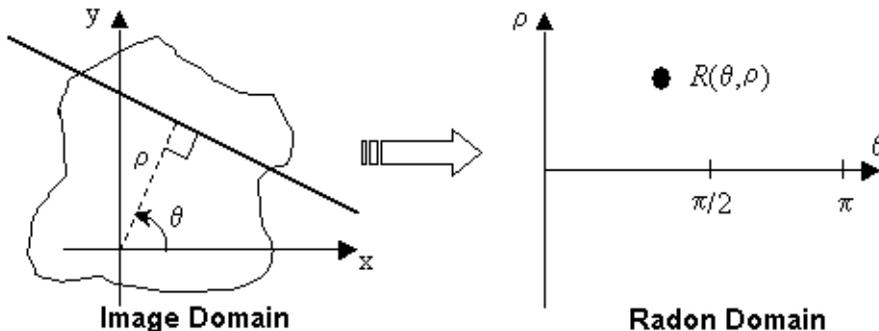


Figure 3-50: The Radon Transform

In medical imaging, each point $R(\theta, \rho)$ is called a ray-sum, while the resulting image is called a shadowgram. An image can be reconstructed from its ray-sums using the backprojection operator:

$$B(x, y) = \int_0^{\pi} R(\theta, x \cos \theta + y \sin \theta) d\theta$$

where the output, $B(x, y)$, is an image of $A(x, y)$ blurred by the Radon transform.

How IDL Implements the Radon Transform

To avoid the use of a two-dimensional interpolation and decrease the interpolation errors, the Radon transform equation is rotated by θ , and the interpolation is then done along the line s . The transform is divided into two regions, one for nearly-horizontal lines ($\theta \leq 45^\circ$; $135^\circ \leq \theta \leq 180^\circ$), and the other for steeper lines ($45^\circ < \theta < 135^\circ$), where θ is assumed to lie on the interval $[0^\circ, 180^\circ]$.

For nearest-neighbor interpolation (the default), the discrete transform formula for an image $A(m, n)$ [$m = 0, \dots, M-1, n = 0, \dots, N-1$] is:

$$R(\theta, \rho) = \begin{cases} \frac{\Delta x}{|\sin \theta|} \sum_m A(m, [am + b]) & |\sin \theta| > \frac{\sqrt{2}}{2} \\ \frac{\Delta y}{|\cos \theta|} \sum_n A([a'n + b'], n) & |\sin \theta| \leq \frac{\sqrt{2}}{2} \end{cases}$$

where brackets [] indicate rounding to the nearest integer, and the slope and offsets are given by:

$$a = -\frac{\Delta x \cos \theta}{\Delta y \sin \theta} \qquad b = \frac{\rho - x_{\min} \cos \theta - y_{\min} \sin \theta}{\Delta y \sin \theta}$$

$$a' = \frac{1}{a} \qquad b' = \frac{\rho - x_{\min} \cos \theta - y_{\min} \sin \theta}{\Delta x \cos \theta}$$

For linear interpolation, the transform is:

$$R(\theta, \rho) = \begin{cases} \frac{\Delta x}{|\sin \theta|} \sum_m (1-w)A(m, \lfloor am + b \rfloor) + wA(m, \lfloor am + b \rfloor + 1) & |\sin \theta| > \frac{\sqrt{2}}{2} \\ \frac{\Delta y}{|\cos \theta|} \sum_n (1-w)A(\lfloor a'n + b' \rfloor, n) + wA(\lfloor a'n + b' \rfloor + 1, n) & |\sin \theta| \leq \frac{\sqrt{2}}{2} \end{cases}$$

where the slope and offsets are the same as above, and $\lfloor \ \rfloor$ indicates flooring to the nearest lower integer. The weighting w is given by the difference between $am + b$ and its floored value, or between $a'n + b'$ and its floored value.

How IDL Implements the Radon Backprojection

For the backprojection transform, the discrete formula for nearest-neighbor interpolation is:

$$B(m,n) = \Delta\theta \sum_t R(\theta_t, \lfloor \rho \rfloor)$$

with the nearest-neighbor for ρ given by:

$$\rho = \{(m\Delta x + x_{\min}) \cos \theta_t + (n\Delta y + y_{\min}) \sin \theta_t - \rho_{\min}\} \Delta\rho^{-1}$$

For backprojection with linear interpolation:

$$B(m,n) = \Delta\theta \sum_t (1-w)R(\theta_t, \lfloor \rho \rfloor) + wR(\theta_t, \lfloor \rho \rfloor + 1)$$

$$w = \rho - \lfloor \rho \rfloor$$

and ρ is the same as in the nearest-neighbor.

Syntax

Radon Transform:

```
Result = RADON(Array [, /DOUBLE] [, DRHO=scalar] [, DX=scalar]
  [, DY=scalar] [, /GRAY] [, /LINEAR] [, NRHO=scalar] [, NTHETA=scalar]
  [, RHO=variable] [, RMIN=scalar] [, THETA=variable] [, XMIN=scalar]
  [, YMIN=scalar] )
```

Radon Backprojection:

```
Result = RADON(Array, /BACKPROJECT, RHO=variable, THETA=variable
  [, /DOUBLE] [, DX=scalar] [, DY=scalar] [, /LINEAR] [, NX=scalar]
  [, NY=scalar] [, XMIN=scalar] [, YMIN=scalar] )
```

Return Value

The result of this function is a two-dimensional floating-point array, or a complex array if the input image is complex. If *Array* is double-precision, or if the **DOUBLE** keyword is set, the result is double-precision, otherwise, the result is single-precision.

Arguments

Array

The two-dimensional array of size M by N to be transformed.

Keywords

BACKPROJECT

If set, the backprojection is computed, otherwise, the forward transform is computed.

Note

The Radon backprojection does not return the original image. Instead, it returns an image blurred by the Radon transform. Because the Radon transform is not one-to-one, multiple (x, y) points are mapped to a single (θ, ρ) .

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

DRHO

Set this keyword equal to a scalar specifying the spacing between ρ coordinates, expressed in the same units as *Array*. The default is one-half of the diagonal distance between pixels, $0.5[(DX^2 + DY^2)]^{1/2}$. Smaller values produce finer resolution, and are useful for zooming in on interesting features. Larger values may result in undersampling, and are not recommended. If **BACKPROJECT** is specified, this keyword is ignored.

DX

Set this keyword equal to a scalar specifying the spacing between the horizontal (x) coordinates. The default is 1.0.

DY

Set this keyword equal to a scalar specifying the spacing between the vertical (y) coordinates. The default is 1.0.

GRAY

Set or omit this keyword to perform a weighted Radon transform, with the weighting given by the pixel values. If GRAY is explicitly set to zero, the image is treated as a binary image with all nonzero pixels considered as 1.

LINEAR

Set this keyword to use linear interpolation rather than the default nearest-neighbor sampling. Results are more accurate but slower when linear interpolation is used.

NRHO

Set this keyword equal to a scalar specifying the number of ρ coordinates to use. The default is $2 \text{ CEIL}([\text{MAX}(x^2 + y^2)]^{1/2} / \text{DRHO}) + 1$. If BACKPROJECT is specified, this keyword is ignored.

NTHETA

Set this keyword equal to a scalar specifying the number of θ coordinates to use over the interval $[0, \pi]$. The default is $\text{CEIL}(\pi [(M^2 + N^2)/2]^{1/2})$. Larger values produce smoother results, and are useful for filtering before backprojection. Smaller values result in broken lines in the transform, and are not recommended. If BACKPROJECT is specified, this keyword is ignored.

NX

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of horizontal coordinates in the output *Result*. The default is $\text{FLOOR}(2 \text{ MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform this keyword is ignored.

NY

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of vertical coordinates in the output *Result*. The default is $\text{FLOOR}(2 \text{ MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform, this keyword is ignored.

RHO

For the forward transform, set this keyword to a named variable that will contain the radial (ρ) coordinates. If BACKPROJECT is specified, this keyword must contain the ρ coordinates of the input *Array*. The ρ coordinates should be evenly spaced and in increasing order.

RMIN

Set this keyword equal to a scalar specifying the minimum ρ coordinate to use for the forward transform. The default is $-0.5(\text{NRHO} - 1) \text{DRHO}$. If `BACKPROJECT` is specified, this keyword is ignored.

THETA

For the forward transform, set this keyword to a named variable containing a vector of angular (θ) coordinates to use for the transform. If `NTHETA` is specified instead, and `THETA` is set to a named variable, on exit `THETA` will contain the θ coordinates. If `BACKPROJECT` is specified, this keyword must contain the θ coordinates of the input *Array*.

XMIN

Set this keyword equal to a scalar specifying the x -coordinate of the lower-left corner of the input *Array*. The default is $-(M-1)/2$, where *Array* is an M by N array. If `BACKPROJECT` is specified, set this keyword equal to a scalar specifying the x -coordinate of the lower-left corner of the *Result*. In this case the default is $-DX (NX-1)/2$.

YMIN

Set this keyword equal to a scalar specifying the y -coordinate of the lower-left corner of the input *Array*. The default is $-(N-1)/2$, where *Array* is an M by N array. If `BACKPROJECT` is specified, set this keyword equal to a scalar specifying the y -coordinate of the lower-left corner of the *Result*. In this case, the default is $-DY (NY-1)/2$.

Examples

This example displays the Radon transform and the Radon backprojection:

```
PRO radon_example

    DEVICE, DECOMPOSED=0

    ;Create an image with a ring plus random noise:
    x = (LINDGEN(128,128) MOD 128) - 63.5
    y = (LINDGEN(128,128)/128) - 63.5
    radius = SQRT(x^2 + y^2)
    array = (radius GT 40) AND (radius LT 50)
    array = array + RANDOMU(seed,128,128)
```

```

;Create display window, set graphics properties:
WINDOW, XSIZE=440,YSIZE=700, TITLE='Radon Example'
!P.BACKGROUND = 255 ; white
!P.COLOR = 0 ; black
!P.FONT=2
ERASE

XYOUTS, .05, .94, 'Ring and Random Pixels', /NORMAL
;Display the image. 255b changes black values to white:
TVSCL, 255b - array, .05, .75, /NORMAL

;Calculate and display the Radon transform:
XYOUTS, .05, .70, 'Radon Transform', /NORMAL
result = RADON(array, RHO=rho, THETA=theta)
TVSCL, 255b - result, .08, .32, /NORMAL
PLOT, theta, rho, /NODATA, /NOERASE, $
    POSITION=[0.08,0.32, 1, 0.68], $
    XSTYLE=9,YSTYLE=9,XTITLE='Theta', YTITLE='R'

;For simplicity in this example, remove everything except
;the two stripes. A better (and more complicated) method would
;be to choose a threshold for the result at each value of theta,
;perhaps based on the average of the result over the theta
;dimension.
result[*,0:55] = 0
result[*,73:181] = 0
result[*,199:*] = 0

;Find the Radon backprojection and display the output:
XYOUTS, .05, .26, 'Radon Backprojection', /NORMAL
backproject = RADON(result, /BACKPROJECT, RHO=rho, THETA=theta)
TVSCL, 255b - backproject, .05, .07, /NORMAL

END

```

The following figure displays the program output. The top image is an image of a ring and random pixels, or noise. The center image is the Radon transform, and displays the line integrals through the image. The bottom image is the Radon backprojection, after filtering all noise except for the two strong horizontal stripes in the middle image.

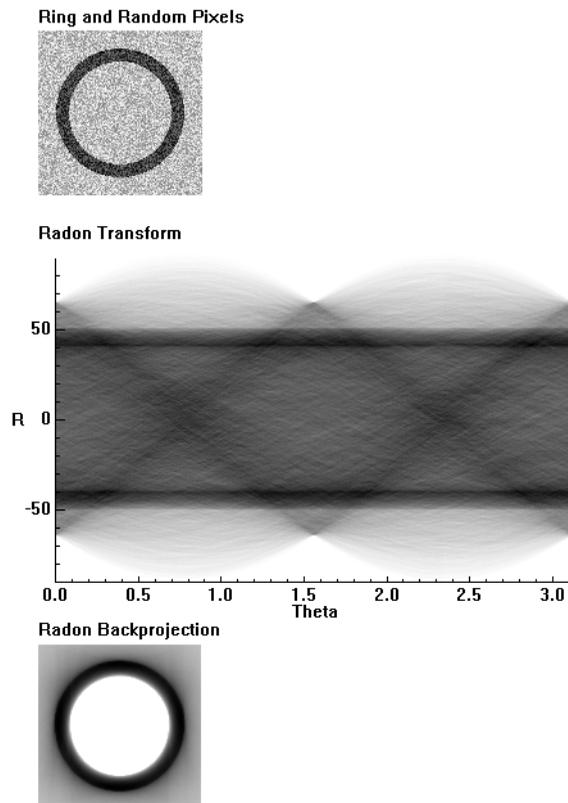


Figure 3-51: Radon Example - Original image (top), Radon transform (center), and backprojection of the altered Radon transform (bottom).

References

1. Herman, Gabor T. *Image Reconstruction from Projections*. New York: Academic Press, 1980.
2. Hiriyannaiah, H. P. X-ray computed tomography for medical imaging. *IEEE Signal Processing Magazine*, March 1997: 42-58.
3. Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

4. Toft, Peter. *The Radon Transform: Theory and Implementation*. Denmark: Technical University; 1996. Ph.D. Thesis.

Version History

Introduced: 5.4

See Also

[HOUGH, VOXEL_PROJ](#)

RANDOMN

The RANDOMN function returns one or more normally-distributed, floating-point, pseudo-random numbers with a mean of zero and a standard deviation of one. RANDOMN uses the Box-Muller method for generating normally-distributed (Gaussian) random numbers.

Syntax

```
Result = RANDOMN( Seed [, D1 [, ..., Dg]] [ [, BINOMIAL=[trials, probability]]
  [, /DOUBLE] [, GAMMA=integer{>0}] [, /NORMAL] [, POISSON=value]
  [, /UNIFORM] | [, /LONG] ] )
```

Return Value

Returns an array containing the random numbers of the specified dimensions.

Arguments

Seed

A variable or constant used to initialize the random sequence on input, and in which the state of the random number generator is saved on output.

The state of the random number generator is contained in a long integer vector. This state is saved in the Seed argument when the argument is a named variable. To continue the pseudo-random number sequence, input the variable containing the saved state as the Seed argument in the next call to RANDOMN or RANDOMU. Each independent random number sequence should maintain its own state variable. To maintain a state over repeated calls to a procedure, the seed variable may be stored in a COMMON block.

In addition to states maintained by the user in variables, the RANDOMU and RANDOMN functions contain a single shared generic state that is used if a named variable is not supplied as the Seed argument or the named variable supplied is undefined. The generic state is initialized once using the time-of-day, and may be re-initialized by providing a Seed argument that is a constant or expression.

If the Seed argument is:

- an undefined variable — the generic state is used and the resulting generic state array is stored in the variable.

- a named variable that contains a longword array of the proper length — it is used to continue the pseudo-random sequence, and is updated.
- a named variable containing a scalar — the scalar value is used to start a new sequence and the resulting state array is stored back in the variable.
- a constant or expression — the value is used to re-initialize the generic state.

Note

RANDOMN and RANDOMU use the same sequence. Starting or restarting the sequence for one starts or restarts the sequence for the other. Some IDL routines use the random number generator, so using them will initialize the seed sequence. An example of such a routine is CLUST_WTS.

Note

Do not alter the seed value returned by this function. The only valid use for the seed argument is to pass it back to a subsequent call. Changing the value of the seed will corrupt the random sequence.

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If no dimensions are specified, RANDOMN returns a scalar result

Keywords

The formulas for the binomial, gamma, and Poisson distributions are from section 7.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press.

BINOMIAL

Set this keyword to a 2-element array, $[n, p]$, to generate random deviates from a binomial distribution. If an event occurs with probability p , with n trials, then the number of times it occurs has a binomial distribution.

Note

For $n > 1.0 \times 10^7$, you should set the DOUBLE keyword.

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

RANDOMN constructs double-precision uniform random deviates using the formula:

$$Y = \frac{(i1 - 1) \cdot imax + i2}{imax^2 + 1}$$

where $i1$ and $i2$ are integer random deviates in the range $[1..imax]$, and $imax = 2^{31} - 2$ is the largest possible integer random deviate. The Y values will be in the range $0 < Y < 1$.

GAMMA

Set this keyword to an integer order $i > 0$ to generate random deviates from a gamma distribution. The gamma distribution is the waiting time to the i th event in a Poisson random process of unit mean. A gamma distribution of order equal to 1 is the same as the exponential distribution.

Note

For $GAMMA > 1.0 \times 10^7$, you should set the *DOUBLE* keyword.

LONG

Set this keyword to return integer uniform random deviates in the range $[1..2^{31} - 2]$. If *LONG* is set, all other keywords are ignored.

NORMAL

Set this keyword to generate random deviates from a normal distribution.

POISSON

Set this keyword to the mean number of events occurring during a unit of time. The *POISSON* keyword returns a random deviate drawn from a Poisson distribution with that mean.

Note

For $POISSON > 1.0 \times 10^7$, you should set the *DOUBLE* keyword.

UNIFORM

Set this keyword to generate random deviates from a uniform distribution.

Examples

If you start the sequence with an *undefined* variable—if RANDOMN has already been called, *Seed* is no longer undefined—IDL initializes the sequence with the system time:

```
; Generate one random variable and initialize the sequence with an
; undefined variable:
randomValue = RANDOMN(seed)
```

The new state is saved in *seed*. To generate repeatable experiments, begin the sequence with a particular seed:

```
seed_value = 5L

; Generate one random variable and initialize the sequence with 5:
randomValue = RANDOMN(seed_value)

PRINT, randomValue
```

IDL prints:

```
0.521414
```

To restart the sequence with a particular seed, re-initialize the variable:

```
seed = 5L

;Get a normal random number, and restart the sequence with a seed
;of 5.
randomValue = RANDOMN(seed)

PRINT, randomValue
```

IDL prints:

```
0.521414
```

To continue the same sequence:

```
PRINT, RANDOMN(seed)
```

IDL prints:

```
-0.945489
```

To create a 10 by 10 array of normally-distributed, random numbers, type:

```
R = RANDOMN(seed, 10, 10)
```

Since seed is undefined, the generic state is used to initialize the random number generator. Print the resulting values by entering:

```
PRINT, R
```

A more interesting example plots the probability function of 2000 numbers returned by RANDOMN. Type:

```
PLOT, HISTOGRAM(RANDOMN(SEED, 2000), BINSIZE=0.1)
```

To obtain a sequence of 1000 exponential (gamma distribution, order 1) deviates, type:

```
Result = RANDOMN(seed, 1000, GAMMA=1)
```

Intuitively, the result contains a random series of waiting times for events occurring an average of one per time period.

To obtain a series of 1000 random elapsed times required for the arrival of two events, type:

```
;Returns a series of 1000 random elapsed times required for the  
;arrival of two events.  
Result = RANDOMN(seed, 1000, GAMMA=2)
```

To obtain a 128 x 128 array filled with Poisson deviates, with a mean of 1.5, type:

```
Result = RANDOMN(seed, 128, 128, POISSON=1.5)
```

To simulate the count of “heads” obtained when flipping a coin 10 times, type:

```
Result = RANDOMN(seed, BINOMIAL=[10, .5])
```

Version History

Introduced: Original

See Also

[RANDOMU](#)

RANDOMU

The RANDOMU function returns one or more uniformly-distributed, floating-point, pseudo-random numbers in the range $0 < Y < 1.0$.

The random number generator is taken from: “Random Number Generators: Good Ones are Hard to Find”, Park and Miller, *Communications of the ACM*, Oct 1988, Vol 31, No. 10, p. 1192. To remove low-order serial correlations, a Bays-Durham shuffle is added, resulting in a random number generator similar to ran1() in Section 7.1 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press.

Syntax

```
Result = RANDOMU( Seed [, D1 [, ..., D8]] [ [, BINOMIAL=[trials, probability]]
  [, /DOUBLE] [, GAMMA=integer{>0}] [, /NORMAL] [, POISSON=value]
  [, /UNIFORM] | [, /LONG] ] )
```

Return Value

Returns an array of uniformly distributed random numbers of the specified dimensions.

Arguments

Seed

A variable or constant used to initialize the random sequence on input, and in which the state of the random number generator is saved on output.

The state of the random number generator is contained in a long integer vector. This state is saved in the Seed argument when the argument is a named variable. To continue the pseudo-random number sequence, input the variable containing the saved state as the Seed argument in the next call to RANDOMN or RANDOMU. Each independent random number sequence should maintain its own state variable. To maintain a state over repeated calls to a procedure, the seed variable may be stored in a COMMON block.

In addition to states maintained by the user in variables, the RANDOMU and RANDOMN functions contain a single shared generic state that is used if a named variable is not supplied as the Seed argument or the named variable supplied is

undefined. The generic state is initialized once using the time-of-day, and may be re-initialized by providing a Seed argument that is a constant or expression.

If the Seed argument is:

- an undefined variable — the generic state is used and the resulting generic state array is stored in the variable.
- a named variable that contains a longword array of the proper length — it is used to continue the pseudo-random sequence, and is updated.
- a named variable containing a scalar — the scalar value is used to start a new sequence and the resulting state array is stored back in the variable.
- a constant or expression — the value is used to re-initialize the generic state.

Note

RANDOMN and RANDOMU use the same sequence, so starting or restarting the sequence for one starts or restarts the sequence for the other. Some IDL routines use the random number generator, so using them will initialize the seed sequence. An example of such a routine is CLUST_WTS.

Note

Do not alter the seed value returned by this function. The only valid use for the seed argument is to pass it back to a subsequent call. Changing the value of the seed will corrupt the random sequence.

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If no dimensions are specified, RANDOMU returns a scalar result

Keywords

The formulas for the binomial, gamma, and Poisson distributions are from Section 7.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press.

BINOMIAL

Set this keyword to a 2-element array, $[n, p]$, to generate random deviates from a binomial distribution. If an event occurs with probability p , with n trials, then the number of times it occurs has a binomial distribution.

Note

For $n > 1.0 \times 10^7$, you should set the DOUBLE keyword.

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

RANDOMU constructs double-precision uniform random deviates using the formula:

$$Y = \frac{(i1 - 1) \cdot imax + i2}{imax^2 + 1}$$

where $i1$ and $i2$ are integer random deviates in the range $[1 \dots imax]$, and $imax = 2^{31} - 2$ is the largest possible integer random deviate. The Y values will be in the range $0 < Y < 1$.

GAMMA

Set this keyword to an integer order $i > 0$ to generate random deviates from a gamma distribution. The gamma distribution is the waiting time to the i th event in a Poisson random process of unit mean. A gamma distribution of order equal to 1 is the same as the exponential distribution.

Note

For GAMMA $> 1.0 \times 10^7$, you should set the DOUBLE keyword.

LONG

Set this keyword to return integer uniform random deviates in the range $[1 \dots 2^{31} - 2]$. If LONG is set, all other keywords are ignored.

NORMAL

Set this keyword to generate random deviates from a normal distribution.

POISSON

Set this keyword to the mean number of events occurring during a unit of time. The POISSON keyword returns a random deviate drawn from a Poisson distribution with that mean.

Note

For POISSON $> 1.0 \times 10^7$, you should set the DOUBLE keyword.

UNIFORM

Set this keyword to generate random deviates from a uniform distribution.

Examples

This example simulates rolling two dice 10,000 times and plots the distribution of the total using RANDOMU. Enter:

```
PLOT, HISTOGRAM(FIX(6 * RANDOMU(S, 10000)) + $
                FIX(6 * RANDOMU(S, 10000)) + 2)
```

In the above statement, the expression RANDOMU(S, 10000) is a 10,000-element, floating-point array of random numbers greater than or equal to 0 and less than 1. Multiplying this array by 6 converts the range to $0 \leq Y < 6$.

Applying the FIX function yields a 10,000-point integer vector with values from 0 to 5, one less than the numbers on one die. This computation is done twice, once for each die, then 2 is added to obtain a vector from 2 to 12, the total of two dice.

The HISTOGRAM function makes a vector in which each element contains the number of occurrences of dice rolls whose total is equal to the subscript of the element. Finally, this vector is plotted by the PLOT procedure.

An example of reusing a state vector to generate a repeatable sequence:

```
; Init seed for a repeatable sequence:
seed = 1001L

; Print 1st 5 numbers of sequence:
print, randomu(seed, 5)
```

IDL prints:

```
0.705884      0.285924      0.231151      0.715447      0.532836
```

Reuse a state vector:

```
; Re-init seed to same sequence:  
seed = 1001L  
  
; Get 5 number of sequence with 5 calls:  
for i=0,4 do print, randomu(seed)
```

IDL prints:

```
0.705884  
0.285924  
0.231151  
0.715447  
0.532836
```

Version History

Introduced: Original

See Also

[RANDOMN](#)

RANKS

The RANKS function computes the magnitude-based ranks of a sample population X . Elements of identical magnitude “ties” are ranked according to the mean of the ranks that would otherwise be assigned.

This routine is written in the IDL language. Its source code can be found in the file `ranks.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = RANKS(*X*)

Return Value

The result is a vector of ranks equal in length to X .

Arguments

X

An n -element integer, single-, or double-precision floating-point vector. The elements of this vector must be in ascending order based on their magnitude.

Keywords

None.

Examples

```
; Define an n-element sample population:
X = [-0.8, 0.1, -2.3, -0.6, 0.2, 1.1, -0.3, 0.6, -0.2, 1.1, $
      -0.7, -0.2, 0.6, 0.4, -0.1, 1.1, -0.3, 0.3, -1.3, 1.1]

; Allocate a two-column, n-row array to store the results:
array = FLTARR(2, N_ELEMENTS(X))

; Sort the sample population and store in the 0th column of ARRAY:
array[0, *] = X[SORT(X)]
; Compute the ranks of the sorted sample population and store in
; the 1st column of ARRAY:
array[1, *] = RANKS(X[SORT(X)])
```

```
    ; Display the sorted sample population and corresponding ranks
    ; with a two-decimal format:
    PRINT, array, FORMAT = '(2(5x, f5.2))'
```

IDL prints:

```
-2.30      1.00
-1.30      2.00
-0.80      3.00
-0.70      4.00
-0.60      5.00
-0.30      6.50
-0.30      6.50
-0.20      8.50
-0.20      8.50
-0.10     10.00
 0.10     11.00
 0.20     12.00
 0.30     13.00
 0.40     14.00
 0.60     15.50
 0.60     15.50
 1.10     18.50
 1.10     18.50
 1.10     18.50
 1.10     18.50
```

Version History

Introduced: 4.0

See Also

[R_CORRELATE](#)

RDPIX

The RDPIX procedure interactively displays the X position, Y position, and pixel value at the cursor.

This routine is written in the IDL language. Its source code can be found in the file `rdpix.pro` in the `lib` subdirectory of the IDL distribution.

Using RDPIX

RDPIX displays a stream of X, Y, and pixel values when the mouse cursor is moved over a graphics window. Press the left or center mouse button to create a new line of output. Press the right mouse button to exit the procedure.

Syntax

RDPIX, *Image* [, *X0*, *Y0*]

Arguments

Image

The array that contains the image being displayed. This array may be of any type. Rather than reading pixel values from the display, values are taken from this parameter, avoiding scaling difficulties.

X0, Y0

The location of the lower-left corner of the image area on screen. If these parameters are not supplied, they are assumed to be zero.

Keywords

None.

Version History

Introduced: Original

See Also

[CURSOR](#), [TVRD](#)

READ/READF

The READ procedures perform formatted input into variables.

READ performs input from the standard input stream (IDL file unit 0), while READF requires a file unit to be explicitly specified.

Syntax

```
READ, [Prompt,] Var1, ..., Varn
```

```
READF, [Prompt,] Unit, Var1, ..., Varn
```

Keywords: [, [AM_PM](#)=*[string, string]*] [, [DAYS_OF_WEEK](#)=*string_array*{7 names}] [, [FORMAT](#)=*value*] [, [MONTHS](#)=*string_array*{12 names}] [, [PROMPT](#)=*string*]

Arguments

Prompt

Note

The PROMPT keyword should be used instead of the *Prompt* argument for compatibility with window-based versions of IDL.

A string or explicit expression (i.e., not a named variable) to be used as a prompt. This argument should not be included if the FORMAT keyword is specified. Also, if this argument begins with the characters “\$”, it is taken to be a format specification as described below under “Format Compatibility”.

Using the *Prompt* argument does not work well with IDL for Windows. The desired prompt string is written to the log window instead of the command input window. To create custom prompts compatible with these versions of IDL, use the PROMPT keyword, described below.

Unit

For READF, Unit specifies the file unit from which the input is taken.

Var_{*i*}

The named variables to receive the input.

Note

Var_i may *not* be an element of an array variable (`arrayvar[n]`) or a field within a structure variable (`structvar.tag`). See “[Parameter Passing Mechanism](#)” in Chapter 4 of the *Building IDL Applications* manual for details.

If the variable specified for the Var_i argument has not been previously defined, the input data is assumed to be of type float, and the variable will be created as a float.

Keywords

AM_PM

Supplies a string array of two names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the FORMAT keyword.

DAYS_OF_WEEK

Supplies a string array of 7 names to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the FORMAT keyword.

FORMAT

If FORMAT is not specified, IDL uses its default rules for formatting the input. FORMAT allows the format of the input to be specified in precise detail, using a FORTRAN-style specification. See “[Using Explicitly Formatted Input/Output](#)” in Chapter 11 of the *Building IDL Applications* manual.

MONTHS

Supplies a string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the FORMAT keyword.

PROMPT

Set this keyword to a scalar string to be used as a customized prompt for the READ command. If the PROMPT keyword or *Prompt* argument is not supplied, IDL uses a colon followed by a space (“: ”) as the input prompt.

Obsolete Keywords

The following keywords are obsolete:

- KEY_ID
- KEY_MATCH
- KEY_VALUE

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Format Compatibility

If the FORMAT keyword is not present and READ is called with more than one argument, and the first argument is a scalar string starting with the characters “\$(”, this initial argument is taken to be the format specification, just as if it had been specified via the FORMAT keyword. This feature is maintained for compatibility with version 1 of VMS IDL.

Examples

To read a string value into the variable B from the keyboard, enter:

```
; Define B as a string before reading:
B = ' '

; Read input from the terminal:
READ, B, PROMPT='Enter Name: '
```

To read formatted data from the previously-opened file associated with logical unit number 7 into variable C, use the command:

```
READF, 7, C
```

Version History

Introduced: Original

See Also

[READS](#), [READU](#), [WRITEU](#)

READ_ASCII

The READ_ASCII function reads data from an ASCII file into an IDL structure variable. READ_ASCII may be used with templates created by the ASCII_TEMPLATE function.

This routine handles ASCII files consisting of an optional header of a fixed number of lines, followed by columnar data. One or more rows of data constitute a *record*. Each data element within a record is considered to be in a different column, or *field*. The data in one field must be of, or promotable to, a single type (e.g., FLOAT). Adjacent fields may be collected into multi-column fields, called *groups*. Files may also contain comments, which exist between a user-specified comment string and the corresponding end-of-line.

READ_ASCII is designed to be used with templates created by the ASCII template function.

This routine is written in the IDL language. Its source code can be found in the file `read_ascii.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = READ_ASCII( [Filename] [, COMMENT_SYMBOL=string]
    [, COUNT=variable] [, DATA_START=lines_to_skip] [, DELIMITER=string]
    [, HEADER=variable] [, MISSING_VALUE=value] [, NUM_RECORDS=value]
    [, RECORD_START=index] [, TEMPLATE=value] [, /VERBOSE] )
```

Arguments

Filename

A string containing the name of an ASCII file to read into an IDL variable. If *filename* is not specified, a dialog allows the user to choose a file.

Keywords

You can define the attributes of a field in two ways. If you use a template, you can either use a previously generated template, or create a template with [ASCII_TEMPLATE](#). You can use COMMENT_SYMBOL, DATA_START, DELIMITER, or MISSING_VALUE to either override template attributes or to provide one-time attributes for the file to be read, without a template.

COMMENT_SYMBOL

Set this keyword to a string that identifies the character used to delineate comments in the ASCII file to be read. When `READ_ASCII` encounters the comment character, it discards data from that point until it reaches the end of the current line, identifying the rest of the line as a comment. The default character is the null string, `'`, specifying that no comments will be recognized.

COUNT

Set this keyword equal to a named variable that will contain the number of records read.

DATA_START

Set this keyword equal to the number of header lines you want to skip. The default value is 0 if no template is specified.

DELIMITER

Set this keyword to a string that identifies the end of a field. If no delimiter is specified, `READ_ASCII` uses information provided by the template in use. The default is a space, `' '`, specifying that an empty element constitutes the end of a field.

HEADER

Set this keyword equal to a named variable that will contain the header in a string array of length `DATA_START`. If no header exists, an empty string is returned.

MISSING_VALUE

Set this keyword equal to a value used to replace any missing or invalid data. The default value, if no template is supplied, is `!VALUES.F_NAN`. See [“!VALUES”](#) on page 4357 for details.

NUM_RECORDS

Set this keyword equal to the number of records to read. The default is to read up to and including the last record.

RECORD_START

Set this keyword equal to the index of the first record to read. The default is the first record of the file (record 0).

TEMPLATE

Use this keyword to specify the ASCII file template (generated by the function [ASCII_TEMPLATE](#)), that defines attributes of the file to be read. Specific attributes of the template may be overridden by the following keywords: COMMENT_SYMBOL, DATA_START, DELIMITER, MISSING_VALUE.

VERBOSE

Set this keyword to print runtime messages.

Examples

To read ASCII data using default file attributes, except for setting the number of skipped header lines to 10, type:

```
data = READ_ASCII(file, DATA_START=10)
```

To use a template to define file attributes, overriding the number of skipped header lines defined in the template, type:

```
data = READ_ASCII(file, TEMPLATE=template, DATA_START=10)
```

To use the ASCII_TEMPLATE GUI to generate a template within the function, type:

```
data = READ_ASCII(myfile, TEMPLATE=ASCII_TEMPLATE(myfile))
```

Version History

Introduced: 5.0

See Also

[ASCII_TEMPLATE](#)

READ_BINARY

The READ_BINARY function reads the contents of a binary file using a passed template or basic command line keywords. Data is read from the given filename or from the current file position in the open file pointed to by FileUnit. If no template is provided, keywords can be used to read a single IDL array of data.

Syntax

```
Result = READ_BINARY ([Filename] | FileUnit [, TEMPLATE=template] |  
  [[, DATA_START=value] [, DATA_TYPE=typecodes] [, DATA_DIMS=array]  
  [, ENDIAN=string])
```

Return Value

The result is an array or anonymous structure containing all of the entities read from the file.

Arguments

Filename

A scalar string containing the name of the binary file to read. If *filename* and file unit are not specified, a dialog allows the user to choose a file.

FileUnit

A scalar containing an open IDL file unit number to read from.

Keywords

DATA_DIMS

Set this keyword to a scalar or array of up to eight elements specifying the size of the data to be read and returned. For example, DATA_DIMS=[512,512] specifies that a two-dimensional, 512 by 512 array be read and returned. DATA_DIMS=0 specifies that a single, scalar value be read and returned. Default is -1, which, if a TEMPLATE is not supplied that specifies otherwise, indicates that READ_BINARY will read to end-of-file and store the result in a 1-D array.

DATA_START

Set this keyword to specify where to begin reading in a file. This value is as an offset, in bytes, that will be applied to the initial position in the file. The default is 0.

DATA_TYPE

Set this keyword to an IDL typecode of the data to be read. See documentation for the [SIZE](#) function for a listing of typecodes. Default is 1 (IDL's BYTE typecode).

ENDIAN

Set this keyword to one of three string values: “big”, “little” or “native” which specifies the byte ordering of the file to be read. If the computer running `READ_BINARY` uses byte ordering that is different than that of the file, `READ_BINARY` will swap the order of bytes in multi-byte data types read from the file. (Default: “native” = perform no byte swapping.)

TEMPLATE

Set this keyword to a template structure (created using the [BINARY_TEMPLATE](#) function) describing the file to be read. The `TEMPLATE` keyword cannot be used simultaneously with keywords `DATA_START`, `DATA_TYPE`, `DATA_DIMS`, or `ENDIAN`.

When a template is used with `READ_BINARY`, the return value is a structure containing fields specified by the template. If a template is not used, the return value is an array.

Version History

Introduced: 5.3

See Also

[BINARY_TEMPLATE](#)

READ_BMP

The READ_BMP function reads a Microsoft Windows Version 3 device independent bitmap file (.BMP) and returns the image.

READ_BMP does not handle 1-bit-deep images or compressed images, and is not fast for 4-bit images. The algorithm works best on images where the number of bytes in each scan-line is evenly divisible by 4.

This routine is written in the IDL language. Its source code can be found in the file `read_bmp.pro` in the `lib` subdirectory of the IDL distribution.

Note

To find information about a potential BMP file before trying to read its data, use the [QUERY_BMP](#) function.

Syntax

```
Result = READ_BMP( Filename, [, R, G, B] [, Ihdr] [, /RGB] )
```

Return Value

Returns a byte array containing the image. Dimensions are taken from the BITMAPINFOHEADER of the file. In the case of 4-bit or 8-bit images, the dimensions of the resulting array are (`biWidth`, `biHeight`).

For 24-bit images, the dimensions are (3, `biWidth`, `biHeight`). Color interleaving is blue, green, red; i.e., `Result[0,i,j]` = blue, `Result[1,i,j]` = green, etc.

Arguments

Filename

A scalar string specifying the full path name of the bitmap file to read.

R, G, B

Named variables that will contain the color tables from the file. There 16 elements each for 4 bit images, 256 elements each for 8 bit images. Color tables are not defined or used for 24 bit images.

lhdr

A named variable that will contain a structure holding the BITMAPINFOHEADER from the file. Tag names are as defined in the MS Windows Programmer's Reference Manual, Chapter 7.

Keywords

RGB

If this keyword is set, color interleaving of 16- and 24-bit images will be R, G, B, i.e., $\text{Result}[0,i,j] = \text{red}$, $\text{Result}[1,i,j] = \text{green}$, $\text{Result}[2,i,j] = \text{blue}$.

Examples

To open, read, and display the BMP file named `foo.bmp` in the current directory and store the color vectors in the variables R, G, and B, enter:

```
; Read and display an image:
TV, READ_BMP('foo.bmp', R, G, B)

; Load its colors:
TVLCT, R, G, B
```

Many applications that use 24-bit BMP files outside IDL expect BMP files to be stored as BGR. For example, enter the following commands.

```
; Make a red square image:
a = BYTARR(3, 200, 200)
a[0, *, *] = 255

;View the image:
TV, a, /TRUE
WRITE_BMP, 'foo.bmp', a
```

If you open the `.bmp` file in certain bitmap editors, you may find that the square is blue.

```
image = READ_BMP('foo.bmp')

; IDL reads the image back in and interprets it as red:
TV, image, /TRUE

; Flip the order of the indices by adding the RGB keyword:
image = READ_BMP('foo.bmp', /RGB)

; Displays the image in blue:
TV, image, /TRUE
```

Version History

Introduced: Pre 4.0

See Also

[WRITE_BMP](#), [QUERY_BMP](#)

READ_DICOM

The READ_DICOM function reads an image from a DICOM file along with any associated color table. The return array type depends on the DICOM image pixel type.

This routine is written in the IDL language. Its source code can be found in the file `read_dicom.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = READ_DICOM (Filename [, Red, Green, Blue] [, IMAGE_INDEX=index])
```

Return Value

The return value can be a 2-D array for grayscale or a 3-D array for TrueColor images. TrueColor images are always returned in pixel interleaved format.

Arguments

Filename

This argument is a scalar string that contains the full pathname of the file to read.

Red, Green, Blue

Named variables that will contain the red, green, and blue color vectors from the DICOM file if they exist.

Note

DICOM color vectors contain 16-bit color values that may need to be converted for use with IDL graphics routines.

Keywords

IMAGE_INDEX

Set this keyword to the index of the image to read from the file.

Examples

```
TVSCL, READ_DICOM(FILEPATH('mr_knee.dcm'), $  
SUBDIR=['examples', 'data'])
```

Version History

Introduced: 5.2

See Also

[QUERY_DICOM](#)

READ_GIF

The READ_GIF procedure reads the contents of a GIF format image file and returns the image and color table vectors (if present) in the form of IDL variables. Only 8-bit images are supported, and local colormaps are not supported.

Note

The GIF format uses compression technologies that may be subject to license restrictions. If you receive a warning when attempting to use this routine, contact RSI for information about licensing options.

This routine is written in the IDL language. Its source code can be found in the file `read_gif.pro` in the `lib` subdirectory of the IDL distribution.

Note

To find information about a potential GIF file before trying to read its data, use the QUERY_GIF function.

Syntax

```
READ_GIF, Filename, Image [, R, G, B] [, /CLOSE] [, /MULTIPLE]
```

Arguments

Filename

A scalar string specifying the full path name of the image file to read.

Image

A named variable that will contain the image data read from the file.

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors, if the rasterfile contains colormaps.

Keywords

CLOSE

Set this keyword to close any open files. The CLOSE keyword is only useful if a file containing multiple images (as specified by the MULTIPLE keyword) is in use. Note that you do not need to specify the normal arguments to READ_GIF (*Filename*, *Image*, etc.) when using this keyword.

MULTIPLE

Set this keyword to read files that contain multiple images. Each call to READ_GIF returns the next image, with the file remaining open between calls. The *Filename* argument is ignored and may be omitted after the first call. Reading past the last image returns a scalar value of -1 in the variable specified in the *Image* argument, and closes the file. R, G, and B color vectors are returned only once, along with the first image.

Example

To open and read the GIF image file named `foo.gif` in the current directory, store the image in the variable `image1`.

```
; Store the color vectors in the variables R, G, and B:  
READ_GIF, 'foo.gif', image1, R, G, B  
  
; Load the new color table and display the image:  
TVLCT, R, G, B  
TV, image1
```

See Also

[WRITE_GIF](#), [QUERY_GIF](#)

READ_IMAGE

The READ_IMAGE function reads the image contents of a file and returns the image in an IDL variable. If the image contains a palette it can be returned as well in three IDL variables. READ_IMAGE returns the image in the form of a two-dimensional array (for grayscale images) or a (3, n, m) array (for TrueColor images). READ_IMAGE can read most types of image files supported by IDL. See QUERY_IMAGE for a list of supported formats.

Syntax

```
Result = READ_IMAGE (Filename [, Red, Green, Blue] [, IMAGE_INDEX=index] )
```

Return Value

Result is the image array read from the file or scalar value of -1 if the file could not be read.

Arguments

Filename

A scalar string containing the name of the file to read.

Red

An optional named variable to receive the red channel of the color table if a color table exists.

Green

An optional named variable to receive the green channel of the color table if a color table exists.

Blue

An optional named variable to receive the blue channel of the color table if a color table exists.

Keywords

IMAGE_INDEX

Set this keyword to the index of the image to read from the file. The default is 0, the first image.

Version History

Introduced: 5.3

READ_INTERFILE

The `READ_INTERFILE` procedure reads image data stored in Interfile (v3.3) format.

`READ_INTERFILE` can only read a series of images if all images have the same height and width. It does not get additional keyword information beyond what is needed to read the image data. If any problems occur when reading the file, `READ_INTERFILE` prints a message and stops.

If the data is stored on a bigendian machine and read on a littleendian machine (or vice versa) the order of bytes in each pixel element may be reversed, requiring a call to `BYTEORDER`

This routine is written in the IDL language. Its source code can be found in the file `read_interfile.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

`READ_INTERFILE, File, Data`

Return Value

Returns a 3-D array containing the image data.

Arguments

File

A scalar string containing the name of the Interfile to read. Note: if the Interfile has a header file and a data file, this should be the name of the header file (also called the administrative file).

Data

A named variable that will contain a 3-D array of data as read from the file. Assumed to be a series of 2-D images.

Keywords

None.

Examples

```
READ_INTERFILE, '0_11.hdr', X
```

Version History

Introduced: Pre 4.0

READ_JPEG

The READ_JPEG procedure reads JPEG (Joint Photographic Experts Group) format compressed images from files or memory. JPEG is a standardized compression method for full-color and gray-scale images. This procedure reads JFIF, the JPEG File Interchange Format, including those produced by WRITE_JPEG. Such files are usually simply called JPEG files

READ_JPEG can optionally quantize TrueColor images contained in files to a pseudo-color palette with a specified number of colors, and with optional color dithering.

This procedure is based in part on the work of the Independent JPEG Group. For a brief explanation of JPEG, see “[WRITE_JPEG](#)” on page 2514.

Note

All JPEG files consist of byte data. Input data is converted to bytes before being written to a JPEG file.

Note

To find information about a potential JPEG file before trying to read its data, use the [QUERY_JPEG](#) function.

Syntax

```
READ_JPEG [, Filename | , UNIT=lun] , Image [, Colortable] [, BUFFER=variable]
  [, COLORS=value{8 to 256}] [, DITHER={0 | 1 | 2}] [, /GRAYSCALE]
  [, /ORDER] [, TRUE={1 | 2 | 3}] [, /TWO\_PASS\_QUANTIZE ]
```

Arguments

Filename

A scalar string specifying the full pathname of the JFIF format (JPEG) file to be read. If this parameter is not present, the UNIT and/or the BUFFER keywords must be specified.

Image

A named variable to contain the image data read from the file.

Colortable

A named variable to contain the colormap, when reading a TrueColor image that is to be quantized. On completion, this variable contains a byte array with dimensions (NCOLORS, 3). This argument is filled only if the image is color quantized (refer to the COLORS keyword).

Keywords

BUFFER

Set this keyword to a named variable that is used for a buffer. A buffer variable need only be supplied when reading multiple images per file. Initialize the buffer variable to empty by setting it to 0.

COLORS

If the image file contains a TrueColor image that is to be displayed on an indexed color destination, set COLORS to the desired number of colors to be quantized. COLORS can be set to a value from 8 to 256. The DITHER and TWO_PASS_QUANTIZE keywords affect the method, speed, and quality of the color quantization. These keywords have no effect if the file contains a grayscale image.

DITHER

Set this keyword to use dithering with color quantization (i.e., if the COLORS keyword is in use). Dithering is a method that distributes the color quantization error to surrounding pixels, to achieve higher-quality results. Set the DITHER keyword to one of the following values:

- 0 = No dithering. Images are read quickly, but quality is low.
- 1 = Floyd-Steinberg dithering. This method is relatively slow, but produces the highest quality results. This is the default behavior.
- 2 = Ordered dithering. This method is faster than Floyd-Steinberg dithering, but produces lower quality results.

GRAYSCALE

Set this keyword to return a monochrome (grayscale) image, regardless of whether the file contains an RGB or grayscale image.

ORDER

JPEG/JFIF images are normally written in top-to-bottom order. If the image is to be stored into the *Image* array in the standard IDL order (from bottom-to-top) set ORDER to 0. This is the default. If the image array is to be top-to-bottom order, set ORDER to 1.

TRUE

Use this keyword to specify the index of the dimension for color interleaving when reading a TrueColor image. The default is 1, for pixel interleaving, $(3, m, n)$. A value of 2 indicates line interleaving $(m, 3, n)$, and 3 indicates band interleaving, $(m, n, 3)$.

TWO_PASS_QUANTIZE

Set this keyword to use a two-pass color quantization method when quantization is in effect (i.e., the COLORS keyword is in use). This method requires more memory and time, but produces superior results to the default one-pass quantization method.

UNIT

This keyword can be used to designate the logical unit number of an already open JFIF file, allowing the reading of multiple images per file or the embedding of JFIF images in other data files. When using this keyword, *Filename* should not be specified.

Note

When opening a file intended for use with the UNIT keyword, if the filename does not end in .jpg, or .jpeg, you must specify the STDIO keyword to OPEN in order for the file to be compatible with READ_JPEG.

Examples

```
; Read a JPEG grayscale image:
READ_JPEG, 'test.jpg', a

; Display the image:
TV, a

; Read and display a JPEG TrueColor image on a TrueColor display:
READ_JPEG, 'test.jpg', a, TRUE=1

; Display the image returned with pixel interleaving
; (i.e., with dimensions 3, m, n):
```

```
TV, a, TRUE=1
```

Read the image, setting the number of colors to be quantized to the maximum number of available colors.

```
; Read a JPEG TrueColor image on an 8-bit pseudo-color display:
READ_JPEG, 'test.jpg', a, ctable, COLORS=!D.N_COLORS-1
```

```
; Display the image:
TV, a
```

```
; Load the quantized color table:
TVLCT, ctable
```

We could have also included the `TWO_PASS_QUANTIZE` and `DITHER` keywords to improve the image's appearance.

Using the `BUFFER` keyword.

```
; Initialize buffer:
buff = 0
OPENR, 1, 'images.jpg'

; Process each image:
FOR i=1, nimages DO BEGIN
  ; Read next image:
  READ_JPEG, UNIT=1, BUFFER=buff, a

  ; Display the image:
  TV, a
ENDFOR

; Done:
CLOSE, 1
```

Version History

Introduced: Pre 4.0

See Also

[WRITE_JPEG](#), [QUERY_JPEG](#)

READ_JPEG2000

The `READ_JPEG2000` function extracts and returns image data from a JPEG2000 file. This function is a wrapper around the `IDLffJPEG2000` object interface that presents JPEG2000 image loading in a familiar way to users of the `READ_*` image routines. However this function is not as efficient as the object interface and the object interface should be used whenever possible. See “`IDLffJPEG2000`” in Chapter 6 of the *IDL Reference Guide* manual for information about the object interface.

Syntax

```
Result = READ_JPEG2000(Filename [, Red, Green, Blue]  
    [, DISCARD_LEVELS=value] [, MAX_LAYERS=value] [, /ORDER]  
    [, REGION=[StartX, StartY, Width, Height] ] )
```

Return Value

The *Result* is a two or three-dimensional array containing the image data. The dimensions of the result are [*nComponents*, *Width*, *Height*], where *nComponents* is the number of components present in the file. Images with only one component will be returned as a two dimensional array [*Width*, *Height*].

The data type of the result is automatically determined using the bit-depth and signed properties of the returned components. For bit-depth = 8 the result will be of type byte. For bit-depths ≥ 9 and ≤ 16 the result will be of type integer or uint. For bit-depths > 16 the result will be long or ulong. Components with differing bit-depths will be returned using the largest bit-depth. Components which are a mix of signed and unsigned will be returned as signed.

Note

The returned image is automatically ordered from bottom-to-top, where the first pixel returned is located at the bottom-left corner of the image. The `ORDER` keyword may be used to override this behavior.

Arguments

Filename

A scalar string containing the full path and filename of the JPEG2000 file to read.

Red, Green, Blue

Set these arguments to named variables that will contain the Red, Green, and Blue color vectors of the color palette from the file.

Keywords

DISCARD_LEVELS

Set this keyword to an integer giving the number of highest resolution levels which will not appear in the result.

The resulting image dimensions are given by:

$$\text{CEIL}(\text{Dimensions} + \text{Offset}) / (2^{\text{DL}}) - \text{CEIL}(\text{Offset} / (2^{\text{DL}}))$$

where *Dimensions* and *Offset* are the image dimensions and offset, and *DL* is the value of DISCARD_LEVELS.

For example, if the image in the file has dimensions of 1024 x 1026, and DISCARD_LEVELS is set to 3, the resulting image will have dimensions of 128 x 129, which is the ceiling of [1024, 1026] divided by 2^3 . The number of resolution levels in a file may be determined by checking the value of the N_LEVELS field within the QUERY_JPEG2000 structure.

MAX_LAYERS

Set this keyword to an integer giving the maximum number of quality layers which will be returned in the result. Each layer contains the information required to represent the image at a higher quality, given the information from all the previous layers. A value of 0, the default, implies that all layers should be returned. For example, specifying MAX_LAYERS=4 processes only the first four quality layers. The number of quality layers in a file may be determined by checking the value of the N_LAYERS field within the QUERY_JPEG2000 structure.

ORDER

JPEG2000 images are assumed to be stored in top-to-bottom order, while IDL usually assumes images are in bottom-to-top order. By default (ORDER = 0), READ_JPEG2000 will automatically flip the result to match IDL's bottom-to-top order. Setting ORDER to a nonzero value will return the JPEG2000 image without doing the vertical flip.

REGION

Set this keyword to a four-element vector containing the rectangular region of the image to read, in the canvas coordinate system. The region is specified as [*StartX*, *StartY*, *Width*, *Height*], where *StartX* and *StartY* must be in the range $0 \dots 2^{32}-2$, and *Width* and *Height* must be in the range $1 \dots 2^{32}-1$. The region must contain at least some portion of the image, or an error will occur. If the region is larger than the image, or extends outside of the image, then only that portion that lies within the image will be returned.

Note

The region parameters should be specified in the canvas coordinate system, taking into account the canvas offset and canvas dimensions. The dimensions and offset may be determined from the QUERY_JPEG2000 structure. See “[The JPEG2000 Canvas Coordinate System](#)” in Chapter 6 of the *IDL Reference Guide* manual for more information.

Example

```

; Read in a 24-bit JPEG image.
input = FILEPATH('marsglobe.jpg', SUBDIR=['examples','data'])
READ_JPEG, input, image

; Create a JPEG2000 with 6 quality layers.
WRITE_JPEG2000, 'marsglobe.jp2', image, N_LAYERS=6

; Verify the file information.
success = QUERY_JPEG2000('marsglobe.jp2', info)
help, info, /STRUCT

WINDOW, 0, XSIZE=2*info.dimensions[0], YSIZE=info.dimensions[1]

; Use the DISCARD_LEVELS keyword.
for discard=0,5 do TV, /TRUE, $
    READ_JPEG2000('marsglobe.jp2', DISCARD_LEVELS=discard)

; Extract a region.
image = READ_JPEG2000('marsglobe.jp2', $
    REGION=[0,0,200,100])
TV, REBIN(image, 3, 400, 200), 400, 0, /TRUE

; Use the MAX_LAYERS keyword.
image = READ_JPEG2000('marsglobe.jp2', MAX_LAYERS=1, $
    REGION=[0,0,200,100])
TV, REBIN(image, 3, 400, 200), 400, 200, /TRUE

```



Figure 3-52: Varying Quality Images

In the previous figure, the left-hand side shows the full resolution image, plus the 5 lower resolution versions (`DISCARD_LEVELS=1,2,3,4,5`). The right-hand side shows a magnified portion of the full quality image (bottom), and a lower quality version (top, `MAX_LAYERS=1`).

Version History

Introduced: 6.1

See Also

[QUERY_JPEG2000](#), [WRITE_JPEG2000](#)

READ_MRSID

The READ_MRSID function extracts and returns image data from a MrSID file at the specified level and location. It is a wrapper around the object interface that presents MrSID image loading in a familiar way to users of the READ_* image routines. However this function is not as efficient as the object interface and the object interface should be used whenever possible. See “IDLffMrSID” on page 2835 for information about the object interface.

Note

This routine is only available under Microsoft Windows.

Syntax

Result = READ_MRSID (*Filename* [, LEVEL=*lvl*] [, SUB_RECT=*rect*])

Return Value

ImageData returns an *n*-by-*w*-by-*h* array containing the image data where *n* is 1 for grayscale or 3 for RGB images, *w* is the width and *h* is the height.

Note

The returned image is ordered bottom-up, the first pixel returned is located at the bottom-left of the image. This differs from how data is stored in the MrSID file where the image is top-down, meaning the pixel at the start of the file is located at the top-left of the image.

Arguments

Filename

A scalar string or one element by one element array containing the full path and filename of the MrSID file to read.

Keywords

LEVEL

Set this keyword to an integer that specifies the level at which to read the image. If this keyword is not set, the maximum level (see [QUERY_MRSID](#)) is used which returns the minimum resolution.

SUB_RECT

Set this keyword to a four-element vector [x, y, xdim, ydim] specifying the position of the lower left-hand corner and the dimensions of the sub-rectangle of the MrSID image to return. This is useful for displaying only a portion of the high-resolution image. If this keyword is not set, the entire image will be returned. This may require significant memory if a high-resolution level is selected. If the sub-rectangle is greater than the bounds of the image at the selected level the area outside the image bounds will be set to black.

Note

The elements of SUB_RECT are measured in pixels at the current level. This means the point x = 10, y = 10 at level 1 will be located at x = 20, y = 20 at level 0 and x = 5, y = 5 at level 2.

Examples

```

; Query the file.
result = QUERY_MRSID(FILE_SEARCH(!DIR, 'test_gs.sid'), info)

; If result is not zero, read in an image from the file and
; display it.
IF (result NE 0) THEN BEGIN
    PRINT, info
    imageData = READ_MRSID(FILE_SEARCH(!DIR, 'test_gs.sid'), $
        SUB_RECT = [0, 0, 200, 200], LEVEL = 3)
    oImage = OBJ_NEW('IDLgrImage', imageData, ORDER = 0)
    XOBJVIEW, oImage, BACKGROUND = [255,255,0]
ENDIF

; Use the file access object to query the file.
oMrSID = OBJ_NEW('IDLffMrSID', FILE_SEARCH(!DIR, 'test_gs.sid'))
oMrSID->GetProperty, PIXEL_TYPE=pt, $
    CHANNELS = chan, DIMENSIONS = dims, $
    TYPE = type, LEVELS = lvlS
PRINT, pt, chan, dims, type, lvlS

```

```
; Use the object to read in an image from the file.  
lvls = -3  
dimsatlvl = oMrSID->GetDimsAtLevel(lvls)  
PRINT, dimsatlvl  
imageData = oMrSID->GetImageData(LEVEL = 3)  
PRINT, size(imageData)  
OBJ_DESTROY, oImage
```

Version History

Introduced: 5.5

READ_PICT

The READ_PICT procedure reads the contents of a PICT (version 2) format image file and returns the image and color table vectors (if present) in the form of IDL variables. The PICT format is used by Apple Macintosh computers.

This routine is written in the IDL language. Its source code can be found in the file `read_pict.pro` in the `lib` subdirectory of the IDL distribution.

Note

To find information about a potential PICT file before trying to read its data, use the [QUERY_PICT](#) function.

Syntax

READ_PICT, *Filename*, *Image* [, *R*, *G*, *B*]

Arguments

Filename

A scalar string specifying the full pathname of the PICT file to read.

Image

A named variable that will contain the 2-D image read from *Filename*.

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors read from the PICT file.

Keywords

None.

Examples

To open and read the PICT image file named `foo.pict` in the current directory, store the image in the variable `image1`, and store the color vectors in the variables `R`, `G`, and `B`, enter:

```
READ_PICT, 'foo.pict', image1, R, G, B
```

To load the new color table and display the image, enter:

```
TVLCT, R, G, B  
TV, image1
```

Version History

Introduced: Pre 4.0

See Also

[WRITE_PICT](#), [QUERY_PICT](#)

READ_PNG

The READ_PNG routine reads the image contents of a Portable Network Graphics (PNG) file and returns the image in an IDL variable. If the image contains a palette (see [QUERY_PNG](#)), it can be returned as well in three IDL variables. READ_PNG supports 1, 2, 3 and 4 channel images with channel depths of 8 or 16 bits.

Note

IDL supports version 1.0.5 of the PNG Library.

Note

Only single channel 8-bit images may have palettes. If an 8-bit, single-channel image has index values marked as “transparent,” these can be retrieved as well.

Note

To find information about a potential PNG file before trying to read its data, use the [QUERY_PNG](#) function.

Syntax

```
Result = READ_PNG ( Filename [, R, G, B] [,/ORDER] [, /VERBOSE]
                  [, TRANSPARENT=variable] )
```

or

```
READ_PNG, Filename, Image [, R, G, B] [,/ORDER] [, /VERBOSE]
          [, TRANSPARENT=variable]
```

Return Value

For 8-bit images, *Result* will be a two- or three-dimensional array of type byte. For 16-bit images, *Result* will be of type unsigned integer (UINT).

Arguments

Filename

A scalar string containing the full pathname of the PNG file to read.

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors if a color table exists.

Keywords

ORDER

Set this keyword to indicate that the rows of the image should be read from bottom to top. The rows are read from top to bottom by default. ORDER provides compatibility with PNG files written using versions of IDL prior to IDL 5.4, which wrote PNG files from bottom to top.

VERBOSE

Set this keyword to produce additional diagnostic output during the read operation.

TRANSPARENT

Set this keyword equal to a named variable that will contain an array of pixel index values that are to be treated as “transparent” for the purposes of image display. If there are no transparent values then TRANSPARENT will be set to a long-integer scalar with the value 0.

Examples

Create an RGBA (16-bits/channel) and a Color Indexed (8-bit/channel) image with a palette:

```

rgbdata = UINDGEN(4,320,240)
cidata = BYTSC(LDIST(256))
red = indgen(256)
green = indgen(256)
blue = indgen(256)
WRITE_PNG,'rgb_image.png',rgbdata
WRITE_PNG,'ci_image.png',cidata,red,green,blue
;Query and read the data
names = ['rgb_image.png','ci_image.png','unknown.png']
FOR i=0,N_ELEMENTS(names)-1 DO BEGIN
  ok = QUERY_PNG(names[i],s)
  IF (ok) THEN BEGIN
    HELP,s,/STRUCTURE
    IF (s.HAS_PALETTE) THEN BEGIN
      img = READ_PNG(names[i],rpal,gpal,bpal)
      HELP,img,rpal,gpal,bpal
    
```

```
        ENDIF ELSE BEGIN
            img = READ_PNG(names[i])
            HELP,img
        ENDELSE
    ENDIF ELSE BEGIN
        PRINT,names[i],' is not a PNG file'
    ENDELSE
ENDFOR
END
```

Version History

Introduced: 5.2

See Also

[WRITE_PNG](#), [QUERY_PNG](#)

READ_PPM

The READ_PPM procedure reads the contents of a PGM (gray scale) or PPM (portable pixmap for color) format image file and returns the image in the form of a 2-D byte array (for grayscale images) or a $(3, n, m)$ byte array (for TrueColor images).

Files to be read should adhere to the PGM/PPM standard. The following file types are supported: P2 (graymap ASCII), P5 (graymap RAWBITS), P3 (TrueColor ASCII pixmaps), and P6 (TrueColor RAWBITS pixmaps). Maximum pixel values are limited to 255. Images are always stored with the top row first.

PPM/PGM format is supported by the PBMPLUS toolkit for converting various image formats to and from portable formats, and by the Netpbm package.

This routine is written in the IDL language. Its source code can be found in the file `read_ppm.pro` in the `lib` subdirectory of the IDL distribution.

Note

To find information about a potential PPM file before trying to read its data, use the [QUERY_PPM](#) function.

Syntax

```
READ_PPM, Filename, Image [, MAXVAL=variable]
```

Arguments

Filename

A scalar string specifying the full path name of the PGM or PPM file to read.

Image

A named variable that will contain the image. For grayscale images, *Image* is a 2-D byte array. For TrueColor images, *Image* is a $(3, n, m)$ byte array.

Keywords

MAXVAL

A named variable that will contain the maximum pixel value.

Examples

To open and read the PGM image file named “foo.pgm” in the current directory and store the image in the variable `IMAGE1`:

```
READ_PPM, 'foo.pgm', IMAGE1
```

Version History

Introduced: 4.0

See Also

[WRITE_PPM](#), [QUERY_PPM](#)

READ_SPR

The READ_SPR function reads a row-indexed sparse array. Row-indexed sparse arrays are created using the SPRSIN function and written to a file using the WRITE_SPR function.

This routine is written in the IDL language. Its source code can be found in the file `read_spr.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = READ_SPR(Filename)
```

Return Value

Returns the row-indexed sparse array from the specified file.

Arguments

Filename

A scalar string specifying the name of the file containing a row-indexed sparse array.

Examples

Suppose we have already saved a row-indexed sparse array to a file named `sprs.as`, as described in the documentation for the WRITE_SPR routine. To read the sparse array from the file and store it in a variable `sprs`, use the following command:

```
sprs = READ_SPR('sprs.as')
```

Version History

Introduced: Pre 4.0

See Also

[FULSTR](#), [LINBCG](#), [SPRSAB](#), [SPRSAX](#), [SPRSIN](#), [WRITE_SPR](#)

READ_SRF

The READ_SRF procedure reads the contents of a Sun rasterfile and returns the image and color table vectors (if present) in the form of IDL variables.

READ_SRF only handles 1-, 8-, 24-, and 32-bit rasterfiles of type RT_OLD and RT_STANDARD. See the file `/usr/include/rasterfile.h` for the structure of Sun rasterfiles.

This routine is written in the IDL language. Its source code can be found in the file `read_srf.pro` in the `lib` subdirectory of the IDL distribution.

Note

To find information about a potential SRF file before trying to read its data, use the [QUERY_SRF](#) function.

Syntax

`READ_SRF, Filename, Image [, R, G, B]`

Arguments

Filename

A scalar string containing the name of the rasterfile to read.

Image

A named variable that will contain the 2-D byte array (image).

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors, if the rasterfile contains colormaps.

Keywords

None.

Examples

To open and read the Sun rasterfile named `sun.srf` in the current directory, store the image in the variable `image1`, and store the color vectors in the variables `R`, `G`, and `B`, enter:

```
READ_SRF, 'sun.srf', image1, R, G, B
```

To load the new color table and display the image, enter:

```
TVLCT, R, G, B  
TV, image1
```

Version History

Introduced: Original

See Also

[WRITE_SRF](#), [QUERY_SRF](#)

READ_SYLK

The `READ_SYLK` function reads the contents of a SYLK (Symbolic Link) format spreadsheet data file and returns the contents of the file, or of a cell data range, in an IDL variable.

Note

This routine reads only numeric and string SYLK data. It ignores all spreadsheet and cell formatting information (cell width, text justification, font type, date, time, and monetary notations, etc). Note also that the data in a given cell range must be of the same data type (all integers, for example) in order for the read operation to succeed. See the example below for further information.

This routine is written in the IDL language. Its source code can be found in the file `read_sylk.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = READ_SYLK( File [, /ARRAY] [, /COLMAJOR] [, NCOLS=columns]
  [, NROWS=rows] [, STARTCOL=column] [, STARTROW=row]
  [, /USEDoubles] [, /USELONGS] )
```

Return Value

`READ_SYLK` returns either a vector of structures or a 2-D array containing the spreadsheet cell data. By default, `READ_SYLK` returns a vector of structures, each of which contains the data from one *row* of the table being read. In this case, the individual fields in the structures have the tag names “Col0”, “Col1”, ..., “Coln”. If the `COLMAJOR` keyword is specified, each of the structures returned contains data from one *column* of the table, and the tag names are “Row0”, “Row1”, ..., “Rown”.

Arguments

File

A scalar string specifying the full path name of the SYLK file to read.

Keywords

ARRAY

Set this keyword to return an IDL array rather than a vector of structures. Note that all the data in the cell range specified must be of the same data type to successfully return an array.

COLMAJOR

Set this keyword to create a vector of structures each containing data from a single *column* of the table being read. If you are creating an array rather than a vector of structures (the ARRAY keyword is set), setting COLMAJOR has the same effect as transposing the resulting array.

This keyword should be set when importing spreadsheet data which has column major organization (data stored in columns rather than rows).

NCOLS

Set this keyword to the number of spreadsheet columns to read. If not specified, all of the cell columns found in the file are read.

NROWS

Set this keyword to the number of spreadsheet rows to read. If not specified, all of the cell rows found in the file are read.

STARTCOL

Set this keyword to the first column of spreadsheet cells to read. If not specified, the read operation begins with the first column found in the file (column 0).

STARTROW

Set this keyword to the first row of spreadsheet cells to read. If not specified, the read operation begins with the first row of cells found in the file (row 0).

USEDoubles

Set this keyword to read any floating-point cell data as double-precision rather than the default single-precision.

USELONGS

Set this keyword to read any integer cell data as long integer type rather than the default integer type.

Examples

Suppose the following spreadsheet table, with the upper left cell (value = “Index”) at location (0, 0), has been saved as the SYLK file “file.slk”:

Index	Name	Gender	Platform
1	Beth	F	Windows
2	Lubos	M	UNIX
3	Louis	M	Windows
4	Thierry	M	UNIX

Note that the data format of the title row (*string, string, string, string*) is inconsistent with the following four rows (*int, string, string, string*) in the table. Because of this, it is impossible to read all of the table into a single IDL variable. The following two commands, however, will read all of the data:

```
title = READ_SYLK("file.slk", NROWS = 1)
table = READ_SYLK("file.slk", STARTROW = 1)

;Display the top row of the table.
PRINT, title
```

IDL prints:

```
{ Index Name Gender Platform}
```

Print the table:

```
PRINT, table
```

IDL prints:

```
{1 Beth F Windows}{2 Lubos M UNIX}{3 Louis M Windows}{4 Thierry M
UNIX}
```

To retrieve only the “Name” column:

```
names = READ_SYLK("file.slk", /ARRAY, STARTROW = 1, $
    STARTCOL = 1, NCOLS = 1)
PRINT, names
```

IDL prints:

```
Beth Lubos Louis Thierry
```

To retrieve the “Name” column in column format:

```
namescol = READ_SYLK("file.slk", /ARRAY, /COLMAJOR, $
    STARTROW = 1, STARTCOL = 1, NCOLS = 1)
PRINT, namescol
```

IDL prints:

```
Beth
Lubos
Louis
Thierry
```

Version History

Introduced: 4.0

See Also

[WRITE_SYLK](#)

READ_TIFF

The READ_TIFF function reads single or multi-channel images from TIFF format files and returns the image and color table vectors in the form of IDL variables.

Note

To find information about a potential TIFF file before trying to read its data, use the [QUERY_TIFF](#) function. The obsolete routine TIFF_DUMP may also be used to examine the structure and tags of a TIFF file.

Syntax

```
Result = READ_TIFF( Filename [, R, G, B] [, CHANNELS=scalar or vector]
  [, DOT_RANGE=variable] [, GEOTIFF=variable] [, ICC_PROFILE=variable]
  [, IMAGE_INDEX=value] [, INTERLEAVE={0 | 1 | 2}]
  [, ORIENTATION=variable] [, PHOTOSHOP=variable]
  [, PLANARCONFIG=variable] [, SUB_RECT=[x, y, width, height]]
  [, /VERBOSE] )
```

Return Value

READ_TIFF returns a byte, unsigned integer, long, or float array (based on the data format in the TIFF file) containing the image data. The dimensions of the *Result* are [*Columns*, *Rows*] for single-channel images, or [*Channels*, *Columns*, *Rows*] for multi-channel images, unless a different type of interleaving is specified with the INTERLEAVE keyword.

For 1-bit (bilevel) images, the image values are 0 or 1. For 4-bit grayscale images the image values are in the range 0 to 15.

RGB images are a special case of multi-channel images, and contain three channels. Most TIFF readers and writers can handle only images with one or three channels.

As a special case, for three-channel TIFF image files that are stored in planar interleave format, and if four parameters are provided, READ_TIFF returns the integer value zero, sets the variable defined by the PLANARCONFIG keyword to 2, and returns three separate images in the variables defined by the *R*, *G*, and *B* arguments.

Arguments

Filename

A scalar string specifying the full pathname of the TIFF file to read.

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors of the color table from the file if one exists. If the TIFF file is written as a three-channel image, interleaved by plane, and the R, G, and B parameters are present, the three channels of the image are returned in the R, G, and B variables.

Keywords

CHANNELS

Set this keyword to a scalar or vector giving the channel numbers to be returned for a multi-channel image, starting with zero. The default is to return all of the channels. This keyword is ignored for single-channel images, or for three-channel planar-interleaved images when the *R*, *G*, and *B* arguments are specified.

DOT_RANGE

Set this keyword to a named variable in which to return a two-element integer array containing the TIFF DotRange tag value. If DotRange is not defined in the TIFF file, then a scalar zero will be returned instead. The DotRange tag is typically only present in CMYK TIFF files. DOT_RANGE[0] gives the image value that corresponds to a 0% dot, while DOT_RANGE[1] gives the image value that corresponds to a 100% dot.

GEOTIFF

Returns an anonymous structure containing one field for each of the GeoTIFF tags and keys found in the file. If no GeoTIFF information is present in the file, the returned variable is undefined.

The GeoTIFF structure is formed using fields named from the following table.

Anonymous Structure Field Name	IDL Datatype
TAGS:	
"MODELPIXELSCALETAG"	DOUBLE[3]
"MODELTRANSFORMATIONTAG"	DOUBLE[4,4]
"MODELTIPOINTTAG"	DOUBLE[6,*]
KEYS:	
"GTMODELTYPEGEOKEY"	INT
"GTRASTERTYPEGEOKEY"	INT
"GTCITATIONGEOKEY"	STRING
"GEOGRAPHICTYPEGEOKEY"	INT
"GEOGCITATIONGEOKEY"	STRING
"GEOGGEODETICDATUMGEOKEY"	INT
"GEOGPRIMEMERIDIANGEOKEY"	INT
"GEOGLINEARUNITSGEOKEY"	INT
"GEOGLINEARUNITSIZEGEOKEY"	DOUBLE
"GEOGANGULARUNITSGEOKEY"	INT
"GEOGANGULARUNITSIZEGEOKEY"	DOUBLE
"GEOGELLIPSOIDGEOKEY"	INT
"GEOGSEMIMAJORAXISGEOKEY"	DOUBLE
"GEOGSEMIMINORAXISGEOKEY"	DOUBLE
"GEOGINVFLATTENINGGEOKEY"	DOUBLE
"GEOGAZIMUTHUNITSGEOKEY"	INT
"GEOGPRIMEMERIDIANLONGGEOKEY"	DOUBLE
"PROJECTEDCSTYPEGEOKEY"	INT

Table 3-100: GEOTIFF Structures

Anonymous Structure Field Name	IDLDatatype
"PCSCITATIONGEOKEY"	STRING
"PROJECTIONGEOKEY"	INT
"PROJCOORDTRANSGEOKEY"	INT
"PROJLINEARUNITSGEOKEY"	INT
"PROJLINEARUNITSIZEGEOKEY"	DOUBLE
"PROJSTDPARALLEL1GEOKEY"	DOUBLE
"PROJSTDPARALLEL2GEOKEY"	DOUBLE
"PROJNATORIGINLONGGEOKEY"	DOUBLE
"PROJNATORIGINLATGEOKEY"	DOUBLE
"PROJFALSEEASTINGGEOKEY"	DOUBLE
"PROJFALSENORTHINGGEOKEY"	DOUBLE
"PROJFALSEORIGINLONGGEOKEY"	DOUBLE
"PROJFALSEORIGINLATGEOKEY"	DOUBLE
"PROJFALSEORIGINEASTINGGEOKEY"	DOUBLE
"PROJFALSEORIGINNORTHINGGEOKEY"	DOUBLE
"PROJCENTERLONGGEOKEY"	DOUBLE
"PROJCENTERLATGEOKEY"	DOUBLE
"PROJCENTEREASTINGGEOKEY"	DOUBLE
"PROJCENTERNORTHINGGEOKEY"	DOUBLE
"PROJSCALEATNATORINGGEOKEY"	DOUBLE
"PROJSCALEATCENTERGEOKEY"	DOUBLE
"PROJAZIMUTHANGLEGEOKEY"	DOUBLE
"PROJSTRAIGHTVERTPOLELONGGEOKEY"	DOUBLE
"VERTICALCSTYPEGEOKEY"	INT

Table 3-100: GEOTIFF Structures (Continued)

Anonymous Structure Field Name	IDLDatatype
"VERTICALCITATIONGEOKEY"	STRING
"VERTICALDATUMGEOKEY"	INT
"VERTICALUNITSGEOKEY"	INT

Table 3-100: GEOTIFF Structures (Continued)

Note

If a GeoTIFF key appears multiple times in a file, only the value for the first instance of the key is returned.

IMAGE_INDEX

Selects the image number within the file to be read (see [QUERY_TIFF](#) to determine the number of images in the file).

ICC_PROFILE

Set this keyword to a named variable in which to return a byte array containing the TIFF ICC_PROFILE tag value. If the ICC_PROFILE tag is not contained in the TIFF file then a scalar zero will be returned instead. The ICC_PROFILE array is returned as an opaque array of byte values which the user can then pass to the WRITE_TIFF routine.

Note

The ICC_PROFILE tag is returned as an opaque array of byte values. It is assumed that the tag will be read from an existing file, and written out to a new file without modification.

INTERLEAVE

For multi-channel images, set this keyword to one of the following values to force the *Result* to have a specific interleaving, regardless of the type of interleaving in the file being read:

Value	Description
0	Pixel interleaved: <i>Result</i> will have dimensions [<i>Channels</i> , <i>Columns</i> , <i>Rows</i>].
1	Scanline (row) interleaved: <i>Result</i> will have dimensions [<i>Columns</i> , <i>Channels</i> , <i>Rows</i>].
2	Planar interleaved: <i>Result</i> will have dimensions [<i>Columns</i> , <i>Rows</i> , <i>Channels</i>].

Table 3-101: INTERLEAVE Keyword Values

If this keyword is not specified, the *Result* will always be pixel interleaved, regardless of the type of interleaving in the file being read. For files stored in planar-interleave format, this keyword is ignored if the *R*, *G*, and *B* arguments are specified.

ORIENTATION

Set this keyword to a named variable that will contain the orientation value from the TIFF file. Possible return values are:

Value	Description
0	Column 0 represents the left-hand side, and row 0 represents the bottom (same as 4)
1	Column 0 represents the left-hand side, and row 0 represents the top.
2	Column 0 represents the right-hand side, and row 0 represents the top.
3	Column 0 represents the right-hand side, and row 0 represents the bottom.

Table 3-102: ORIENTATION Keyword Values

Value	Description
4	Column 0 represents the left-hand side, and row 0 represents the bottom (same as 0)
5	Column 0 represents the top, and row 0 represents the left-hand side.
6	Column 0 represents the top, and row 0 represents the right-hand side.
7	Column 0 represents the bottom, and row 0 represents the right-hand side.
8	Column 0 represents the bottom, and row 0 represents the left-hand side.

Table 3-102: ORIENTATION Keyword Values (Continued)

If an orientation value does not appear in the TIFF file, an orientation of 0 is returned.

PHOTOSHOP

Set this keyword to a named variable in which to return a byte array containing the TIFF PHOTOSHOP tag value. If the PHOTOSHOP tag is not contained in the TIFF file, then a scalar zero will be returned instead. The PHOTOSHOP array is returned as an opaque array of byte values which the user can then pass to the WRITE_TIFF routine.

Note

The PHOTOSHOP tag is returned as an opaque array of byte values. It is assumed that the tag will be read from an existing file, and written out to a new file without modification.

PLANARCONFIG

Set this keyword to a named variable that will contain the interleave parameter for the TIFF file. This parameter is returned as 1 for TIFF files that are GrayScale, Palette, or interleaved by pixel. This parameter is returned as 2 for multi-channel TIFF files interleaved by image.

SUB_RECT

Set this keyword to a four-element array, [*x*, *y*, *width*, *height*], that specifies a rectangular region within the file to extract. Only the rectangular portion of the image selected by this keyword is read and returned. The rectangle is measured in pixels from the lower left corner (right hand coordinate system).

Tip

For tiled TIFF images, use the `TILE_SIZE` tag returned by `QUERY_TIFF` to determine the optimal sizes for the `SUB_RECT` keyword.

VERBOSE

Produce additional diagnostic output during the read.

Obsolete Keywords

The following keywords are obsolete:

- ORDER
- UNSIGNED

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

Example 1

Read the file `my.tif` in the current directory into the variable `image`, and save the color tables in the variables, `R`, `G`, and `B` by entering:

```
image = READ_TIFF('my.tif', R, G, B)
```

To view the image, load the new color table and display the image by entering:

```
TVLCT, R, G, B
TV, image
```

Example 2

Write and read a multi-image TIFF file. The first image is a 16-bit single-channel image stored using compression. The second image is an RGB image stored using 32-bits/channel uncompressed.

```
; Write the image data:
data = FIX(DIST(256))
```

```

rgbdata = LONARR(3,320,240)
WRITE_TIFF,'multi.tif',data,COMPRESSION=1,/SHORT
WRITE_TIFF,'multi.tif',rgbdata,/LONG,/APPEND

; Read the image data back:
ok = QUERY_TIFF('multi.tif',s)
IF (ok) THEN BEGIN
FOR i=0,s.NUM_IMAGES-1 DO BEGIN
imp = QUERY_TIFF('multi.tif',t,IMAGE_INDEX=i)
img = READ_TIFF('multi.tif',IMAGE_INDEX=i)
HELP,t,/STRUCTURE
HELP,img
ENDFOR
ENDIF

```

Example 3

Write and read a multi-channel image:

```

data = LINDGEN(10, 256, 256) ; 10 channels

; Write the image data:
WRITE_TIFF, 'multichannel.tif', data, /LONG

; Read back only channels [0,2,4,6,8], using planar-interleaving
img = READ_TIFF('multichannel.tif', CHANNELS=[0,2,4,6,8], $
INTERLEAVE=2)

HELP, img

```

IDL prints:

```

IMG      LONG      = Array[256, 256, 5]

```

Version History

Introduced: 5.0

DOT_RANGE, ICC_PROFILE, and PHOTOSHOP keywords added: 6.1

See Also

[WRITE_TIFF](#), [QUERY_TIFF](#)

READ_WAV

The READ_WAV function reads the audio stream from the named .WAV file. Optionally, it can return the sampling rate of the audio stream.

Syntax

```
Result = READ_WAV ( Filename [, Rate] )
```

Return Value

In the case of a single channel stream, the returned variable is a BYTE or INT (depending on the number of bits per sample) one-dimensional array. In the case of a file with multiple channels, a similar two-dimensional array is returned, with the leading dimension being the channel number.

Arguments

Filename

A scalar string containing the full pathname of the .WAV file to read.

Rate

Returns an IDL long containing the sampling rate of the stream in samples per second.

Keywords

None.

Version History

Introduced: 5.3

READ_WAVE

The READ_WAVE procedure reads a .wave or .bwave file created by the Wavefront Advanced Data Visualizer into a series of IDL variables.

Note

READ_WAVE only preserves the structure of the variables if they are regularly gridded.

This routine is written in the IDL language. Its source code can be found in the file `read_wave.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

READ_WAVE, *File*, *Variables*, *Names*, *Dimensions* [, MESHNAMES=*variable*]

Arguments

File

A scalar string containing the name of the Wavefront file to read.

Variables

A named variable that will contain a block of the variables contained in the wavefront file. Since each variable in a wavefront file can have more than one field (for instance, a vector variable has 3 fields), the fields of each variable make up the major index into the variable block. For instance, if a Wavefront file had one scalar variable and one vector variable, the scalar would be extracted as follows:

```
scalar_variable = variables[0,*,*,*]
```

and the vector variable would be extracted as follows:

```
vector_variable = variables[1:3,*,*,*]
```

To find the dimensions of the returned variable, see the description of the *Dimensions* argument.

Names

A named variable that will contain the string names of each variable contained in the file.

Dimensions

A named variable that will contain a long array describing how many fields in the large returned variable block each variable occupies. In the above example of one scalar variable followed by a vector variable, the dimension variable would be `[1, 3]`.

This indicates that the first field of the returned variable block would be the scalar variable and the following 3 fields would comprise the vector variable.

Keywords

MESHNAMES

Set this keyword to a named variable that will contain the name of the mesh used in the Wavefront file for each variable.

Version History

Introduced: Pre 4.0

See Also

[WRITE_WAVE](#)

READ_X11_BITMAP

The `READ_X11_BITMAP` procedure reads bitmaps stored in the X Windows X11 format. The X Windows `bitmap` program produces a C header file containing the definition of a bitmap produced by that program. This procedure reads such a file and creates an IDL byte array containing the bitmap. It is used primarily to read bitmaps to be used as IDL widget button labels.

This routine is written in the IDL language. Its source code can be found in the file `read_x11_bitmap.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
READ_X11_BITMAP, File, Bitmap [, X, Y] [, /EXPAND_TO_BYTES]
```

Arguments

File

A scalar string containing the name of the file containing the bitmap.

Bitmap

A named variable that will contain the bitmap. This variable is returned as a byte array.

X

A named variable that will contain the width of the bitmap.

Y

A named variable that will contain the height of the bitmap.

Keywords

EXPAND_TO_BYTES

Set this keyword to instruct `READ_X11_BITMAP` to return a 2-D array which has one bit per byte (0 for a 0 bit, 255 for a 1 bit) instead.

Examples

To open and read the X11 bitmap file named `my.x11` in the current directory, store the bitmap in the variable `bitmap1`, and the width and height in the variables `x` and `y`, enter:

```
READ_X11_BITMAP, 'my.x11', bitmap1, x, y
```

To display the new bitmap, enter:

```
READ_X11_BITMAP, 'my.x11', image, /EXPAND_TO_BYTES  
TV, image, /ORDER
```

Version History

Introduced: Pre 4.0

See Also

[READ_XWD](#)

READ_XWD

The `READ_XWD` function reads the contents of a file created by the `xwd` (X Windows Dump) command and returns the image and color table vectors in the form of IDL variables.

Note

This function is intended to be used only on files containing 8-bit pixmaps created with `xwd` version 6 or later.

This routine is written in the IDL language. Its source code can be found in the file `read_xwd.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = `READ_XWD`(*Filename*[, *R*, *G*, *B*])

Return Value

`READ_XWD` returns a 2-D byte array containing the image. If the file cannot be open or read, the return value is zero.

Arguments

Filename

A scalar string specifying the full pathname of the XWD file to read.

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors, if the XWD file contains color tables.

Keywords

None.

Examples

To open and read the X Windows Dump file named `my.xwd` in the current directory, store the image in the variable `image1`, and store the color vectors in the variables, `R`, `G`, and `B`, enter:

```
image1 = READ_XWD('my.xwd', R, G, B)
```

To load the new color table and display the image, enter:

```
TVLCT, R, G, B  
TV, image1
```

Version History

Introduced: Pre 4.0

See Also

[READ_X11_BITMAP](#)

READS

The READS procedure performs formatted input from a string variable and writes the results into one or more output variables. This procedure differs from the READ procedure only in that the input comes from memory instead of a file.

This routine is useful when you need to examine the format of a data file before reading the information it contains. Each line of the file can be read into a string using READF. Then the components of that line can be read into variables using READS.

Syntax

```
READS, Input, Var1, ..., Varn [, AM_PM=[string, string]]
    [, DAYS_OF_WEEK=string_array{7 names}] [, FORMAT=value]
    [, MONTHS=string_array{12 names}]
```

Arguments

Input

The string variable from which the input is taken. If the supplied argument is not a string, it is automatically converted. The argument can be scalar or array. If *Input* is an array, the individual string elements are treated as successive lines of input.

*Var*_{*i*}

The named variables to receive the input.

Note

If the variable specified for the *Var*_{*i*} argument has not been previously defined, the input data is assumed to be of type float, and the variable will be cast as a float.

Keywords

AM_PM

Supplies a string array of 2 names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the FORMAT keyword.

DAYS_OF_WEEK

Supplies a string array of 7 names to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the FORMAT keyword.

FORMAT

If FORMAT is not specified, IDL uses its default rules for formatting the input. FORMAT allows the format of the input to be specified in precise detail, using a FORTRAN-style specification. See [“Using Explicitly Formatted Input/Output”](#) in Chapter 11 of the *Building IDL Applications* manual.

MONTHS

Supplies a string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the FORMAT keyword.

Version History

Introduced: Pre 4.0

See Also

[READ/READF, READU](#)

READU

The READU procedure reads unformatted binary data from a file into IDL variables. READU transfers data directly with no processing of any kind performed on the data.

Syntax

```
READU, Unit, Var1, ..., Varn [, TRANSFER_COUNT=variable]
```

Arguments

Unit

The IDL file unit from which input is taken.

Var_i

Named variables to receive the data. For non-string variables, the number of bytes required for Var are read. When READU is used with a variable of type string, IDL reads exactly the number of bytes contained in the existing string. For example, to read a 5-character string, enter:

```
temp = '12345'  
READU, unit, temp
```

Keywords

TRANSFER_COUNT

Set this keyword to a named variable in which to return the number of elements transferred by the input operation. Note that the number of elements is not the same as the number of bytes (except in the case where the data type being transferred is bytes). For example, transferring 256 floating-point numbers yields a transfer count of 256, not 1024 (the number of bytes transferred).

This keyword is useful with files opened with the RAWIO keyword to the OPEN routines. Normally, attempting to read more data than is available from a file causes the unfilled space to be zeroed and an error to be issued. This does not happen with files opened with the RAWIO keyword. Instead, the programmer must keep track of the transfer count.

Obsolete Keywords

The following keywords are obsolete:

- KEY_ID
- KEY_MATCH
- KEY_VALUE

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

The following commands can be used to open the IDL distribution file `people.dat` and read an image from that file:

```
; Open the file for reading as file unit 1:
OPENR, 1, FILEPATH('people.dat', SUBDIR = ['examples', 'data'])

; The image is a 192 by 192 byte array, so make B that size:
B = BYTARR(192, 192)

; Read the data into B:
READU, 1, B

; Close the file:
CLOSE, 1

; Display the image:
TV, B
```

Version History

Introduced: Original

See Also

[READ/READF](#), [READS](#), [WRITEU](#), Chapter 11, “Files and Input/Output” in the *Building IDL Applications* manual, “Unformatted Input/Output with Structures” in Chapter 9 of the *Building IDL Applications* manual.

REAL_PART

The `REAL_PART` function returns the real part of its complex-valued argument.

Syntax

Result = `REAL_PART(Z)`

Return Value

If the complex-valued argument is double-precision, the result will be double-precision, otherwise the result will be single-precision floating-point. If the argument is not complex, then the result will be double-precision if the argument is double-precision, otherwise the result will be single-precision.

Arguments

Z

A scalar or array for which the real part is desired. Z may be of any numeric type.

Examples

The following example demonstrates how you can use `REAL_PART` to obtain the real parts of an array of complex variables.

```
; Create an array of complex values:
cValues = COMPLEX([1, 2, 3],[4, 5, 6])

; Print just the real parts of each element in cValues:
PRINT, REAL_PART(cValues)
```

IDL prints:

```
1.00000  2.00000  3.00000
```

Version History

Introduced: 5.5

See Also

[COMPLEX](#), [DCOMPLEX](#), [IMAGINARY](#)

REBIN

The REBIN function resizes a vector or array to dimensions given by the parameters D_i . The supplied dimensions must be integral multiples or factors of the original dimension. The expansion or compression of each dimension is independent of the others, so that each dimension can be expanded or compressed by a different value.

If the dimensions of the desired result are not integer multiples of the original dimensions, use the CONGRID function.

Syntax

```
Result = REBIN( Array, D1 [, ..., D8] [, /SAMPLE] )
```

Return Value

Returns the resized array or vector of the specified dimensions.

Arguments

Array

The array to be resampled. *Array* can be of any basic type except complex or string.

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Note

The dimensions of the resulting resampled array must be integer multiples or factors of the corresponding original dimensions.

Keywords

SAMPLE

Normally, REBIN uses bilinear interpolation when magnifying and neighborhood averaging when minifying. Set the SAMPLE keyword to use nearest neighbor

sampling for both magnification and minification. Bilinear interpolation gives higher quality results but requires more time.

Rules Used by REBIN

Assume the original vector X has n elements and the result is to have m elements.

Let $f = n/m$, the ratio of the size of the original vector, X to the size of the result. $1/f$ must be an integer if $n < m$ (expansion). f must be an integer if compressing, ($n > m$). The various resizing options can be described as:

- Expansion, $n < m$, SAMPLE = 0: $Y_i = F(X, f \cdot i) \quad i = 0, 1, \dots, m-1$

The linear interpolation function, $F(X, p)$ that interpolates X at location p , is defined as:

$$F(X, p) = \begin{cases} X_{\lfloor p \rfloor} + (p - \lfloor p \rfloor) \cdot (X_{\lfloor p \rfloor + 1} - X_{\lfloor p \rfloor}) & \text{if } p < n - 1 \\ X_{\lfloor p \rfloor} & \text{if } p \geq n - 1 \end{cases}$$

- Expansion, $n < m$, SAMPLE = 1:

$$Y_i = X_{\lfloor fi \rfloor}$$

- Compression, $n > m$, SAMPLE = 0:

$$Y_i = (1/f) \sum_{j=fi}^{f(i+1)-1} X_j$$

- Compression, $n > m$, SAMPLE = 1:

$$Y_i = X_{\lfloor fi \rfloor}$$

- No change, $n = m$: $Y_i = X_i$

Endpoint Effects When Expanding

When expanding an array, REBIN *interpolates*, it never *extrapolates*. Each of the $n-1$ intervals in the n -element input array produces m/n interpolates in the m -element output array. The last m/n points of the result are obtained by duplicating element $n-1$ of the input array because their interpolates would lie outside the input array.

For example

```
; A four point vector:
A = [0, 10, 20, 30]

; Expand by a factor of 3:
B = REBIN(A, 12)

PRINT, B
```

IDL prints:

```
0 3 6 10 13 16 20 23 26 30 30 30
```

Note that the last element is repeated three times. If this effect is undesirable, use the INTERPOLATE function. For example, to produce 12 equally spaced interpolates from the interval 0 to 30:

```
B = INTERPOLATE(A, 3./11. * FINDGEN(12))
PRINT, B
```

IDL prints:

```
0 2 5 8 10 13 16 19 21 24 27 30
```

Here, the sampling ratio is $(n - 1)/(m - 1)$.

Examples

Create and display a simple image by entering:

```
D = SIN(DIST(50)/4) & TVSCL, D
```

Resize the image to be 5 times its original size and display the result by entering:

```
D = REBIN(D, 250, 250) & TVSCL, D
```

Version History

Introduced: Original

See Also

[CONGRID](#)

RECALL_COMMANDS

The `RECALL_COMMANDS` function returns a string array containing the entries in IDL's command recall buffer.

Syntax

Result = `RECALL_COMMANDS()`

Return Value

The size of the returned array is the size of recall buffer, even if fewer than commands have been entered (any “empty” buffer entries will contain null strings). The default size of the command recall buffer is 20 lines. (See “[!EDIT_INPUT](#)” on page 4367 for more information about the command recall buffer.)

Element zero of the returned array contains the most recent command.

Arguments

None.

Keywords

None.

Version History

Introduced: 5.0

RECON3

The RECON3 function can reconstruct a three-dimensional data array from two or more images (or projections) of an object. For example, if you placed a dark object in front of a white background and then photographed it three times (each time rotating the object a known amount) then these three images could be used with RECON3 to approximate a 3-D volumetric representation of the object. RECON3 also works with translucent projections of an object. RECON3 returns a 3-D byte array.

This routine is written in the IDL language. Its source code can be found in the file `recon3.pro` in the `lib` subdirectory of the IDL distribution.

Using RECON3

Images used in reconstruction should show strong light/dark contrast between the object and the background. If the images contain low (dark) values where the object is and high (bright) values where the object isn't, the `MODE` keyword should be set to `+1` and the returned volume will have low values where the object is, and high values where the object isn't. If the images contain high (bright) values where the object is and low (dark) values where the object isn't, the `MODE` keyword should be set to `-1` and the returned volume will have high values where the object is, and low values where the object isn't.

In general, the object must be `CONVEX` for a good reconstruction to be possible. Concave regions are not easily reconstructed. An empty coffee cup, for example, would be reconstructed as if it were full.

The more images the better. Images from many different angles will improve the quality of the reconstruction. It is also important to supply images that are parallel and perpendicular to any axes of symmetry. Using the coffee cup as an example, at least one image should be looking through the opening in the handle. Telephoto images are also better for reconstruction purposes than wide angle images.

Syntax

```
Result = RECON3( Images, Obj_Rot, Obj_Pos, Focal, Dist, Vol_Pos, Img_Ref,  
               Img_Mag, Vol_Size [, /CUBIC] [, MISSING=value] [, MODE=value] [, /QUIET]  
               )
```

Return Value

Returns a 3-D data array.

Arguments

Images

A 3-D array containing the images to use to reconstruct the volume. Execution time increases linearly with more images. *Images* must be an 8-bit (byte) array with dimensions (x, y, n) where x is the horizontal image dimension, y is the vertical image dimension, and n is the number of images. Note that n must be at least 2.

Obj_Rot

A $3 \times n$ floating-point array specifying the amount the object is rotated to make it appear as it does in each image. The object is first rotated about the X axis, then about the Y axis, and finally about the Z axis (with the object's reference point at the origin). *Obj_Rot*[0, *] is the X rotation for each image, *Obj_Rot*[1, *] is the Y rotation, and *Obj_Rot*[2, *] is the Z rotation.

Obj_Pos

A $3 \times n$ floating-point array specifying the position of the object's reference point relative to the camera lens. The camera lens is located at the coordinate origin and points in the negative Z direction (the view up vector points in the positive Y direction). *Obj_Pos* should be expressed in this coordinate system. *Obj_Pos*[0, *] is the X position for each image, *Obj_Pos*[1, *] is the Y position, and *Obj_Pos*[2, *] is the Z position. All the values in *Obj_Pos*[2, *] should be less than zero. Note that the values for *Obj_Pos*, *Focal*, *Dist*, and *Vol_Pos* should all be expressed in the same units (mm, cm, m, in, ft, etc.).

Focal

An n -element floating-point array specifying the focal length of the lens for each image. Focal may be set to zero to indicate a parallel image projection (infinite focal length).

Dist

An n -element floating-point array specifying the distance from the camera lens to the image plane (film) for each image. *Dist* should be greater than *Focal*.

Vol_Pos

A 3×2 floating-point array specifying the two opposite corners of a cube that surrounds the object. *Vol_Pos* should be expressed in the object's coordinate system

relative to the object's reference point. *Vol_Pos*[*, 0] specifies one corner and *Vol_Pos*[*, 1] specifies the opposite corner.

Img_Ref

A 2 x *n* integer or floating-point array that specifies the pixel location at which the object's reference point appears in each of the images. *Img_Ref*[0, *] is the X coordinate for each image and *Img_Ref*[1, *] is the Y coordinate.

Img_Mag

A 2 x *n* integer or floating-point array that specifies the magnification factor for each image. This number is actually the length (in pixels) that a test object would appear in an image if it were *n* units long and *n* units distant from the camera lens. *Img_Mag*[0, *] is the X dimension (in pixels) of a test object for each image, and *Img_Mag*[1, *] is the Y dimension. All elements in *Img_Mag* should be greater than or equal to 1.

Vol_Size

A 3-element integer or floating-point array that specifies the size of the 3-D byte array to return. Execution time (and resolution) increases exponentially with larger values for *Vol_Size*. *Vol_Size*[0] specifies the X dimension of the volume, *Vol_Size*[1] specifies the Y dimension, and *Vol_Size*[2] specifies the Z dimension.

Keywords

CUBIC

Set this keyword to use cubic interpolation. The default is to use tri-linear interpolation, which is slightly faster.

MISSING

Set this keyword equal to a byte value for cells in the 3-D volume that do not map to any of the supplied images. The value of MISSING is passed to the INTERPOLATE function. The default value is zero.

MODE

Set this keyword to a value less than zero to define each cell in the 3-D volume as the *minimum* of the corresponding pixels in the images. Set MODE to a value greater than zero to define each cell in the 3-D volume as the *maximum* of the corresponding pixels in the images. If MODE is set equal to zero then each cell in the 3-D volume is defined as the *average* of the corresponding pixels in the images.

MODE should usually be set to -1 when the images contain a bright object in front of a dark background or to +1 when the images contain a dark object in front of a light background. Setting MODE=0 (the default) requires more memory since the volume array must temporarily be kept as an integer array instead of a byte array.

QUIET

Set this keyword to suppress the output of informational messages when the processing of each image is completed.

Examples

Assumptions for this example:

- The object's major axis is parallel to the Z axis.
- The object's reference point is at its center.
- The camera lens is pointed directly at this reference point.
- The reference point is 5000 mm in front of the camera lens.
- The focal length of the camera lens is 200 mm.

If the camera is focused on the reference point, then the distance from the lens to the camera's image plane must be:

$$\text{dist} = (d * f) / (d - f) = (5000 * 200) / (5000 - 200) = (1000000 / 4800) = 208.333 \text{ mm}$$

The object is roughly 600 mm wide and 600 mm high. The reference point appears in the exact center of each image.

If the object is 600 mm high and 5000 mm distant from the camera lens, then the object image height must be:

$$\text{hi} = (h * f) / (d - f) = (600 * 200) / (5000 - 200) = (120000 / 4800) = 25.0 \text{ mm}$$

The object image appears 200 pixels high so the final magnification factor is:

$$\text{img_mag} = (200 / 25) = 8.0$$

From these assumptions, we can set up the following reconstruction:

```

; First, define the variables:
imgx = 256
imgy = 256
frames = 3
images = BYTARR(imgx, imgy, frames)
obj_rot = Fltarr(3, frames)
obj_pos = Fltarr(3, frames)
focal = Fltarr(frames)

```

```

dist = Fltarr(frames)
vol_pos = Fltarr(3, 2)
img_ref = Fltarr(2, frames)
img_mag = Fltarr(2, frames)
vol_size = [40, 40, 40]

; The object is 5000 mm directly in front of the camera:
obj_pos[0, *] = 0.0
obj_pos[1, *] = 0.0
obj_pos[2, *] = -5000.0

; The focal length of the lens is constant for all the images:
focal[*] = 200.0

; The distance from the lens to the image plane is also constant:
dist[*] = 208.333

; The cube surrounding the object is 600 mm x 600 mm:
vol_pos[* , 0] = [-300.0, -300.0, -300.0]
vol_pos[* , 1] = [ 300.0, 300.0, 300.0]

; The image reference point appears at the center of all the
; images:
img_ref[0, *] = imgx / 2
img_ref[1, *] = imgy / 2

; The image magnification factor is constant for all images.
; (The images haven't been cropped or resized):
img_mag[* ,*] = 8.0

; Only the object rotation changes from one image to the next.
; Note that the object is rotated about the X axis first, then Y,
; and then Z. Create some fake images for this example:
images[30:160, 20:230, 0] = 255
images[110:180, 160:180, 0] = 180
obj_rot[* , 0] = [-90.0, 0.0, 0.0]
images[70:140, 100:130, 1] = 255
obj_rot[* , 1] = [-70.0, 75.0, 0.0]
images[10:140, 70:170, 2] = 255
images[80:90, 170:240, 2] = 150
obj_rot[* , 2] = [-130.0, 215.0, 0.0]

; Reconstruct the volume:
vol = RECON3(images, obj_rot, obj_pos, focal, dist, $
    vol_pos, img_ref, img_mag, vol_size, Missing=255B, Mode=(-1))

; Display the volume:
shade_volume, vol, 8, v, p
scale3, xrange=[0,40], yrange=[0,40], zrange=[0,40]

```

```
image = polyshade(v, p, /t3d, xs=400, ys=400)  
tvsc1, image
```

Version History

Introduced: Pre 4.0

See Also

[POLYSHADE](#), [SHADE_VOLUME](#), [VOXEL_PROJ](#)

REDUCE_COLORS

The REDUCE_COLORS procedure reduces the number of colors used in an image by eliminating pixel values without members.

The pixel distribution histogram is obtained and the WHERE function is used to find bins with non-zero values. Next, a lookup table is made where table[old_pixel_value] contains new_pixel_value, and is then applied to the image.

This routine is written in the IDL language. Its source code can be found in the file reduce_colors.pro in the lib subdirectory of the IDL distribution.

Syntax

REDUCE_COLORS, *Image*, *Values*

Arguments

Image

On input, a variable that contains the original image array. On output, this variable contains the color-reduced image array, writing over the original.

Values

A named variable that, on output, contains a vector of non-zero pixel values. If *Image* contains pixel values from 0 to M, *Values* will be an M+1 element vector containing the mapping from the old values to the new. *Values[i]* contains the new color index of old pixel index *i*.

Keywords

None.

Examples

To reduce the number of colors and display an image with the original color tables R, G, B enter the commands:

```
REDUCE_COLORS, image, v
TVLCT, R[V], G[V], B[V]
```

Version History

Introduced: Pre 4.0

See Also

[COLOR_QUAN](#)

REFORM

The REFORM function changes the dimensions of an array without changing the total number of elements.

Syntax

Result = REFORM(*Array*, *D*₁ [, ..., *D*₈] [, /OVERWRITE])

Return Value

If no dimensions are specified, REFORM returns a copy of *Array* with all dimensions of size 1 removed. If dimensions are specified, the result is given those dimensions. Only the dimensions of *Array* are changed – the actual data remains unmodified.

Arguments

Array

The array to have its dimensions modified.

*D*_{*i*}

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. *Array* must have the same number of elements as specified by the product of the new dimensions.

Keywords

OVERWRITE

Set this keyword to cause the specified dimensions to overwrite the present dimensions of the *Array* parameter. No data are copied, only the internal array descriptor is changed. The result of the function, in this case, is the *Array* parameter with its newly-modified dimensions. For example, to change the dimensions of the variable *a*, without moving data, enter:

```
a = REFORM(a, n1, n2, /OVERWRITE)
```

Examples

REFORM can be used to remove “degenerate” leading dimensions of size one. Such dimensions can appear when a subarray is extracted from an array with more dimensions. For example

```
; a is a 3-dimensional array:
a = INTARR(10,10,10)

; Extract a "slice" from a:
b = a[5,*,*]

; Use HELP to show what REFORM does:
HELP, b, REFORM(b)
```

Executing the above statements produces the output:

```
B          INT = Array[1, 10, 10]
<Expression> INT = Array[10, 10]
```

The statements:

```
b = REFORM(a,200,5)
b = REFORM(a,[200,5])
```

have identical effect. They create a new array, b, with dimensions of (200, 5), from a.

Version History

Introduced: Original

See Also

[REVERSE](#), [ROT](#), [ROTATE](#), [TRANSPOSE](#)

REGION_GROW

The REGION_GROW function performs region growing for a given region within an N-dimensional array by finding all pixels within the array that are connected neighbors to the region pixels and that fall within provided constraints. The constraints are specified either as a threshold range (a minimum and maximum pixel value) or as a multiple of the standard deviation of the region pixel values. If the threshold is used (this is the default), the region is grown to include all connected neighboring pixels that fall within the given threshold range. If the standard deviation multiplier is used, the region is grown to include all connected neighboring pixels that fall within the range of the mean (of the region's pixel values) plus or minus the given multiplier times the sample standard deviation.

Syntax

```
Result = REGION_GROW(Array, ROIPixels [, /ALL_NEIGHBORS] [, /NAN]
    [, STDDEV_MULTIPLIER=value | THRESHOLD=[min,max] ] )
```

Return Value

REGION_GROW returns the vector of array indices that represent pixels within the grown region. The grown region will not include pixels at the edges of the input array. If no pixels fall within the grown region, this function will return the value -1.

Arguments

Array

An N-dimensional array of data values. The region will be grown according to the data values within this array.

ROIPixels

A vector of indices into *Array* that represent the initial region that is to be grown.

Keywords

ALL_NEIGHBORS

Set this keyword to indicate that all adjacent neighbors to a given pixel should be considered during region growing (sometimes known as 8-neighbor searching when

the array is two-dimensional). The default is to search only the neighbors that are exactly one unit in distance from the current pixel (sometimes known as 4-neighbor searching when the array is two-dimensional).

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

STDDEV_MULTIPLIER

Set this keyword to a scalar value that serves as the multiplier of the sample standard deviation of the original region pixel values. The expanded region includes neighboring pixels that fall within the range of the mean of the region’s pixel values plus or minus the given multiplier times the sample standard deviation as follows:

$$\text{Mean} \pm \text{StdDevMultiplier} * \text{StdDev}$$

This keyword is mutually exclusive of THRESHOLD. If both keywords are specified, a warning message will be issued and the THRESHOLD value will be used.

THRESHOLD

Set this keyword to a two-element vector, $[min,max]$, of the inclusive range within which the pixel values of the grown region must fall. The default is the range of pixel values within the initial region. This keyword is mutually exclusive of STDDEV_MULTIPLIER. If both keywords are specified, a warning message will be issued and the THRESHOLD value will be used.

Note

If neither keyword is specified, THRESHOLD is used by default. The range of threshold values is based upon the pixel values within the original region and therefore does not have to be provided.

Examples

The following example demonstrates how you can grow a pre-defined region within an image of human red blood cells.

```
; Load an image.
fname = FILEPATH('rbcells.jpg', SUBDIR=['examples','data'])
```

```

READ_JPEG, fname, img
imgDims = SIZE(img, /DIMENSIONS)

; Define original region pixels.
x = FINDGEN(16*16) MOD 16 + 276.
y = LINDGEN(16*16) / 16 + 254.
roiPixels = x + y * imgDims[0]

; Grow the region.
newROIPixels = REGION_GROW(img, roiPixels)

; Load a grayscale color table.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Set the topmost color table entry to red.
topClr = !D.TABLE_SIZE-1
TVLCT, 255, 0, 0, topClr

; Show the results.
tmpImg = BYTSCL(img, TOP=(topClr-1))
tmpImg[roiPixels] = topClr
WINDOW, 0, XSIZE=imgDims[0], YSIZE=imgDims[1], $
    TITLE='Original Region'
TV, tmpImg

tmpImg = BYTSCL(img, TOP=(topClr-1))
tmpImg[newROIPixels] = topClr
WINDOW, 2, XSIZE=imgDims[0], YSIZE=imgDims[1], $
    TITLE='Grown Region'
TV, tmpImg

```

Version History

Introduced: 5.5

NAN keyword added: 6.1

REGISTER_CURSOR

The REGISTER_CURSOR procedure associates the given name with the given cursor information. This name can then be used with the IDLgrWindow::SetCurrentCursor method.

Tip

You can use the [CREATE_CURSOR](#) function to translate a string array into an image array suitable for the *Image* argument. You can also return values for the MASK and HOTSPOT keywords using this function.

Syntax

```
REGISTER_CURSOR, Name, Image[, MASK=value] [, HOTSPOT=value]  
[, /OVERWRITE]
```

Arguments

Name

This argument sets the name to associate with this cursor. The name is case-insensitive. Once registered, the name can be used with the IDLgrWindow::SetCurrentCursor method.

Image

Set this argument to a 16 line by 16 column bitmap, contained in a 16-element short integer vector, specifying the cursor pattern. The offset from the upper-left pixel to the point that is considered the hot spot can be provided using the [HOTSPOT](#) keyword.

Keywords

MASK

This keyword can be used to simultaneously specify the mask that should be used. In the mask, bits that are set indicate bits in the IMAGE that should be seen and bits that are not are masked out.

HOTSPOT

Set this keyword to a two-element vector specifying the $[x, y]$ pixel offset of the cursor "hot spot", the point which is considered to be the mouse position, from the lower-left corner of the cursor image. The cursor image is displayed top-down (the first row is displayed at the top).

OVERWRITE

By default, if the cursor already exists, the values are not changed. By setting this keyword to true, the current cursor value is updated with the values provided by this routine call.

Version History

Introduced: 5.6

See Also

[IDLgrWindow::SetCurrentCursor](#)

REGRESS

The REGRESS function performs a multiple linear regression fit and returns an *Nterm*-element column vector of coefficients.

REGRESS fits the function:

$$y_i = \text{const} + a_0x_{0,i} + a_1x_{1,i} + \dots + a_{Nterms-1}x_{Nterms-1,i}$$

This routine is written in the IDL language. Its source code can be found in the file `regress.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = REGRESS( X, Y, [, CHISQ=variable] [, CONST=variable]
  [, CORRELATION=variable] [, /DOUBLE] [, FTEST=variable]
  [, MCORRELATION=variable] [, MEASURE_ERRORS=vector]
  [, SIGMA=variable] [, STATUS=variable] [, YFIT=variable] )
```

Return Value

REGRESS returns a 1 x *Nterm* array of coefficients. If the DOUBLE keyword is set, or if *X* or *Y* are double-precision, then the result will be double precision, otherwise the result will be single precision.

Arguments

X

An *Nterms* by *Npoints* array of independent variable data, where *Nterms* is the number of coefficients (independent variables) and *Npoints* is the number of samples.

Y

An *Npoints*-element vector of dependent variable points.

Weights

The Weights argument is obsolete, and has been replaced by the MEASURE_ERRORS keyword. Code that uses the Weights argument will continue to work as before, but new code should use the MEASURE_ERRORS keyword instead. Note that the definition of the MEASURE_ERRORS keyword is different from that of the Weights argument. Using the Weights argument, SQRT(1/Weights[i]) represents

the measurement error for each point $Y[i]$. Using the `MEASURE_ERRORS` keyword, the measurement error for each point is represented as simply `MEASURE_ERRORS[i]`. Also note that the `RELATIVE_WEIGHTS` keyword is not necessary when using the `MEASURE_ERRORS` keyword.

Yfit, Const, Sigma, Ftest, R, Rmul, Chisq, Status

The `Yfit`, `Const`, `Sigma`, `Ftest`, `R`, `Rmul`, `Chisq`, and `Status` arguments are obsolete, and have been replaced by the `YFIT`, `CONST`, `SIGMA`, `FTEST`, `CORRELATION`, `MCORRELATION`, `CHISQ`, and `STATUS` keywords, respectively. Code that uses these arguments will continue to work as before, but new code should use the keywords instead.

Keywords

CHISQ

Set this keyword equal to a named variable that will contain the value of the unreduced chi-square goodness-of-fit statistic.

CONST

Set this keyword to a named variable that will contain the constant term of the fit.

CORRELATION

Set this keyword to a named variable that will contain the vector of linear correlation coefficients.

DOUBLE

Set this keyword to force computations to be done in double-precision arithmetic.

FTEST

Set this keyword to a named variable that will contain the F-value for the goodness-of-fit test.

MCORRELATION

Set this keyword to a named variable that will contain the multiple linear correlation coefficient.

MEASURE_ERRORS

Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y .

Note

For Gaussian errors (e.g., instrumental uncertainties), `MEASURE_ERRORS` should be set to the standard deviations of each point in Y . For Poisson or statistical weighting, `MEASURE_ERRORS` should be set to `SQRT(Y)`.

RELATIVE_WEIGHT

This keyword is obsolete. Code using the `Weights` argument and `RELATIVE_WEIGHT` keyword will continue to work as before, but new code should use the `MEASURE_ERRORS` keyword, for which case the `RELATIVE_WEIGHT` keyword is not necessary. Using the `Weights` argument, it was necessary to specify the `RELATIVE_WEIGHT` keyword if no weighting was desired. This is not the case with the `MEASURE_ERRORS` keyword—when `MEASURE_ERRORS` is omitted, `REGRESS` assumes you want no weighting.

SIGMA

Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters.

Note

If `MEASURE_ERRORS` is omitted, then you are assuming that the regression model is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in `SIGMA` are multiplied by $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number of coefficients. See section 15.2 of *Numerical Recipes in C* (Second Edition) for details.

STATUS

Set this keyword to a named variable that will contain the status of the operation. Possible status values are:

- 0 = successful completion
- 1 = singular array (which indicates that the inversion is invalid)
- 2 = warning that a small pivot element was used and that significant accuracy was probably lost.

Note

If STATUS is not specified, any error messages will be output to the screen.

YFIT

Set this keyword to a named variable that will contain the vector of calculated *Y* values.

Examples

```

; Create two vectors of independent variable data:
X1 = [1.0, 2.0, 4.0, 8.0, 16.0, 32.0]
X2 = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
; Combine into a 2x6 array
X = [TRANSPPOSE(X1), TRANSPPOSE(X2)]

; Create a vector of dependent variable data:
Y = 5 + 3*X1 - 4*X2

; Assume Gaussian measurement errors for each point:
measure_errors = REPLICATE(0.5, N_ELEMENTS(Y))

; Compute the fit, and print the results:
result = REGRESS(X, Y, SIGMA=sigma, CONST=const, $
    MEASURE_ERRORS=measure_errors)
PRINT, 'Constant: ', const
PRINT, 'Coefficients: ', result[*]
PRINT, 'Standard errors: ', sigma

```

IDL prints:

```

Constant:      4.99999
Coefficients:   3.00000   -3.99999
Standard errors: 0.0444831   0.282038

```

Version History

Introduced: Original

See Also

[CURVEFIT](#), [GAUSSFIT](#), [LINFIT](#), [LMFIT](#), [POLY_FIT](#), [SFIT](#), [SVDFIT](#)

REPEAT...UNTIL

The REPEAT...UNTIL statement repeats its subject statement(s) until an expression evaluates to true. The condition is checked after the subject statement is executed. Therefore, the subject statement is always executed at least once, even if the expression evaluates to true the first time.

Note

For information on using REPEAT_UNTIL and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

REPEAT *statement* UNTIL *expression*

or

REPEAT BEGIN

statements

ENDREP UNTIL *expression*

Examples

This example shows that because the subject of a REPEAT statement is evaluated before the expression, it is always executed at least once:

```
i = 1  
  
REPEAT BEGIN  
  
    PRINT, i  
  
ENDREP UNTIL (i EQ 1)
```

Version History

Introduced: Original

REPLICATE

The REPLICATE function returns an array with the given dimensions, filled with the scalar value specified as the first parameter.

Syntax

$$Result = \text{REPLICATE}(Value, D_1 [, \dots, D_8])$$

Return Value

Returns the array of the given dimensions.

Arguments

Value

The scalar value with which to fill the resulting array. The type of the result is the same as that of *Value*. *Value* can be any single element expression such as a scalar or 1 element array. This includes structures.

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Create D, a 5-element by 5-element array with every element set to the string “IDL” by entering:

```
D = REPLICATE('IDL', 5, 5)
```

REPLICATE can also be used to create arrays of structures. For example, the following command creates a structure named “emp” that contains a string name field and a long integer employee ID field:

```
employee = {emp, NAME:' ', ID:0L}
```

To create a 10-element array of this structure, enter:

```
emps = REPLICATE(employee, 10)
```

Version History

Introduced: Original

See Also

[MAKE_ARRAY](#)

REPLICATE_INPLACE

The REPLICATE_INPLACE procedure updates an existing array by replacing all or selected parts of it with a specified value. REPLICATE_INPLACE can be faster and use less memory than the IDL function REPLICATE or the IDL array notation for large arrays that already exist.

Note

REPLICATE_INPLACE is much faster when operating on entire arrays and rows, than when used on columns or higher dimensions.

Syntax

```
REPLICATE_INPLACE, X, Value [, D1, Loc1 [, D2, Range]]
```

Arguments

X

The array to be updated. *X* can be of any numeric type. REPLICATE_INPLACE does not change the size and type of *X*.

Value

The value which will fill all or part of *X*. *Value* may be any scalar or one-element array that IDL can convert to the type of *X*. REPLICATE_INPLACE does not change *Value*.

D1

An optional parameter indicating which dimension of *X* is to be updated.

Loc1

An array with the same number of elements as the number of dimensions of *X*. The *Loc1* and *D1* arguments together determine which one-dimensional subvector (or subvectors, if *D1* and *Range* are provided) of *X* is to be updated.

D2

An optional parameter, indicating in which dimension of *X* a group of one-dimensional subvectors are to be updated. *D2* should be different from *D1*.

Range

An array of indices of dimension $D2$ of X , indicating where to put one-dimensional updates of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

```
; Create a multidimensional zero array:
A = FLTARR( 40, 90, 10)

; Populate it with the value 4.5. (i.e., A[*]= 4.5 ):
REPLICATE_INPLACE, A, 4.5
;Update a single subvector.(i.e., A[* ,4,0]= 20. ):
REPLICATE_INPLACE, A, 20, 1, [0,4,0]

; Update a group of subvectors.(i.e., A[ 0, [0, 5,89], * ] = -8 ):
REPLICATE_INPLACE, A, -8, 3, [0,0,0], 2, [0,5,89]

; Update a 2-dimensional slice of A (i.e., A[9,* , *] = 0.):
REPLICATE_INPLACE, A, 0., 3, [9,0,0] , 2, LINDGEN(90)
```

Version History

Introduced: 5.1

See Also

[REPLICATE](#), [BLAS_AXPY](#)

RESOLVE_ALL

The `RESOLVE_ALL` procedure iteratively resolves (by compiling) any uncompiled procedures or functions that are called in any already-compiled procedure or function. The process ends when there are no unresolved routines left to compile. If an unresolved procedure or function is not in the IDL search path, `RESOLVE_ALL` exits with an error, and no additional routines are compiled.

`RESOLVE_ALL` is of special interest when using `SAVE` to construct an IDL `SAVE` file containing the compiled code for a package of routines. If you are constructing such a `SAVE` file and it contains calls to built-in IDL system functions that are not present under all operating systems (e.g., `IOCTL`), you must make sure to use `FORWARD_FUNCTION` to tell IDL that these names are functions. Otherwise, IDL may interpret them as arrays and generate unintended results.

Warning

`RESOLVE_ALL` does not resolve procedures or functions that are called via quoted strings such as `CALL_PROCEDURE`, `CALL_FUNCTION`, or `EXECUTE`, or in keywords that can contain procedure names such as `TICKFORMAT` or `EVENT_PRO`. You must manually compile these routines if they are not called normally elsewhere.

Similarly, `RESOLVE_ALL` does not resolve widget event handler procedures based on a call to the widget routine that uses the event handler. In general, it is best to include the event handling routine in the same program file as the widget creation routine—building widget programs in this way ensures that `RESOLVE_ALL` will “catch” the event handler for a widget application.

This routine is written in the IDL language. Its source code can be found in the file `resolve_all.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
RESOLVE_ALL [, CLASS=string] [, /CONTINUE_ON_ERROR] [, /QUIET]
             [, RESOLVE_EITHER=string] [, RESOLVE_FUNCTION=string]
             [, RESOLVE_PROCEDURE=string] [, SKIP_ROUTINES=string]
             [, UNRESOLVED=variable]
```

Note

To properly resolve routines, make sure you run `RESOLVE_ALL` in an IDL session in which no compilation errors have occurred. This is important because if IDL has previously attempted (and failed) to compile a routine, that routine will be added to

IDL's list of user-defined routines as an uncompiled routine. A subsequent call to `RESOLVE_ALL` will attempt to compile the uncompiled routine, potentially causing another error.

You can ensure that an IDL session is "clean" either by exiting and restarting IDL or by issuing the `.RESET_SESSION` executive command.

Arguments

None.

Keywords

CLASS

Set this keyword to a string or string array containing object class names.

`RESOLVE_ALL`'s rules for finding uncompiled functions and procedures are not able to find object definitions or methods, because those things are not known to IDL until the object classes are actually instantiated and the methods called. However, if the `CLASS` keyword is set, `RESOLVE_ALL` will ensure that the `*__define.pro` files for the specified classes and their superclasses are compiled and executed. `RESOLVE_ALL` then locates all methods for the specified classes and their superclasses, and makes sure they are also compiled.

CONTINUE_ON_ERROR

Set this keyword to allow continuation upon error.

QUIET

Set this keyword to suppress informational messages.

RESOLVE_EITHER

Set this keyword to a string or string array of routine names to resolve.

Use this keyword instead of `RESOLVE_FUNCTION` or `RESOLVE_PROCEDURE` if you do not know the type of the routine being resolved. If the routines are already compiled, they are not recompiled.

RESOLVE_FUNCTION

Set this keyword to a string or string array of function names to resolve. If the functions are already compiled, they are not recompiled.

RESOLVE_PROCEDURE

Set this keyword to a string or string array of procedure names to resolve. If the procedures are already compiled, they are not recompiled.

SKIP_ROUTINES

Set this keyword to a string array containing the names of routines *not* to resolve. Using this keyword is useful when a library file containing the designated routines will be later included.

UNRESOLVED

Set this keyword to a variable that will contain the names of the unresolved procedures and functions if `CONTINUE_ON_ERROR` is set. Routines in the `SKIP_ROUTINES` list are also included in the result.

Version History

Introduced: 4.0

CLASS keyword added: 6.0

See Also

[.COMPILE](#), [RESOLVE_ROUTINE](#), [ROUTINE_INFO](#)

RESOLVE_ROUTINE

The RESOLVE_ROUTINE procedure compiles user-written or library procedures or functions, given their names. Routines are compiled even if they are already defined. RESOLVE_ROUTINE looks for the given filename in IDL's search path. If the file is not found in the path, then the routine exits with an error.

Syntax

```
RESOLVE_ROUTINE, Name [, /COMPILE_FULL_FILE ]  
                [, /EITHER | , /IS_FUNCTION] [, /NO_RECOMPILE]
```

Arguments

Name

A scalar string or string array containing the name or names of the procedures to compile. If *Name* contains functions rather than procedures, set the IS_FUNCTION keyword. The *Name* argument cannot contain the path to the .pro file—it must contain only a .pro filename. If you want to specify a path to the .pro file, use the .COMPILE executive command.

Keywords

COMPILE_FULL_FILE

When compiling a file to find a specified routine, IDL normally stops compiling when the desired routine (*Name*) is found. If set, COMPILE_FULL_FILE compiles the entire file.

EITHER

If set, indicates that the caller does not know whether the supplied routine names are functions or procedures, and will accept either. This keyword overrides the IS_FUNCTION keyword.

IS_FUNCTION

Set this keyword to compile functions rather than procedures.

NO_RECOMPILE

Normally, RESOLVE_ROUTINE compiles all specified routines even if they have already been compiled. Setting NO_RECOMPILE indicates that such routines are not recompiled.

Version History

Introduced: 4.0

See Also

[.COMPILE](#), [RESOLVE_ALL](#), [ROUTINE_INFO](#)

RESTORE

The RESTORE procedure restores the IDL variables and routines saved in a file by the SAVE procedure.

RESTORE provides a very simple interface that restores the entire contents of the specified SAVE file in a single atomic operation. If you wish to query the contents of a SAVE file without first restoring the data, or if you wish to restore only selected items, use the interface provided by the [IDL_Savefile](#) object class instead.

Warning

While files containing IDL data variables can be restored by any version of IDL that supports the data types of the saved variables (in particular, by any version of IDL later than the version that created the SAVE file), files containing IDL routines and system variables can only be restored by versions of IDL that share the same internal code representation. Since the internal code representation changes regularly, you should always archive the IDL language source files (`.pro` files) for routines you are placing in IDL `.sav` files so you can recompile the code when a new version of IDL is released.

Syntax

```
RESTORE [, Filename] | [, FILENAME=name] [, DESCRIPTION=variable]
      [, /RELAXED_STRUCTURE_ASSIGNMENT]
      [, RESTORED_OBJECTS=variable] [, /VERBOSE]
```

Arguments

Filename

A scalar string that contains the name of the SAVE file to be restored. If *Filename* is not an absolute path name, it is considered to be relative to the current working directory. If neither the *Filename* argument or the FILENAME keyword are present, IDL attempts to restore a file named `idlsave.dat` in the current working directory.

Keywords

DESCRIPTION

Set this keyword equal to a named variable that will contain the user-supplied *description string* for the specified save file. If no description was provided when the

save file was created (via the DESCRIPTION keyword to the [SAVE](#) routine), the variable specified by this keyword will contain a null string.

FILENAME

This keyword serves exactly the same purpose as the *Filename* argument—only one of them needs to be provided. If both the *Filename* argument and the FILENAME keyword are provided, the file specified by the *Filename* argument is restored.

RELAXED_STRUCTURE_ASSIGNMENT

Normally, RESTORE is unable to restore a structure variable if the definition of its type has changed since the SAVE file was written. A common case where this occurs is when objects are saved and the class structure of the objects change before they are restored in another IDL session. In such cases, RESTORE issues an error, skips the structure, and continues restoring the remainder of the SAVE file.

Setting the RELAXED_STRUCTURE_ASSIGNMENT keyword causes RESTORE to restore such incompatible values using “relaxed structure assignment,” in which all possible data are restored using a field-by-field copy. (See the description of the [STRUCT_ASSIGN](#) procedure for additional details.)

RESTORED_OBJECTS

Set this keyword equal to a named variable that will contain an array of object references for any objects restored. The resulting list of objects is useful for programmatically calling the objects’ restore methods. If no objects are restored, the variable will contain a null object reference.

VERBOSE

Set this keyword to have IDL print an informative message for each restored object.

Examples

Suppose that you have saved all the variables from a previous IDL session with the command:

```
SAVE, /VARIABLES, FILENAME = 'session1.sav'
```

If the file `session1.sav` is located in your current working directory, the variable associated with the file can be restored by entering:

```
RESTORE, 'session1.sav'
```

Note

To restore a file that is not in your current working directory, you must specify the file path.

Version History

Introduced: Original

DESCRIPTION keyword: 6.1

See Also

[JOURNAL](#), [SAVE](#), [STRUCT_ASSIGN](#), [IDL_Savefile](#)

RETALL

The RETALL command returns control to the main program level. The effect is the same as entering the RETURN command at the interactive command prompt until the main level is reached.

Syntax

RETALL

Arguments

None

Version History

Introduced: Original

See Also

[RETURN](#)

RETURN

The RETURN command causes the program context to revert to the next-higher program level. RETURN can be called at the interactive command prompt (see “.RETURN” on page 75), inside a procedure definition, or inside a function definition.

Calling RETURN from the main program level has no effect other than to print an informational message in the command log.

Calling RETURN inside a procedure definition returns control to the calling routine, or to the main level. Since the END statement in a procedure definition also returns control to the calling routine, it is only necessary to use RETURN in a procedure definition if you wish control to revert to the calling routine before the procedure reaches its END statement.

In a function definition, RETURN serves to define the value passed out of the function. Only a single value can be returned from a function.

Note

The value can be an array or structure containing multiple data items.

Syntax

```
RETURN [, Return_value]
```

Arguments

Return_value

In a function definition, the *Return_value* is the value passed out of the function when it completes its processing.

Return values are not allowed in procedure definitions, or when calling RETURN at the interactive command prompt.

Examples

You can use RETURN within a procedure definition to exit the procedure at some point other than the end. For example, note the following procedure:

```
PRO RET_EXAMPLE, value
  IF (value NE 0) THEN BEGIN
```

```
        PRINT, value, ' is nonzero'  
        RETURN  
    END  
    PRINT, 'Input argument was zero.'  
END
```

If the input argument is non-zero, the routine prints the value and exits back to the calling procedure or main level. If the input argument is zero, control proceeds until the END statement is reached.

When defining functions, use RETURN to specify the value returned from the function. For example, the following function:

```
FUNCTION RET_EXAMPLE2, value  
    RETURN, value * 2  
END
```

multiplies the input value by two and returns the result. If this function is defined at the main level, calling it from the IDL command prompt produces the following:

```
PRINT, RET_EXAMPLE2(4)
```

IDL prints:

```
8
```

Version History

Introduced: Original

See Also

[RETALL](#)

REVERSE

The REVERSE function reverses the order of one dimension of an array.

This routine is written in the IDL language. Its source code can be found in the file `reverse.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = REVERSE( Array [, Subscript_Index] [, /OVERWRITE] )
```

Return Value

Returns the reversed dimension of the array.

Arguments

Array

The array containing the original data.

Subscript_Index

An integer specifying the dimension index (1, 2, 3, etc.) that will be reversed. This argument must be less than or equal to the number of dimensions of *Array*. If this argument is omitted, the first subscript is reversed.

Keywords

OVERWRITE

Set this keyword to conserve memory by doing the transformation “in-place.” The result overwrites the previous contents of the array. This keyword is ignored for one- or two-dimensional arrays.

Examples

Reverse the order of an array where each element is set to the value of its subscript:

```
; Create an array:  
A = [[0,1,2],[3,4,5],[6,7,8]]
```

```
; Print the array:
PRINT, 'Original Array:'
PRINT, A

; Reverse the columns of A.
PRINT, 'Reversed Columns:'
PRINT, REVERSE(A)

; Reverse the rows of A:
PRINT, 'Reversed Rows:'
PRINT, REVERSE(A, 2)
```

IDL prints:

```
Original Array:
   0   1   2
   3   4   5
   6   7   8
Reversed Columns:
   2   1   0
   5   4   3
   8   7   6
Reversed Rows:
   6   7   8
   3   4   5
   0   1   2
```

Version History

Introduced: Original

See Also

[INVERT](#), [REFORM](#), [ROT](#), [ROTATE](#), [SHIFT](#), [TRANSPOSE](#)

RK4

The RK4 function uses the fourth-order Runge-Kutta method to advance a solution to a system of ordinary differential equations one time-step H , given values for the variables Y and their derivatives $Dydx$ known at X .

RK4 is based on the routine `rk4` described in section 16.1 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

$Result = RK4(Y, Dydx, X, H, Derivs [, /DOUBLE])$

Return Value

Returns the integrations of the ordinary differential equations.

Arguments

Y

A vector of values for Y at X

Dydx

A vector of derivatives for Y at X .

X

A scalar value for the initial condition.

H

A scalar value giving interval length or step size.

Derivs

A scalar string specifying the name of a user-supplied IDL function that calculates the values of the derivatives $Dydx$ at X . This function must accept two arguments: A scalar floating value X , and one n -element vector Y . It must return an n -element vector result.

For example, suppose the values of the derivatives are defined by the following relations:

$$dy_0 / dx = -0.5y_0, \quad dy_1 / dx = 4.0 - 0.3y_1 - 0.1y_0$$

We can write a function `DIFFERENTIAL` to express these relationships in the IDL language:

```
FUNCTION differential, X, Y
    RETURN, [-0.5 * Y[0], 4.0 - 0.3 * Y[1] - 0.1 * Y[0]]
END
```

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To integrate the example system of differential equations for one time step, `H`:

```
; Define the step size:
H = 0.5

; Define an initial X value:
X = 0.0

; Define initial Y values:
Y = [4.0, 6.0]

; Calculate the initial derivative values:
dydx = DIFFERENTIAL(X,Y)

; Integrate over the interval (0, 0.5):
result = RK4(Y, dydx, X, H, 'differential')

; Print the result:
PRINT, result
```

IDL prints:

```
3.11523 6.85767
```

This is the exact solution vector to five-decimal precision.

Version History

Introduced: 4.0

See Also

[BROYDEN, NEWTON](#)

ROBERTS

The ROBERTS function returns an approximation to the Roberts edge enhancement operator for images:

$$G_{jk} = |F_{jk} - F_{j+1, k+1}| + |F_{j, k+1} - F_{j+1, k}|$$

where (j, k) are the coordinates of each pixel F_{jk} in the *Image*. This is equivalent to a convolution using the masks,

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

where the underline indicates the current pixel F_{jk} . The last column and row are set to zero.

Syntax

Result = ROBERTS(*Image*)

Return Value

ROBERTS returns a two-dimensional array of the same size as *Image*. If *Image* is of type byte or integer, then the result is of integer type, otherwise the result is of the same type as *Image*.

Note

To avoid overflow for integer types, the computation is done using the next larger signed type and the result is transformed back to the correct type. Values larger than the maximum for that integer type are truncated. For example, for integers the function is computed using type long, and on output, values larger than 32767 are set equal to 32767.

Arguments

Image

The two-dimensional array containing the image to which edge enhancement is applied.

Keywords

None.

Examples

If the variable `myimage` contains a two-dimensional image array, a Roberts sharpened version of `myimage` can be displayed with the command:

```
TVSCL, ROBERTS(myimage)
```

Version History

Introduced: 4.0

See Also

[SOBEL](#)

ROT

The ROT function rotates an image by an arbitrary amount. At the same time, it can magnify, demagnify, and/or translate an image.

Note

If you want to rotate an array by a multiple of 90 degrees, you should use the ROTATE function for faster results.

This routine is written in the IDL language. Its source code can be found in the file `rot.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

$$\text{Result} = \text{ROT}(A, \text{Angle}, [\text{Mag}, X_0, Y_0] [, /\text{INTERP}] [, \text{CUBIC}=\text{value}\{-1 \text{ to } 0\}] [, \text{MISSING}=\text{value}] [, /\text{PIVOT}])$$

Return Value

Returns the rotated and optionally transformed image.

Arguments

A

The image array to be rotated. This array can be of any type, but must have two dimensions. The output image has the same dimensions and data type of the input image.

ANGLE

Angle of rotation in degrees *clockwise*.

MAG

An optional magnification factor. A value of 1.0 results in no change. A value greater than one performs magnification. A value less than one but greater than zero performs demagnification.

X₀

X subscript for the center of rotation. If omitted, X_0 equals the number of columns in the image divided by 2.

Y₀

Y subscript for the center of rotation. If omitted, Y_0 equals the number of rows in the image divided by 2.

Keywords**INTERP**

Set this keyword to use bilinear interpolation. The default is to use nearest neighbor sampling.

CUBIC

Set this keyword to a value between -1 and 0 to use the cubic convolution interpolation method with the specified value as the interpolation parameter. Setting this keyword equal to a value greater than zero specifies a value of -1 for the interpolation parameter. Park and Schowengerdt (see reference below) suggest that a value of -0.5 significantly improves the reconstruction properties of this algorithm.

Cubic convolution is an interpolation method that closely approximates the theoretically optimum sinc interpolation function using cubic polynomials. According to sampling theory, details of which are beyond the scope of this document, if the original signal, f , is a band-limited signal, with no frequency component larger than ω_0 , and f is sampled with spacing less than or equal to $1/(2\omega_0)$, then f can be reconstructed by convolving with a sinc function: $\text{sinc}(x) = \sin(\pi x) / (\pi x)$.

In the one-dimensional case, four neighboring points are used, while in the two-dimensional case 16 points are used.

Note

Cubic convolution interpolation is significantly slower than bilinear interpolation.

For further details see:

Rifman, S.S. and McKinnon, D.M., "Evaluation of Digital Correction Techniques for ERTS Images; Final Report", Report 20634-6003-TU-00, TRW Systems, Redondo Beach, CA, July 1974.

S. Park and R. Schowengerdt, 1983 “Image Reconstruction by Parametric Cubic Convolution”, Computer Vision, Graphics & Image Processing 23, 256.

MISSING

Set this keyword to a value to be substituted for pixels in the output image that map outside the input image.

PIVOT

Set this keyword to cause the image to pivot around the point (X_0, Y_0) so that this point maps into the same point in the output image. By default, the point (X_0, Y_0) in the input image is mapped into the center of the output image.

Examples

```
; Create a byte image:
A = BYTSCL(DIST(256))

; Display it:
TV, A

; Rotate the image 33 degrees, magnify it 15 times, and use
; bilinear interpolation to make the output look nice:
B = ROT(A, 33, 1.5, /INTERP)

; Display the rotated image:
TV, B
```

Version History

Introduced: Original

See Also

[ROTATE](#)

ROTATE

The ROTATE function returns a rotated and/or transposed copy of *Array*. ROTATE can only rotate arrays in multiples of 90 degrees. To rotate by amounts other than multiples of 90 degrees, use the ROT function. Note, however, that ROTATE is more efficient.

ROTATE can also be used to reverse the order of elements in vectors. For example, to reverse the order of elements in the vector *X*, use the expression ROTATE(*X*, 2). If *X* = [0,1,2,3] then ROTATE(*X*, 2) yields the resulting array, [3,2,1,0].

Note

Transposition, if specified, is performed before rotation.

Syntax

Result = ROTATE(*Array*, *Direction*)

Return Value

Returns the rotated and/or transposed array.

Arguments

Array

The array to be rotated. *Array* can have only one or two dimensions. The result has the same type as *Array*. The dimensions of the result are the same as those of *Array* if *Direction* is equal to 0 or 2. The dimensions are transposed if the direction is 4 or greater.

Direction

Direction specifies the operation to be performed as follows:

Direction	Transpose?	Rotation Counterclockwise	X_1	Y_1
0	No	None	X_0	Y_0
1	No	90°	$-Y_0$	X_0

Table 3-103: Rotation Directions

Direction	Transpose?	Rotation Counterclockwise	X_1	Y_1
2	No	180°	$-X_0$	$-Y_0$
3	No	270°	Y_0	$-X_0$
4	Yes	None	Y_0	X_0
5	Yes	90°	$-X_0$	Y_0
6	Yes	180°	$-Y_0$	$-X_0$
7	Yes	270°	X_0	$-Y_0$

Table 3-103: Rotation Directions (Continued)

In the table above, (X_0, Y_0) are the original subscripts, and (X_1, Y_1) are the subscripts of the resulting array. The notation $-Y_0$ indicates a reversal of the Y axis, $Y_1 = N_y - Y_0 - 1$. *Direction* is taken modulo 8, so a rotation of -1 is the same as 7, 9 is the same as 1, etc.

Note

The assertion that *Array* is rotating counterclockwise may cause some confusion. Remember that when arrays are displayed on the screen (using TV or TVSCL, for example), the image is drawn by default with the origin (0,0) at the bottom left corner of the window. (This default can be changed by changing the value of the !ORDER system variable.) When arrays are printed on the console or command log window (using the PRINT command, for example), the (0,0) element is drawn in the upper left corner of the array. This means that while an image displayed in a graphics window appears to rotate counterclockwise, an array printed in the command log appears to rotate clockwise.

Examples

Create and display a wedge image by entering:

```
F = REPLICATE(1, 256) # FINDGEN(256) & TVSCL, F
```

To display the image rotated 90 degrees counterclockwise, enter:

```
TVSCL, ROTATE(F, 1)
```

Version History

Introduced: Original

See Also

[ROT](#), [TRANSPOSE](#)

ROUND

The ROUND function rounds the argument to its closest integer.

Syntax

Result = ROUND(*X* [, /L64])

Return Value

Returns the integer closest to its argument. If the input value *X* is integer type, *Result* has the same value and type as *X*. Otherwise, *Result* is returned as a 32-bit longword integer with the same structure as *X*.

Arguments

X

The value for which the ROUND function is to be evaluated. This value can be any numeric type (integer, floating, or complex). Note that only the real part of a complex argument is rounded and returned.

Keywords

L64

If set, the result type is 64-bit integer no matter what type the input has. This is useful for situations in which a floating point number contains a value too large to be represented in a 32-bit integer.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Print the rounded values of a 2-element vector:

```
PRINT, ROUND([5.1, 5.9])
```

IDL prints:

```
5      6
```

Print the result of rounding 3000000000.1, a value that is too large to represent in a 32-bit integer:

```
PRINT, ROUND(3000000000.1D, /L64)
```

IDL prints:

```
3000000000
```

Version History

Introduced: Pre 4.0

See Also

[CEIL](#), [COMPLEXROUND](#), [FLOOR](#)

ROUTINE_INFO

The ROUTINE_INFO function provides information about currently-compiled procedures and functions.

Syntax

```
Result = ROUTINE_INFO( [Routine [, /PARAMETERS {must specify Routine}]  
[, /SOURCE ] [, /UNRESOLVED] [, /VARIABLES] | [, /SYSTEM]]  
[, /DISABLED] [, /ENABLED] [, /FUNCTIONS] )
```

Return Value

Returns a string array consisting of the names of defined procedures or functions, or of parameters or variables used by a single procedure or function.

Arguments

Routine

A scalar string containing the name of routine for which information will be returned. *Routine* can be either a procedure or a function. If *Routine* is not supplied, ROUTINE_INFO returns a list of all currently-compiled procedures.

Keywords

DISABLED

Set this keyword to get the names of currently disabled system procedures or functions (in conjunction with the FUNCTIONS keyword). Use of DISABLED implies use of the SYSTEM keyword, since user routines cannot be disabled.

ENABLED

Set this keyword to get the names of currently enabled system procedures or functions (in conjunction with the FUNCTIONS keyword). Use of ENABLED implies use of the SYSTEM keyword, since user routines cannot be disabled.

FUNCTIONS

Set this keyword to return a string array containing currently-compiled functions. By default, ROUTINE_INFO returns a list of compiled procedures. If the SYSTEM

keyword is also set, `ROUTINE_INFO` returns a list of all IDL built-in internal functions.

PARAMETERS

Set this keyword to return an anonymous structure with the following fields:

- **NUM_ARGS** — An integer containing the number of positional parameters used in *Routine*.
- **NUM_KW_ARGS** — An integer containing the number of keyword parameters used in *Routine*.
- **ARGS** — A string array containing the names of the positional parameters used in *Routine*.
- **KW_ARGS** — A string array containing the names of the keyword parameters used in *Routine*.

You must supply the *Routine* argument when using this keyword. If *Routine* is a function, you must specify the `FUNCTIONS` keyword as well. Note that specifying the `SYSTEM` keyword along with this keyword will generate an error.

If *Routine* does not take any arguments, the `ARGS` field is not included in the anonymous structure. Similarly, if *Routine* does not take any keywords, the `KW_ARGS` field is not included.

SOURCE

Set this keyword to return an array of anonymous structures with the following fields:

- **NAME** — A string containing the name of the procedure or function.
- **PATH** — A string containing the full path specification of the file that contains the definition of the procedure or function.

If *Routine* is specified, information for the specified procedure is returned. If *Routine* is not specified, information for all compiled procedures is returned. If the `FUNCTIONS` keyword is also set, `ROUTINE_INFO` returns information for the specified function or for all compiled functions.

If a routine is unresolved or its path information is unavailable, the `PATH` field will contain a null string. If a routine has been `SAVED` and then `RESTORED`, the `PATH` field will contain the path to the `SAVE` file.

Note

Specifying the `SYSTEM` keyword along with this keyword will generate an error.

SYSTEM

Set this keyword to return a string array listing all IDL built-in internal procedures. Built-in internal procedures are part of the IDL executable, and are *not* written in the IDL language. If the FUNCTIONS keyword is also set, ROUTINE_INFO returns a list of all IDL built-in internal functions.

UNRESOLVED

Set this keyword to return a string array listing procedures that are referenced in any currently-compiled procedure or function, but which are themselves not yet compiled. If the FUNCTIONS keyword is also set, ROUTINE_INFO returns a list of functions that are referenced but not yet compiled.

Note that specifying the SYSTEM keyword along with this keyword will generate an error.

VARIABLES

Set this keyword to return a string array listing variables defined in the procedure. If the FUNCTIONS keyword is also set, ROUTINE_INFO returns a string array listing variables defined in the function.

You must supply the *Routine* argument when using this keyword. Note that specifying the SYSTEM keyword along with this keyword will generate an error.

Version History

Introduced: 5.0

See Also

[RESOLVE_ALL](#), [RESOLVE_ROUTINE](#)

RS_TEST

The RS_TEST function tests the hypothesis that two sample populations X and Y have the same mean of distribution against the hypothesis that they differ. X and Y may be of different lengths. This type of test is often referred to as the “Wilcoxon Rank-Sum Test” or the “Mann-Whitney U-Test.”

The Mann-Whitney statistics for X and Y are defined as follows:

$$U_x = N_x N_y + \frac{N_x(N_x + 1)}{2} - W_x$$

$$U_y = N_x N_y + \frac{N_y(N_y + 1)}{2} - W_y$$

where N_x and N_y are the number of elements in X and Y , respectively, and W_x and W_y are the rank sums for X and Y , respectively. The test statistic Z , which closely follows a normal distribution for sample sizes exceeding 10 elements, is defined as follows:

$$Z = \frac{U_x - (N_x N_y)/2}{\sqrt{(N_x N_y (N_x + N_y + 1))/12}}$$

This routine is written in the IDL language. Its source code can be found in the file `rs_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = RS_TEST(X , Y [, **UX**=variable] [, **UY**=variable])

Return Value

The result is a two-element vector containing the nearly-normal test statistic Z and the one-tailed probability of obtaining a value of Z or greater.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An m -element integer, single-, or double-precision floating-point vector.

Keywords

UX

Set this keyword to a named variable that will contain the Mann-Whitney statistic for X .

UY

Set this keyword to a named variable that will contain the Mann-Whitney statistic for Y .

Examples

```

; Define two sample populations:
X = [-14, 3, 1, -16, -21, 7, -7, -13, -22, -17, -14, -8, $
      7, -18, -13, -9, -22, -25, -24, -18, -13, -13, -18, -5]
Y = [-18, -9, -16, -14, -3, -9, -16, 10, -11, -3, -13, $
      -21, -2, -11, -16, -12, -13, -6, -9, -7, -11, -9]

; Test the hypothesis that two sample populations, {xi, yi}, have
; the same mean of distribution against the hypothesis in that they
; differ at the 0.05 significance level:
PRINT, RS_TEST(X, Y, UX = ux, UY = uy)

; Print the Mann-Whitney statistics:
PRINT, 'Mann-Whitney Statistics: Ux = ', ux, ', Uy = ', uy

```

IDL prints:

```

[1.45134, 0.0733429]
Mann-Whitney Statistics: Ux = 330.000, Uy = 198.000

```

The computed probability (0.0733429) is greater than the 0.05 significance level and therefore we do not reject the hypothesis that X and Y have the same mean of distribution.

Version History

Introduced: 4.0

See Also

[FV_TEST](#), [KW_TEST](#), [S_TEST](#), [TM_TEST](#)

S_TEST

The `S_TEST` function tests the hypothesis that two sample populations X and Y have the same mean of distribution against the hypothesis that they differ.

This routine is written in the IDL language. Its source code can be found in the file `s_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = S_TEST( X, Y [, ZDIFF=variable] )
```

Return Value

The result is a two-element vector containing the maximum number of signed differences between corresponding pairs of x_i and y_i and its one-tailed significance. This type of test is often referred to as the “Sign Test.”

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An n -element integer, single-, or double-precision floating-point vector.

Keywords

ZDIFF

Set this keyword to a named variable that will contain the number of differences between corresponding pairs of x_i and y_i resulting in zero. Paired data resulting in a difference of zero are excluded from the ranking and the sample size is correspondingly reduced.

Examples

```
; Define two n-element sample populations:  
X = [47, 56, 54, 49, 36, 48, 51, 38, 61, 49, 56, 52]  
Y = [71, 63, 45, 64, 50, 55, 42, 46, 53, 57, 75, 60]
```

```
    ; Test the hypothesis that the two sample populations have the same  
    ; mean of distribution against the hypothesis that they differ at  
    ; the 0.05 significance level:  
    PRINT, S_TEST(X, Y, ZDIFF = zdiff)
```

IDL prints:

```
[9.00000, 0.0729981]
```

The computed probability (0.0729981) is greater than the 0.05 significance level and therefore we do not reject the hypothesis that X and Y have the same mean of distribution.

Version History

Introduced: 4.0

See Also

[FV_TEST](#), [KW_TEST](#), [MD_TEST](#), [RS_TEST](#), [TM_TEST](#)

SAVE

The SAVE procedure saves variables, system variables, and IDL routines in a SAVE file using the XDR (eXternal Data Representation) format for later recovery by RESTORE. Note that variables and routines cannot be saved in the same file. Note also that SAVE files containing routines may not be compatible between different versions of IDL, but that files containing data are always backwards-compatible.

Note

While IDL routines or data can be saved in a file with any extension, it is common to use the extension `.sav` for SAVE files. Using the `.sav` extension has two benefits: it makes it clear to another IDL user that the file contains IDL routines or data that can be restored, and it allows IDL to locate routines with the same base name as the file in SAVE files located in IDL's path.

Syntax

```
SAVE [, Var1, ..., Varn] [, /ALL] [, /COMM, /VARIABLES] [, /COMPRESS]  
    [, /DESCRIPTION=string] [, /FILENAME=string] [, /ROUTINES]  
    [, /SYSTEM_VARIABLES] [, /VERBOSE]
```

Arguments

*Var*_{*n*}

The names of one or more variables that are to be saved. If the ROUTINES keyword is present, the *Var* arguments should be strings containing the names of the routines to be saved. If no *Var* arguments are provided, the items specified by the supplied keywords are saved.

Keywords

ALL

Set this keyword to save all common blocks, system variables, and local variables from the current IDL session.

Note

Routines and variables cannot be saved in the same file. Setting the ALL keyword does not save routines.

COMM

Set this keyword to save all main level common block definitions. Note that setting this keyword does not cause the contents of the common block to be saved unless the `VARIABLES` keyword is also set.

COMPRESS

If `COMPRESS` is set, IDL writes all data to the `SAVE` file using the `ZLIB` compression library to reduce its size. IDL's `SAVE` file compression support is based on the freely available `ZLIB` library by Mark Adler and Jean-loup Gailly.

Compressed `SAVE` files can be restored by the `RESTORE` procedure in exactly the same manner as any other `SAVE` file. The only visible differences are that the files will be smaller, and writing and reading them will be somewhat slower under typical conditions.

DESCRIPTION

Set this keyword equal to a string that will be saved in the `SAVE` file:

```
SAVE, data, $
    DESCRIPTION="Solar Obs: 2452884.9, 40 deg 0' N 105 deg 16' W"
```

This string can be later recovered using the [RESTORE](#) procedure or the `IDL_Savefile` object class. The ability to associate a descriptive string to a `SAVE` file can be used by IDL programs to identify data, and can also be used as an aid in managing large collections of data.

FILENAME

A string containing the name of the file into which the IDL objects should be saved. While there are no arbitrary limitations on the name of a file created with the `SAVE` procedure, RSI recommends that you use the `.sav` extension for your files. If this keyword is not specified, the file `idlsave.dat` is used.

ROUTINES

Set this keyword to save user defined procedures and functions in a machine independent, binary form. If parameters are present, they must be strings containing the names of the procedures and/or functions to be saved. If no parameters are present, all compiled routines are saved. Routines and variables cannot be saved in the same file.

Warning

`.sav` files containing routines are not guaranteed to be compatible between successive versions of IDL. You will not be able to `RESTORE` `.sav` files containing routines if they are made with incompatible versions of IDL. In this case, you should recompile your original code with the newer version of IDL. A `.sav` file containing data will always be restorable.

SYSTEM_VARIABLES

Set this keyword to save the current state of all system variables.

Warning

Saving system variables is not recommended, as the structure may change between versions of IDL.

VARIABLES

Set this keyword to save all variables in the current program unit. This option is the default.

VERBOSE

Set this keyword to print an informative message for each saved object.

Examples

Save the status of all currently-defined variables in the file `variables1.sav` by entering:

```
SAVE, /VARIABLES, FILENAME = 'variables1.sav'
```

The variables can be restored with the `RESTORE` procedure.

Save the user procedures `MYPROC` and `MYFUN`:

```
SAVE, /ROUTINES, 'MYPROC', 'MYFUN', FILENAME = 'myroutines.sav'
```

Version History

Introduced: Original

DESCRIPTION keyword: 6.1

See Also

[JOURNAL](#), [RESOLVE_ALL](#), [RESTORE](#), [IDL_Savefile](#)

SAVGOL

The SAVGOL function returns the coefficients of a Savitzky-Golay smoothing filter, which can then be applied using the CONVOL function. The Savitzky-Golay smoothing filter, also known as least squares or DISPO (digital smoothing polynomial), can be used to smooth a noisy signal.

The filter is defined as a weighted moving average with weighting given as a polynomial of a certain degree. The returned coefficients, when applied to a signal, perform a polynomial least-squares fit within the filter window. This polynomial is designed to preserve higher moments within the data and reduce the bias introduced by the filter. The filter can use any number of points for this weighted average.

This filter works especially well when the typical peaks of the signal are narrow. The heights and widths of the curves are generally preserved.

Tip

You can use this function in conjunction with the CONVOL function for smoothing and optionally for numeric differentiation.

This routine is written in the IDL language. Its source code can be found in the file `savgol.pro` in the `lib` subdirectory of the IDL distribution.

SAVGOL is based on the Savitzky-Golay Smoothing Filters described in section 14.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

$$Result = SAVGOL(Nleft, Nright, Order, Degree [, /DOUBLE])$$

Return Value

This function returns an array of floating-point numbers that are the coefficients of the smoothing filter.

Arguments

Nleft

An integer specifying the number of data points to the left of each point to include in the filter.

Nright

An integer specifying the number of data points to the right of each point to include in the filter.

Note

Larger values of *Nleft* and *Nright* produce a smoother result at the expense of flattening sharp peaks.

Order

An integer specifying the order of the derivative desired. For smoothing, use order 0. To find the smoothed first derivative of the signal, use order 1, for the second derivative, use order 2, etc.

Note

Order must be less than or equal to the value specified for *Degree*.

Note

For $ORDER > 0$, SAVGOL returns unnormalized coefficients. To use the SAVGOL coefficients with CONVOL, you should multiply the returned coefficients by $FACTORIAL(ORDER)/(dx^{ORDER})$, where dx is the sampling interval between data points. See the example below.

Degree

An integer specifying the degree of smoothing polynomial. Typical values are 2 to 4. Lower values for *Degree* will produce smoother results but may introduce bias, higher values for *Degree* will reduce the filter bias, but may “over fit” the data and give a noisier result.

Note

Degree must be less than the filter width ($Nleft + Nright + 1$).

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Tip

The DOUBLE keyword is recommended for *Degree* greater than 9.

Examples

The following example creates a noisy 400-point vector with 4 Gaussian peaks of decreasing width. It then plots the original vector, the vector smoothed with a 33-point Boxcar smoother (the SMOOTH function), and the vector smoothed with 33-point wide Savitzky-Golay filter of degree 4. The bottom plot shows the second derivative of the signal (without noise) and the second derivative of the noisy data using the Savitzky-Golay filter of degree 4:

```

PRO Savgol_example

n = 401 ; number of points
np = 4  ; number of peaks

; Form the baseline:
y = REPLICATE(0.5, n)

; Sampling interval:
dt = 0.1

; Index the array:
x = dt*FINDGEN(n)

; Add each Gaussian peak:
FOR i=0, np-1 DO BEGIN
  c = dt*(i + 0.5) * FLOAT(n)/np; Center of peak
  peak = 3 * (x-c) / (dt*(75. / 1.5 ^ i))
  ; Add Gaussian. Cutoff of -50 avoids underflow errors for
  ; tiny exponentials:
  y = y + EXP((-peak^2)>(-50))
ENDFOR

; Add noise:
y1 = y + 0.10 * RANDOMN(-121147, n)

; Display first plot
iPlot, x, y1, NAME='Signal+Noise', VIEW_GRID=[1,2]

; Get an object reference to the iTool and insert legend.
void = ITGETCURRENT(TOOL=oTool)
void = oTool->DoAction('Operations/Insert/Legend')

iPlot, x, SMOOTH(y1, 33, /EDGE_TRUNCATE), /OVERPLOT, $

```

```

        COLOR=[255, 0, 0], $
        NAME='Smooth (width 33)'
void = oTool->DoAction('Operations/Insert/LegendItem')

; Savitzky-Golay with 33, 4th degree polynomial:
savgolFilter = SAVGOL(16, 16, 0, 4)
iPlot, x, CONVOL(y1, savgolFilter, /EDGE_TRUNCATE), /OVERPLOT, $
        COLOR=[0, 0, 255], THICK=2, $
        NAME='Savitzky-Golay (width 33, 4th degree)'
void = oTool->DoAction('Operations/Insert/LegendItem')

iPlot, x, DERIV(x, DERIV(x, y)), YRANGE=[-4, 2], /VIEW_NEXT, $
        NAME='Second derivative'

void = oTool->DoAction('Operations/Insert/Legend')

order = 2
; Don't forget to normalize the coefficients.
savgolFilter = SAVGOL(16, 16, order, 4)*(FACTORIAL(order)/ $
        (dt^order))
iPlot, x, CONVOL(y1, savgolFilter, /EDGE_TRUNCATE), /OVERPLOT, $
        COLOR=[0, 0, 255], THICK=2, $
        NAME='Savitzky-Golay(width 33, 4th degree, order 2)'
void = oTool->DoAction('Operations/Insert/LegendItem')

END

```

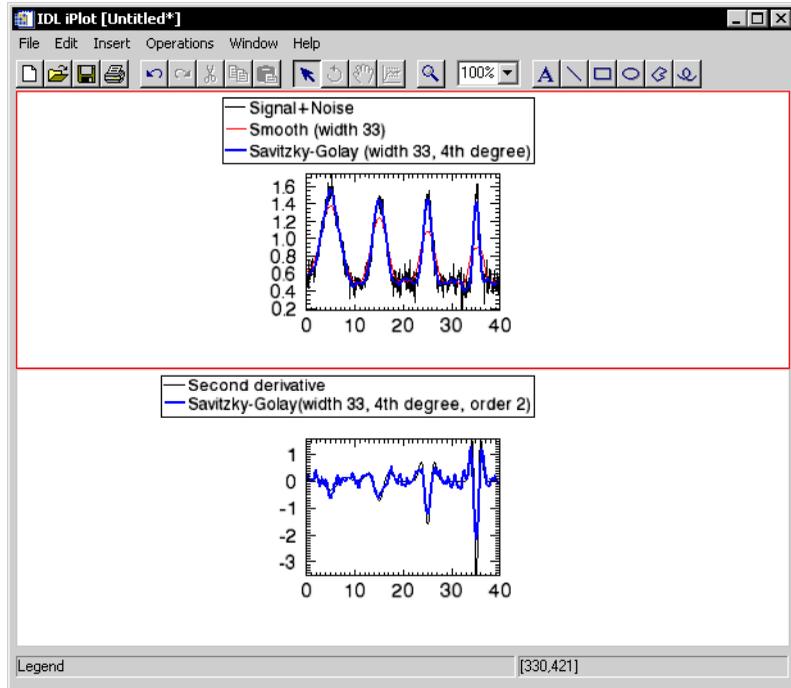


Figure 3-53: SAVGOL Example

Version History

Introduced: 5.4

See Also

[CONVOL](#), [DIGITAL_FILTER](#), [SMOOTH](#)

SCALE3

The SCALE3 procedure sets up transformation and scaling parameters for basic 3-D viewing. This procedure is similar to SURFR and SCALE3D, except that the data ranges must be specified and the scaling does not vary with rotation. Results are stored in the system variables !P.T, !X.S, !Y.S, and !Z.S.

This routine is written in the IDL language. Its source code can be found in the file `scale3.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SCALE3 [, XRANGE=vector] [, YRANGE=vector] [, ZRANGE=vector]  
      [, AX=degrees] [, AZ=degrees]
```

Arguments

None.

Keywords

XRANGE

A two-element vector containing the minimum and maximum X values. If omitted, the X-axis scaling remains unchanged.

YRANGE

A two-element vector containing the minimum and maximum Y values. If omitted, the Y-axis scaling remains unchanged.

ZRANGE

A two-element vector containing the minimum and maximum Z values. If omitted, the Z-axis scaling remains unchanged.

AX

Angle of rotation about the X axis. The default is 30 degrees.

AZ

Angle of rotation about the Z axis. The default is 30 degrees.

Examples

Set up a 3-D transformation where the data range is 0 to 20 for each of the 3 axes and the viewing area is rotated 20 degrees about the X axis and 55 degrees about the Z axis:

```
SCALE3, XRANGE=[0, 20], YRANGE=[0, 20], ZRANGE=[0, 20], AX=20,  
AZ=55
```

Version History

Introduced: Pre 4.0

See Also

[SCALE3D](#), [SURFR](#), [T3D](#)

SCALE3D

The SCALE3D procedure scales the 3-D unit cube (a cube with the length of each side equal to 1) into the viewing area. Eight data points are created at the vertices of the 3-D unit cube. The vertices are then transformed by the value of the system variable !P.T. The system is translated to bring the minimum (x,y,z) point to the origin, and then scaled to make each coordinate's maximum value equal to 1. The !P.T system variable is modified as a result.

This routine is written in the IDL language. Its source code can be found in the file `scale3d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

SCALE3D

Arguments

None.

Keywords

None.

Version History

Introduced: Original

See Also

[SCALE3](#), [SURFR](#), [T3D](#)

SCOPE_LEVEL

The `SCOPE_LEVEL` function returns the scope level of the currently running procedure or function.

Note

For more information, see [“Interpreter Frames, Level, and Variable Scope”](#) on page 1846.

Syntax

Result = `SCOPE_LEVEL()`

Return Value

Returns an integer specifying the level of the currently running procedure or function.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

See Also

[HELP](#), [SCOPE_VARFETCH](#), [SCOPE_VARNAME](#), [ROUTINE_INFO](#)

SCOPE_VARFETCH

The SCOPE_VARFETCH function returns variables outside the local scope of the currently running procedure or function. You can use it to examine or alter variables that are not otherwise visible, or to create new variables in other scopes.

In normal IDL operation, routines are not allowed to see variables outside their own scope. This restriction is a fundamental and important property of structured modular programming. Code that accesses non-local scope to alter variables within other active routines can quickly become extremely difficult to understand and maintain. For this reason, RSI recommends against the use of the IDL scope routines.

Despite this warning, there is an important class of application that requires the functionality provided by these routines. Programs, usually with graphical user interfaces, that import and export data from the callers' scope need to be able to access the user's data variables directly, and without requiring them to explicitly pass those variables to the application as parameters. The SCOPE_VARFETCH function is used to access those variables, and SCOPE_VARNAME is used to obtain the correct names with which to refer to them. SCOPE_LEVEL returns the scope level of the currently executing routine. The IDL iTools and the ENVI image processing software are examples of programs that need to be able to perform these operations.

Note

To use this routine successfully, you need to understand the information in [“Altering Accessed Variables and Avoiding Unwanted Copies”](#) on page 1849.

Interpreter Frames, Level, and Variable Scope

When IDL starts, it is implicitly running within a main program named \$MAIN\$. You can call procedures or functions from \$MAIN\$, and those routines can in turn call other routines. Each such routine invocation is represented internally by IDL using an *interpreter frame*, which IDL uses to maintain access to all variables known to that invocation of that routine. These variables can be local to the routine, parameters passed into the routine by its caller, or global variables (such as common block variables or heap variables). The set of variables visible within a given interpreter frame forms the *variable scope* of the corresponding routine. As routines are called, IDL pushes interpreter frames onto a call stack, and as those routines return, their frames are removed from the call stack. In the case of a recursive routine (a routine that calls itself), each invocation of the routine is given its own distinct and unshared frame.

The frames on the interpreter stack are numbered starting at 1 and increasing by 1 for each routine. This number is the *level* of the frame and, by implication, the scope that the frame accesses. The call stack always contains the frame for \$MAIN\$, which is the routine in which IDL execution starts and the only routine in IDL from which you cannot return. \$MAIN\$ is therefore always level 1. Any routine called from \$MAIN\$ will be at level 2, routines called from there will be level 3, and so forth.

Syntax

```
Result = SCOPE_VARFETCH(VarName [, COMMON=string] [, /ENTER]
                        [, LEVEL=value] [, /REF_EXTRA] )
```

Return Value

Returns the variable specified by *VarName* relative to the scope specified by the **LEVEL** keyword. The value returned by `SCOPE_VARFETCH` can be used by the caller to directly access, or alter, that variable.

Arguments

VarName

If the **COMMON** keyword is not set: *VarName* is the name of the variable to be returned from scope specified by the **LEVEL** keyword.

If the **COMMON** keyword is set: If *VarName* is of non-string type, it is the positional index of the variable to be returned within the specified common block. If *VarName* is a string, it is the name of the variable to be returned within the common block. In this case, the name corresponds to the name used for that common block variable in the first routine that defined the common block.

Note

For more information, see “[COMMON Variable Names](#)” on page 1855.

Keywords

COMMON

Set this keyword to a string containing the name of a common block to cause `SCOPE_VARFETCH` to return a variable contained within the specified block. The variable to return is determined by the value of *VarName*, as discussed above.

The **COMMON** keyword cannot be used in conjunction with **ENTER**, **LEVEL**, or **REF_EXTRA**.

ENTER

By default, **SCOPE_VARFETCH** will return only variables that already exist in the specified scope. Attempts to access a nonexistent variable will cause IDL to issue an error and halt execution of the program. Set this keyword to alter this behavior. If **ENTER** is set and the desired variable does not exist in the specified scope, **SCOPE_VARFETCH** will create a new variable within that scope and return the new variable. This keyword can be used to export data into other scopes. Generally, the scope of the calling routine and that of **\$MAIN\$** are most likely to be useful destinations.

The **ENTER** keyword cannot be used in conjunction with either **COMMON** or **REF_EXTRA**.

LEVEL

Set this keyword to an integer that specifies the scope within which the desired variable should be found. The level can be either an absolute value or relative to the current scope, as shown in the following table:

Value	Meaning
0	Level 0 always refers to the frame for the currently executing routine. This is the default value.
< 0	Negative levels are a relative specification and refer to the frames of active routines relative to the currently executing routine. Level -1 is the direct caller of the current routine, -2 is the caller of the caller of the current routine, and so forth. If you specify a level that is deeper than the current call stack, SCOPE_VARFETCH clips the value not to go past \$MAIN\$.
> 0	Positive levels are an absolute specification and refer directly to the specified frame. Level 1 is \$MAIN\$, level 2 is the routine called from \$MAIN\$, and so forth. If you specify a level that is past the currently executing routine, SCOPE_VARFETCH clips the value to the level of the current routine.

Table 7: LEVEL Values

The **LEVEL** keyword cannot be used in conjunction with the **COMMON** keyword.

REF_EXTRA

Set this keyword to cause `SCOPE_VARFETCH` to return the variable with the keyword name given by *VarName* from the `_REF_EXTRA` keywords within the specified level. Normally, a routine is not able to access the `_REF_EXTRA` keywords passed to it, and can pass them on only to other routines it calls. However, the `SCOPE_VARFETCH REF_EXTRA` keyword provides a mechanism by which a routine can examine its own `_REF_EXTRA` keywords.

Note

See “[Keyword Inheritance](#)” in Chapter 4 of the *Building IDL Applications* manual for more information.

The `REF_EXTRA` keyword cannot be used in conjunction with either `COMMON` or `ENTER`.

Altering Accessed Variables and Avoiding Unwanted Copies

The variable returned by `SCOPE_VARFETCH` can be used directly in IDL expressions to access the value of a variable from another scope. For example, the following statements access the value of a variable named `mydata` within the scope of the main program, `$MAIN$`:

```
PRINT, SCOPE_VARFETCH('mydata', LEVEL=1)
local_data = SCOPE_VARFETCH('mydata', LEVEL=1) + 100
```

Note that `local_data` contains only a copy of `mydata`, not a reference, so you can change `local_data` without affecting `mydata`.

If you do want to alter the value of a variable such as `mydata`, you must surround the call to `SCOPE_VARFETCH` with parentheses. For example, the following statements set `mydata` to a `FINDGEN` vector, and then increment it and add a constant to it. Finally, the value of the 5th element of the array is changed to 1000:

```
(SCOPE_VARFETCH('mydata', LEVEL=1)) = FINDGEN(100)
(SCOPE_VARFETCH('mydata', LEVEL=1))++
(SCOPE_VARFETCH('mydata', LEVEL=1)) += 57
(SCOPE_VARFETCH('mydata', LEVEL=1))[4] = 1000
```

To prevent IDL from making unwanted copies of the variable returned from `SCOPE_VARFETCH`, you should operate on the result of the `SCOPE_VARFETCH` function directly and not assign it to other variables. For example, consider the following code, which prints the value of a variable `mydata` from the main program, `$MAIN$`:

```
temp = SCOPE_VARFETCH('mydata', LEVEL=1)
PRINT, temp
```

This code will produce the desired result, but at a higher than necessary cost. Assigning the variable returned by `SCOPE_VARFETCH` to the variable `temp` causes IDL to allocate new memory for `temp` and to make a copy of the `mydata` variable from `$MAIN$` into `temp`. This process is slower, and will use twice as much memory as necessary, which can be a significant problem if the original `mydata` variable is very large. Instead, you should write the code this way:

```
PRINT, SCOPE_VARFETCH('mydata', LEVEL=1)
```

In cases where your code simply needs the value of a variable from another scope, assigning the result of `SCOPE_VARFETCH` to a local variable will use more memory and time, but is otherwise equivalent to using the direct result of the function. However, if your code wants to alter the original variable, using a local variable will produce the wrong answer. Consider the assignment code from above, modified to use a local variable:

```
temp = SCOPE_VARFETCH('mydata', LEVEL=1)
temp = FINDGEN(100)
temp++
temp += 57
temp[4] = 1000
```

This code alters the value of the local variable `temp`, but the original variable `mydata` remains unaltered. The first line makes a copy of `mydata`, the second line discards that copy, and the remaining lines operate on the local variable.

`SCOPE_VARFETCH` allows transparent access to variables from other scopes as long as you call it in every place where the variable is needed. Any assignment of the result from `SCOPE_VARFETCH` to another variable will create a copy. Altering the value of this copy will not alter the original.

Examples

Example 1: Creating a New Variable in the Main Level

The following creates a variable in the main program scope, named `newdata`. It sets the value using the `DIST` function, and then uses `SURFACE` to display it:

```
(SCOPE_VARFETCH('newdata', /ENTER, LEVEL=1)) = DIST(128)
SURFACE, SCOPE_VARFETCH('newdata', LEVEL=1)
```

Example 2: Using the REF_EXTRA Keyword

The following procedure will print the values of all keywords passed to it via the by-reference `_REF_EXTRA` keyword inheritance mechanism. In addition, it displays the name(s) by which that variable is known in all currently active interpreter frames:

```
PRO example, _REF_EXTRA=ref

  FOR ridx = 0, n_elements(ref)-1 DO BEGIN
    kw = ref[ridx]
    PRINT, 'Value of _REF_EXTRA keyword ', kw, ':'
    HELP, SCOPE_VARFETCH(kw, /REF_EXTRA)

    HELP, calls=calls
    FOR i=0, n_elements(calls)-1 DO BEGIN
      name = SCOPE_VARNAME(SCOPE_VARFETCH(kw, /REF_EXTRA), $
        LEVEL=-i)
      cname = STREGEX(calls[i], '[^ ]*', /EXTRACT)
      PRINT, format='(%" _REF_EXTRA %s name in level %s: $
        %s")', $
        kw, cname, name ? name : ''
    ENDFOR
  ENDFOR
END
```

Using this procedure, consider the output from the following IDL statements:

```
IDL> a = 1
IDL> b = 2
IDL> example, k_a=a, k_b=b
Value of _REF_EXTRA keyword K_A:
<No name>      INT      =      1
_REF_EXTRA K_A name in level EXAMPLE:
_REF_EXTRA K_A name in level $MAIN$: A
Value of _REF_EXTRA keyword K_B:
<No name>      INT      =      2
_REF_EXTRA K_B name in level EXAMPLE:
_REF_EXTRA K_B name in level $MAIN$: B
```

Version History

Introduced: 6.1

See Also

[HELP](#), [SCOPE_LEVEL](#), [SCOPE_VARNAME](#), [ROUTINE_INFO](#)

SCOPE_VARNAME

The `SCOPE_VARNAME` function returns the names of variables outside the local scope of the currently running procedure or function. Given a variable within the local scope of the currently running routine, `SCOPE_VARNAME` can supply the name under which it is known in an earlier routine in the current chain of active routines. Similarly, given the positional index of a common block variable, `SCOPE_VARNAME` can report the name that the variable was given in the routine that first defined that common block within the current IDL session.

In normal IDL operation, routines are not allowed to see variables outside their own scope. This restriction is a fundamental and important property of structured modular programming. Code that accesses non-local scope to alter variables within other active routines can quickly become extremely difficult to understand and maintain. For this reason, RSI recommends against the use of the IDL scope routines.

Despite this warning, there is an important class of application that requires the functionality provided by these routines. Programs, usually with graphical user interfaces, that import and export data from the callers' scope need to be able to access the user's data variables directly, and without requiring them to explicitly pass those variables to the application as parameters. The `SCOPE_VARFETCH` function is used to access those variables, and `SCOPE_VARNAME` is used to obtain the correct names with which to refer to them. `SCOPE_LEVEL` returns the scope level of the currently executing routine. The IDL iTools and the ENVI image processing software are examples of programs that need to be able to perform these operations.

Note

For more information, see [“Interpreter Frames, Level, and Variable Scope”](#) on page 1846.

Syntax

```
Result = SCOPE_VARNAME( [Var1[, ..., Varn]] [, COMMON=string]
                        [, COUNT=variable] [, LEVEL=value] )
```

Return Value

The value returned by `SCOPE_VARNAME` depends on whether or not the `COMMON` keyword is set, and whether or not plain arguments are provided in the call. In all cases, the result is of a scalar string or a string array of variable names.

COMMON Keyword Not Set

Arguments Present

The result is a string array, with each element giving the name of the corresponding argument variable within the scope of the routine specified by the [LEVEL](#) keyword. If an argument does not exist within the target scope, a scalar null string is returned.

Arguments Not Present

The result is a string array containing the names of all known variables within the scope of the routine specified by the [LEVEL](#) keyword. If the specified scope has no variables, a scalar null string is returned.

See the description of the [LEVEL](#) keyword for additional information.

COMMON Keyword Set

Arguments Present

Each argument supplies the positional index of a variable within the common block. The result is a string array, with each element giving the name of the corresponding variable with that index within that common block.

Arguments Not Present

The result is a string array containing the names of all known variables defined within a specified common block, given in the order in which they exist in the block.

Arguments

Var_i

If the [COMMON](#) keyword is not set, the arguments to `SCOPE_VARNAME` are variables for which names are to be returned relative to the scope specified by the [LEVEL](#) keyword.

If the [COMMON](#) keyword is set, the arguments to `SCOPE_VARNAME` are integers, giving the positional index of variables within the specified common block for which names are to be returned.

Keywords

COMMON

Set this keyword to a string containing the name of a common block to cause `SCOPE_VARNAME` to return the names of variables contained within the specified block. If `COMMON` is set, the arguments to `SCOPE_VARNAME` are integers that give the positional index of variables within the specified common block for which names are to be returned.

Note

See “[COMMON Variable Names](#)” on page 1855 for more information on the names returned.

The `COMMON` keyword cannot be used in conjunction with the [LEVEL](#) keyword.

COUNT

Set this keyword to a named variable that will contain the number of elements returned in the array value returned by `SCOPE_VARNAME`. If `SCOPE_VARNAME` returns a null string, a 0 is returned. This circumstance will occur if there are no arguments, the [COMMON](#) keyword is not set, and the specified scope has no variables.

LEVEL

Set this keyword to an integer that specifies the scope within which the desired variable should be found. The level can be either an absolute value or relative to the current scope, as shown in the following table:

Value	Meaning
0	Level 0 always refers to the frame for the currently executing routine. This is the default value.
< 0	Negative levels are a relative specification and refer to the frames of active routines relative to the currently executing routine. Level -1 is the direct caller of the current routine, -2 is the caller of the caller of the current routine, and so forth. If you specify a level that is deeper than the current call stack, <code>SCOPE_VARNAME</code> clips the value not to go past <code>\$MAIN\$</code> .

Table 8: LEVEL Values

Value	Meaning
> 0	Positive levels are an absolute specification and refer directly to the specified frame. Level 1 is \$MAIN\$, level 2 is the routine called from \$MAIN\$, and so forth. If you specify a level that is past the currently executing routine, SCOPE_VARNAME clips the value to the level of the current routine.

Table 8: LEVEL Values

The LEVEL keyword cannot be used in conjunction with the COMMON keyword.

COMMON Variable Names

Variables in IDL common blocks do not actually have names. Rather, IDL represents common blocks internally as an array of variables, and these variables are referenced by their positional index. Hence, the first variable is at position 0, the second at position 1, and so forth. When you specify a COMMON declaration in an IDL routine, you specify names to be used for these variables *within the scope of that routine*. In fact, different routines are allowed to use entirely different names to refer to a given variable. However, this can quickly become confusing, and most programmers use the same names in every case.

The first routine in which a COMMON block is defined is remembered by IDL as part of the state of that block. When another routine defines the same common block, it is allowed to omit the variable names. In this case, IDL uses the same names used in the original defining routine.

When the COMMON keyword is specified to the SCOPE_VARNAME function, the names returned are the names that were defined by the originally defining routine. Because good programming practice dictates that the same names be used everywhere, this result usually causes no confusion. However, in an environment where different names are in use in different scopes, you should be aware of the difference. For example, execute the following COMMON statement at the IDL prompt:

```
COMMON DOGS, poodle, spaniel, retriever
```

Now execute the following IDL routine:

```
PRO dogs
```

```
COMMON DOGS, breed1, breed2, breed3
```

```
PRINT, 'In this scope: ', SCOPE_VARNAME(breed1, breed2, breed3)
```

```

PRINT, 'In $MAIN$: ', SCOPE_VARNAME(breed1, breed2, breed3, $
    LEVEL=-1)
PRINT, 'In COMMON: ', SCOPE_VARNAME(COMMON='dogs', 0, 1, 2)
END

```

The following output is produced when procedure DOGS is called:

```

In this scope: BREED1 BREED2 BREED3
In $MAIN$: POODLE SPANIEL RETRIEVER
In COMMON: POODLE SPANIEL RETRIEVER

```

If you do not specify the COMMON keyword, SCOPE_VARNAME translates the names of all variables, common or regular, relative to the specified scope. Hence, the variables in the DOGS common block appear as POODLE, SPANIEL, and RETRIEVER in \$MAIN\$, and as BREED1, BREED2, and BREED3 within procedure DOGS, despite the fact that they refer to the same variables. When you direct SCOPE_VARNAME to work from the common block instead of from an interpreter frame, the function uses the names from the scope of the routine that defined the common block, which in this example is \$MAIN\$.

Following this advice can minimize confusion:

- Use the same names for common block variables in all definitions. A convenient way to do so is to take advantage of the abbreviated form of the COMMON declaration, in which second and following declarations can leave off the variable names, and IDL will substitute the names from the initial definition.
- When possible, routines that use SCOPE_VARNAME should not specify the COMMON keyword. Instead, translate their names relative to the caller's scope.

Example

The following procedure outputs the names of all variables currently defined in the IDL main program, \$MAIN\$. It also prints the name of its argument, in its own scope, in that of its caller, and in that of \$MAIN\$.

```

PRO test_scope_varname, arg

PRINT, 'All variables in $MAIN$: ', SCOPE_VARNAME(LEVEL=1)
PRINT, 'Argument name in local scope: ', SCOPE_VARNAME(arg)
PRINT, 'Argument name in caller scope: ', SCOPE_VARNAME(arg, $
    LEVEL=-1)
PRINT, 'Argument name in $MAIN$ scope: ', SCOPE_VARNAME(arg, $
    LEVEL=1)

END

```

Version History

Introduced: 6.1

See Also

[HELP](#), [SCOPE_LEVEL](#), [SCOPE_VARFETCH](#), [ROUTINE_INFO](#)

SEARCH2D

The SEARCH2D function finds “objects” or regions of similar data values within a two-dimensional array. Given a starting location and a range of values to search for, SEARCH2D finds all the cells within the array that are within the specified range and have some path of connectivity through these cells to the starting location. In addition to searching for cells within a global range of data values, SEARCH2D can also search for adjacent cells whose values deviate from their neighbors within specified tolerances.

This routine is written in the IDL language. Its source code can be found in the file `search2d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = SEARCH2D( Array, Xpos, Ypos, Min_Val, Max_Val [, DECREASE=value,  
INCREASE=value[, LPF_BAND=integer{≥3}]] [, /DIAGONAL] )
```

Return Value

SEARCH2D returns a longword array that contains a list of the array subscripts that define the located object or region. The original X and Y indices of the array subscripts returned by SEARCH2D can be found with the following IDL code:

```
index_y = Result / (SIZE(Array))[1]  
index_x = Result - (index_y * (SIZE(Array))[1])
```

where *Result* is the array returned by SEARCH2D and *Array* is the original input array. The object within *Array* can be subscripted as *Array*(*Region*) or *Array*(*index_x*, *index_y*).

Arguments

Array

A two-dimensional array, of any data type, to be searched.

Xpos

The X coordinate (dimension 0 of *Array*) of the starting location.

Ypos

The Y coordinate (dimension 1 of *Array*) of the starting location.

Min_Val

The minimum data value for which to search. All array subscripts of all cells that are connected to the starting cell, and have a value between *Min_Val* and *Max_Val* (inclusive) are returned.

Max_Val

The maximum data value for which to search.

Keywords

DECREASE

This keyword and the INCREASE keyword allow you to compensate for changing intensities of data values within an object. An edge-enhanced copy of *Array* is made and compared to the original array if this keyword is set. The search is limited to pixels for which the edge-enhanced gradient value lies between the DECREASE and INCREASE values. When DECREASE or INCREASE is set, any adjacent cells are found if their corresponding data values in the edge enhanced array are greater than DECREASE and less than INCREASE. In any case, the adjacent cells will never be selected if their data values are not between *Min_Val* and *Max_Val*. The default for this keyword is 0.0 if INCREASE is specified.

INCREASE

This keyword and the DECREASE keyword allow you to compensate for changing intensities of data values within an object. An edge-enhanced copy of *Array* is made and compared to the original array if this keyword is set. The search is limited to pixels for which the edge-enhanced gradient value lies between the DECREASE and INCREASE values. When DECREASE or INCREASE is set, any adjacent cells are found if their corresponding data values in the edge enhanced array are greater than DECREASE and less than INCREASE. In any case, the adjacent cells will never be selected if their data values are not between *Min_Val* and *Max_Val*. The default for this keyword is 0.0 if DECREASE is specified.

LPF_BAND

Set this keyword to an integer value of 3 or greater to perform low-pass filtering on the edge-enhanced array. The value of LPF_BAND is used as the width of the smoothing window. This keyword is only effective when the DECREASE or INCREASE keywords are also specified. The default is no smoothing.

DIAGONAL

Set this keyword to cause SEARCH2D to find cells meeting the search criteria whose surrounding squares share a common corner. Normally, cells are considered adjacent only when squares surrounding the cells share a common edge. Setting this option requires more memory and execution time.

Examples

Find all the indices corresponding to an object in an image:

```

; Create an image with different valued regions:
img = FLTARR(512, 512)
img[3:503, 9:488] = 0.7
img[37:455, 18:438] = 0.5
img[144:388, 90:400] = 0.7
img[200:301, 1:255] = 1.0
img[155:193, 333:387] = 0.3
TVSCL, img ;Display the image.

; Search for an object starting at (175, 300) whose data values are
; between (0.6) and (0.8):
region = SEARCH2D(img, 175, 300, 0.6, 0.8, /DIAGONAL)

; Scale the background cells into the range 0 to 127:
img = BYTSCL(img, TOP=127B)

; Highlight the object region by setting it to 255:
img[region] = 255B

; Display the array with the highlighted object in it:
TVSCL, img

```

Version History

Introduced: Pre 4.0

See Also

[SEARCH3D](#)

SEARCH3D

The SEARCH3D function finds “objects” or regions of similar data values within a 3-D array of data. Given a starting location and a range of values to search for, SEARCH3D finds all the cells within the volume that are within the specified range of values and have some path of connectivity through these cells to the starting location. In addition to searching for cells within a global range of data values, SEARCH3D can also search for adjacent cells whose values deviate from their neighbors within specified tolerances.

This routine is written in the IDL language. Its source code can be found in the file `search3d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = SEARCH3D(Array, Xpos, Ypos, Zpos, Min_Val, Max_Val [, /DECREASE,
  /INCREASE [, LPF_BAND=integer{≥3}]] [, /DIAGONAL])
```

Return Value

SEARCH3D returns a longword array that contains a list of the array subscripts that define the selected object or region. The original X and Y indices of the array subscripts returned by SEARCH3D can be found with the following IDL code:

```
S = SIZE(Array)
index_z = Result / (S[1] * S[2])
index_y = (Result - (index_z * S[1] * S[2])) / S[1]
index_x = (Result - (index_z * S[1] * S[2])) - (index_y * S[1])
```

where *Result* is the array returned by SEARCH3D and *Array* is the original input volume. The object within *Array* can be subscripted as `Array[Region]` or `Array[index_x, index_y, index_z]`.

Arguments

Array

The three-dimensional array, of any data type except string, to be searched.

Xpos

The X coordinate (dimension 0 or *Array*) of the starting location.

Ypos

The Y coordinate (dimension 1 of *Array*) of the starting location.

Zpos

The Z coordinate (dimension 2 of *Array*) of the starting location.

Min_Val

The minimum data value for which to search. All array subscripts of all the cells that are connected to the starting cell, and have a value between *Min_Val* and *Max_Val* (inclusive) are returned.

Max_Val

The maximum data value for which to search.

Keywords

DECREASE

This keyword and the INCREASE keyword allow you to compensate for changing intensities of data values within an object. An edge-enhanced copy of *Array* is made and compared to the original array if this keyword is set. When DECREASE or INCREASE is set, any adjacent cells are found if their corresponding data values in the edge enhanced array are greater than DECREASE and less than INCREASE. In any case, the adjacent cells will never be selected if their data values are not between *Min_Val* and *Max_Val*. The default for this keyword is 0.0 if INCREASE is specified.

INCREASE

This keyword and the DECREASE keyword allow you to compensate for changing intensities of data values within an object. An edge-enhanced copy of *Array* is made and compared to the original array if this keyword is set. When DECREASE or INCREASE is set, any adjacent cells are found if their corresponding data values in the edge enhanced array are greater than DECREASE and less than INCREASE. In any case, the adjacent cells will never be selected if their data values are not between *Min_Val* and *Max_Val*. The default for this keyword is 0.0 if DECREASE is specified.

LPF_BAND

Set this keyword to an integer value of 3 or greater to perform low-pass filtering on the edge-enhanced array. The value of LPF_BAND is used as the width of the smoothing window. This keyword is only effective when the DECREASE or INCREASE keywords are also specified. The default is no smoothing.

DIAGONAL

Set this keyword to cause SEARCH3D to find cells meeting the search criteria whose surrounding cubes share a common corner or edge. Normally, cells are considered adjacent only when cubes surrounding the cells share a common edge. Setting this option requires more memory and execution time.

Examples

Find all the indices corresponding to an object contained in a 3-D array:

```

; Create some data.
vol = RANDOMU(s, 40, 40, 40)
vol[3:13, 1:15, 17:33] = 1.3
vol[15:25, 5:25, 15:25] = 0.2
vol[5:30,17:38,7:28] = 1.3
vol[9:23, 16:27, 7:33] = 1.5

; Search for an object starting at (6, 22, 16) whose data values
; are between (1.2) and (1.4):
region = SEARCH3D(vol, 6, 22, 16, 1.2, 1.4, /DIAGONAL)

; Scale the background cells into the range 0 to 127:
vol = BYTSCL(vol, TOP=127B)

; Highlight the object region by setting it to 255:
vol[Region] = 255B
WINDOW, 0, XSIZE=640, YSIZE=512, RETAIN=2

; Set up a 3-D view:
CREATE_VIEW, XMAX=39, YMAX=39, ZMAX=39, AX=(-30), AZ=30, ZOOM=0.8

; Display the volume with the highlighted object in it:
TVSCL, PROJECT_VOL(vol, 64, 64, 40, DEPTH_Q=0.4)

```

Version History

Introduced: Pre 4.0

See Also

[SEARCH2D](#)

SET_PLOT

The SET_PLOT procedure sets the output device used by the IDL graphics procedures. Keyword parameters control how the color tables are transferred to the newly selected graphics device. SET_PLOT performs the following actions:

- It sets the read-only system variable !D to reflect the configuration of the new device.
- It sets the default color !P.COLOR to the maximum color index minus one or, in the case of devices with white backgrounds, such as PostScript, to 0 (black).
- If the COPY keyword is set, the device color tables are copied directly from IDL's internal color tables. If the new device's color tables contain more indices than those of the old device, the new device's tables are not completely filled.
- If the INTERPOLATE keyword is set, the internal color tables are interpolated to fill the range of the new device.
- It sets the clipping rectangle to the entire device surface.

Warning

After calling SET_PLOT to change graphics devices, the scaling contained in the axis structures !X, !Y, and !Z is invalid. Any routines that rely on data coordinates should not be called until a new data coordinate system has been established. Be careful when switching devices as the number of color indices frequently differs between devices. When in doubt, reload the color table of the new device explicitly.

Syntax

```
SET_PLOT, Device [, /COPY] [, /INTERPOLATE]
```

Arguments

Device

A scalar string containing the name of the device to use. The case of *Device* is ignored by IDL. See [Appendix A, "IDL Graphics Devices"](#) for a list of device names.

Keywords

COPY

Set this keyword to copy the device's color table from the internal color table, preserving the current color mapping. The default is not to load the color table upon selection.

Warning

Unless this keyword is set, IDL's internal color tables will incorrectly reflect the state of the device's color tables until they are reloaded by TVLCT or the LOADCT procedure. Assuming that the previously-selected device's color table contains M elements, and the new device's color table contains N elements, then the minimum of M and N elements are loaded.

INTERPOLATE

Set this keyword to indicate that the current contents of the internal color table should be interpolated to cover the range of the newly-selected device. Otherwise, the internal color tables are not changed.

Examples

Change the IDL graphics device to PostScript by entering:

```
SET_PLOT, 'PS'
```

After changing the plotting device, all graphics commands are sent to that device until changed again by another use of the SET_PLOT routine.

Version History

Introduced: Original

SET_SHADING

The SET_SHADING procedure modifies the light source shading parameters that affect the output of SHADE_SURF and POLYSHADE. Parameters can be changed to control the light-source direction, shading method, and the rejection of hidden surfaces. SET_SHADING first resets the shading parameters to their default values. The parameter values specified in the call then overwrite the default values. To reset all parameters to their default values, simply call this procedure with no parameters.

Syntax

```
SET_SHADING [, /GOURAUD] [, LIGHT=[x, y, z]] [, /REJECT]  
            [, VALUES=[darkest, brightest]]
```

Arguments

None.

Keywords

GOURAUD

This keyword controls the method of shading the surface polygons by the POLYSHADE procedure. The SHADE_SURF procedure always uses the Gouraud method. Set this keyword to a nonzero value (the default), to use Gouraud shading. Set this keyword to zero to shade each polygon with a constant intensity.

Gouraud shading interpolates intensities from each vertex along each edge. Then, when scan converting the polygons, the shading is interpolated along each scan line from the edge intensities. Gouraud shading is slower than constant shading but usually results in a more realistic appearance.

LIGHT

A three-element vector that specifies the direction of the light source. The default light source vector is [0,0,1], with the light rays parallel to the Z axis.

REJECT

Set this keyword (the default) to reject polygons as being hidden if their vertices are ordered in a clockwise direction as seen by the viewer. This keyword should always be set when rendering enclosed solids whose original vertex lists are in

counterclockwise order. When rendering surfaces that are not closed or are not in counterclockwise order this keyword can be set to zero although shading anomalies at boundaries between visible and hidden surfaces may occur.

VALUES

A two-element array that specifies the range of pixel values (color indices) to use. The first element is the color index for the darkest pixel. The second element is the color index for the brightest pixel. For example, to render a shaded surface with the darkest shade set to pixel value 100 and the brightest value set to 150, use the commands:

```
SET_SHADING, VALUES=[100, 150]  
SHADE_SURF, dataset
```

Examples

Change the light source so that the light rays are parallel to the X axis:

```
SET_SHADING, LIGHT = [1, 0, 0]
```

Version History

Introduced: Pre 4.0

See Also

[POLYSHADE](#), [SHADE_SURF](#)

SETENV

The SETENV procedure adds or changes an environment string in the process environment.

Syntax

SETENV, *Environment_Expression*

Arguments

Environment_Expression

A scalar string or string array containing an environment expressions to be added to the environment.

Keywords

None.

Examples

Change the current shell variable by entering:

```
SETENV, 'SHELL=/bin/sh'
```

Make sure to eliminate any whitespace around the equal sign:

```
; This is an incorrect usage--there are spaces around the equal  
; sign:  
SETENV, 'VAR = H:\rsi'
```

```
; This is correct--VAR is set to H:\rsi:  
SETENV, 'VAR=H:\rsi'
```

Version History

Introduced: Original

See Also

[GETENV](#)

SETUP_KEYS

The `SETUP_KEYS` procedure sets function keys for use with UNIX versions of IDL when used with the standard tty command interface.

Under UNIX, the number of function keys, their names, and the escape sequences they send to the host computer vary enough between various keyboards that IDL cannot be written to understand all keyboards. Therefore, IDL provides a very general routine named `DEFINE_KEY` that allows the user to specify the names and escape sequences of function keys.

`SETUP_KEYS` provides a convenient interface to `DEFINE_KEY`, using user input (via the keywords described below), the `TERM` environment variable and the type of machine the current IDL is running on to determine what kind of keyboard you are using, and then uses `DEFINE_KEY` to enter the proper definitions for the function keys.

The new mappings for the keys can be viewed using the command

```
HELP, /KEYS.
```

The need for `SETUP_KEYS` has diminished in recent years because most UNIX terminal emulators have adopted the ANSI standard for function keys, as represented by VT100 terminals and their many derivatives, as well as `xterm` and the newer CDE based `dterm`.

The current version of IDL already knows the function keys of such terminals, so `SETUP_KEYS` is not required. However, `SETUP_KEYS` is still needed to define keys on non-ANSI terminals such as the Sun shelltool, SGI Iris-ansi terminal emulator, or IBM's `aixterm`.

IDL does not support the function keys from the `hpterm` terminal emulator supplied on HP systems. `Hpterm` uses non ANSI-standard escape sequences which IDL cannot parse. RSI recommends the use of the `xterm` or `dterm` terminal emulators instead.

This routine is written in the IDL language. Its source code can be found in the file `setup_keys.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SETUP_KEYS [, /ANSI] [, /EIGHTBIT] [, /SUN | , /VT200 | , /HP9000 | , /IBM |  
  , /MIPS | , /SGI] [, /APP_KEYPAD] [, /NUM_KEYPAD]
```

Arguments

None.

Keywords

Note

If no keyword is specified, `SETUP_KEYS` uses `!VERSION` to determine the type of machine running IDL. It assumes that the keyboard involved is of the same type (this assumption is correct).

ANSI

Set this keyword to establish function key definitions for ANSI keyboards.

EIGHTBIT

Set this keyword to use the 8-bit versions of the escape codes (instead of the default 7-bit) when establishing VT200 function key definitions.

SUN

Set this keyword to establish function key definitions for a Sun3 keyboard.

VT200

Set this keyword to establish function key definitions for a DEC VT200 keyboard.

HP9000

Set this keyword to establish function key definitions for an HP 9000 series 300 keyboard. Although the HP 9000 series 300 supports both `xterm` and `hterm` windows, IDL supports only user-definable key definitions in `xterm` windows—`hterm` windows use non-standard escape sequences which IDL does not attempt to handle.

IBM

Set this keyword to establish function key definitions for IBM keyboards.

MIPS

Set this keyword to establish function key definitions for a Mips RS series keyboard.

SGI

Set this keyword to establish function key definitions for SGI keyboards.

APP_KEYPAD

Set this keyword to define escape sequences for the group of keys in the numeric keypad, enabling these keys to be programmed within IDL.

NUM_KEYPAD

Set this keyword to disable programmability of the numeric keypad.

Version History

Introduced: Pre 4.0

See Also

[DEFINE_KEY](#)

SFIT

The SFIT function determines a polynomial fit to a surface and returns a fitted array. The function fitted is:

$$f(x, y) = \sum k_{j,i} \cdot x^i \cdot y^j$$

This routine is written in the IDL language. Its source code can be found in the file `sfit.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = SFIT(*Data*, *Degree* [, /IRREGULAR, KX=*variable*, /MAX_DEGREE])

Arguments

Data

The two-dimensional array of data to fit. The sizes of the dimensions might be unequal, as indicated by the **IRREGULAR** keyword.

Degree

The maximum degree of fit (in one dimension).

Keywords

IRREGULAR

If set, *Data* is a 3-by-*n* array containing the *x*, *y*, and *z* locations of *n* points sampled on the surface. If not set, *Data* is assumed to be sampled over a regular 2D grid, and should be in an *n*-by-*n* array, where the column and row subscripts implicitly contain the *x* and *y* location of the point.

KX

Set this keyword to a named variable that will contain the array of coefficients for a polynomial function of *x* and *y* to fit data. If **MAX_DEGREE** is not set, this parameter is returned as a *Degree*+1 by *Degree*+1 array. If **MAX_DEGREE** is set, this parameter is returned as a *Degree*+1 by (*Degree*+2)/2 element vector.

MAX_DEGREE

If set, *Degree* represents the maximum degree of the fitting polynomial of all dimensions combined, rather than the maximum degree of the polynomial in a single variable.

For example, if *Degree* is 2 and MAX_DEGREE is not set, then the terms returned are $[[k, y, y^2], [x, xy, xy^2], [x^2, x^2y, x^2y^2]]$. If MAX_DEGREE is set, the terms returned are in a vector, $[k, y, y^2, x, xy, x^2]$, in which no terms has a power higher than two in x and y combined, and the powers of y vary the fastest.

Examples

```

; Create a grid from zero to six radians in the X and Y directions:
X = (FINDGEN(61)/10) # REPLICATE(1,61)
Y = TRANSPOSE(X)

; Evaluate a function at each point:
F = -SIN(2*X) + COS(Y/2)

; Compute a sixth-degree polynomial fit to the function data:
result = SFIT(F, 6)

; Display the original function on the left and the fitted function
; on the right, using identical axis scaling:
WINDOW, XSIZE = 800, YSIZE = 400

; Set up side-by-side plots:
!P.MULTI = [0, 2, 1]

; Set background color to white:
!P.BACKGROUND = 255

; Set plot color to black:
!P.COLOR = 0

SURFACE, F, X, Y, ZRANGE = [-3, 3], ZSTYLE = 1
SURFACE, result, X, Y

```

The following figure shows the result of this example:

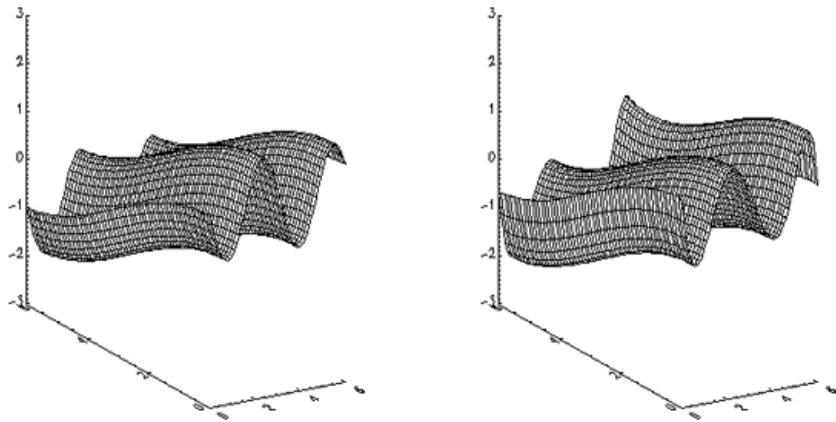


Figure 3-54: The Original Function (Left) and the Fitted Function (Right)

Version History

Introduced: Pre 4.0

See Also

[CURVEFIT](#), [GAUSSFIT](#), [LINFIT](#), [LMFIT](#), [POLY_FIT](#), [REGRESS](#), [SVDFIT](#)

SHADE_SURF

The SHADE_SURF procedure creates a shaded-surface representation of a regular or nearly-regular gridded surface with shading from either a light source model or from a user-specified array of intensities. This procedure and its parameters are similar to SURFACE. Given a regular or near-regular grid of elevations it produces a shaded-surface representation of the data with hidden surfaces removed.

The SET_SHADING procedure can be used to control the direction of the light source and other shading parameters.

If the graphics output device has scalable pixels (e.g., PostScript), the output image is scaled so that its largest dimension is less than or equal to 512 (unless the PIXELS keyword is set to some other value). This default resolution may not be high enough for some datasets. If your output looks jagged or “stair-stepped”, try specifying a larger value with the PIXELS keyword.

When outputting to a device that prints black on a white background, (e.g., PostScript), pixels that contain the background color index of 0 are set to white.

Restrictions

If the (X, Y) grid is not regular or nearly regular, errors in hidden line removal will occur. If the T3D keyword is set, the 3-D to 2-D transformation matrix contained in !P.T must project the Z axis to a line parallel to the device Y axis, or errors will occur. The SHADE_SURF_IRR procedure can be used to render many datasets that do not meet these requirements. Irregularly-gridded data can also be made interpolated to a regular grid using the TRIGRID and TRIANGULATE routines.

Syntax

```
SHADE_SURF, Z [, X, Y] [, AX=degrees] [, AZ=degrees] [, IMAGE=variable]
  [, MAX_VALUE=value] [, MIN_VALUE=value] [, PIXELS=pixels] [, /SAVE]
  [, SHADES=array] [, /XLOG] [, /YLOG]
```

```
Graphics Keywords: [, CHARSIZE=value] [, CHARTHICK=integer]
  [, COLOR=value][, /DATA | , /DEVICE | , /NORMAL] [, FONT=integer]
  [, /NODATA] [, POSITION=[X0, Y0, X1, Y1]] [, SUBTITLE=string] [, /T3D]
  [, THICK=value] [, TICKLEN=value] [, TITLE=string]
  [, {X | Y | Z}CHARSIZE=value]
  [, {X | Y | Z}GRIDSTYLE=integer{0 to 5}]
  [, {X | Y | Z}MARGIN=[left, right]]
  [, {X | Y | Z}MINOR=integer]
```

```
[, {X | Y | Z}RANGE=[min, max]]
[, {X | Y | Z}STYLE=value]
[, {X | Y | Z}THICK=value]
[, {X | Y | Z}TICKFORMAT=string]
[, {X | Y | Z}TICKINTERVAL=value]
[, {X | Y | Z}TICKLAYOUT=scalar]
[, {X | Y | Z}TICKLEN=value]
[, {X | Y | Z}TICKNAME=string_array]
[, {X | Y | Z}TICKS=integer]
[, {X | Y | Z}TICKUNITS=string]
[, {X | Y | Z}TICKV=array]
[, {X | Y | Z}TICK_GET=variable]
[, {X | Y | Z}TITLE=string]
[, ZVALUE=value{0 to 1}]
```

Arguments

Z

The two-dimensional array to be displayed. If X and Y are provided, the surface is plotted as a function of the (X, Y) locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of Z .

This argument is converted to double-precision floating-point before plotting. Plots created with SHADE_SURF are limited to the range and precision of double-precision floating-point values.

X

A vector or two-dimensional array specifying the X coordinates of the grid. If this argument is a vector, each element of X specifies the X coordinate for a column of Z (e.g., $X[0]$ specifies the X coordinate for $Z[0, *]$). If X is a two-dimensional array, each element of X specifies the X coordinate of the corresponding point in Z (X_{ij} specifies the X coordinate for Z_{ij}).

This argument is converted to double-precision floating-point before plotting.

Y

A vector or two-dimensional array specifying the Y coordinates of the grid. If this argument is a vector, each element of Y specifies the Y coordinate for a row of Z (e.g., $Y[0]$ specifies the Y coordinate for $Z[* , 0]$). If Y is a two-dimensional array, each element of Y specifies the Y coordinate of the corresponding point in Z (Y_{ij} specifies the Y coordinate for Z_{ij}).

This argument is converted to double-precision floating-point before plotting.

Keywords

AX

This keyword specifies the angle of rotation, about the X axis, in degrees towards the viewer. This keyword is effective only if !P.T3D and the T3D keyword are not set. If !P.T3D is set, the three-dimensional to two-dimensional transformation used by SURFACE is contained in the 4 by 4 array !P.T.

The surface represented by the two-dimensional array is first rotated, AZ (see below) degrees about the Z axis, then by AX degrees about the X axis, tilting the surface towards the viewer ($AX > 0$), or away from the viewer.

The AX and AZ keyword parameters default to +30 degrees if omitted.

The three-dimensional to two-dimensional transformation represented by AX and AZ, can be saved in !P.T by including the SAVE keyword.

AZ

This keyword specifies the counterclockwise angle of rotation about the Z axis. This keyword is effective only if !P.T3D is not set. The order of rotation is AZ first, then AX.

IMAGE

A named variable into which an image containing the shaded surface is stored. If this keyword is omitted, the image is displayed but not saved.

MAX_VALUE

The maximum value to be plotted. If this keyword is present, data values greater than the value of MAX_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

MIN_VALUE

The minimum value to be plotted. If this keyword is present, data values less than the value of MIN_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point](#)

[Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

PIXELS

Set this keyword to a scalar value that specifies the maximum size of the image dimensions, in pixels. PIXELS only applies when the output device uses scalable pixels (e.g., the PostScript device). Use this keyword to increase the resolution of the output image if the default looks jagged or “stair-stepped”.

SAVE

Set this keyword to save the 3-D to 2-D transformation matrix established by SHADE_SURF in the system variable field !P.T. Use this keyword when combining the output of SHADE_SURF with the output of other routines in the same plot.

SHADES

An array expression, of the same dimensions as Z, that contains the color index at each point. The shading of each pixel is interpolated from the surrounding SHADE values. If this parameter is omitted, light-source shading is used. For most displays, this parameter should be scaled into the range of bytes.

Warning

When using the SHADES keyword on True Color devices, we recommend that decomposed color support be turned off, by setting DECOMPOSED=0 for [DEVICE](#).

XLOG

Set this keyword to specify a logarithmic X axis.

YLOG

Set this keyword to specify a logarithmic Y axis.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above. [CHARSIZE](#), [CHARTHICK](#), [COLOR](#), [DATA](#), [DEVICE](#), [FONT](#), [NODATA](#), [NORMAL](#), [POSITION](#), [SUBTITLE](#), [T3D](#), [THICK](#), [TICKLEN](#), [TITLE](#), [\[XYZ\]CHARSIZE](#), [\[XYZ\]GRIDSTYLE](#), [\[XYZ\]MARGIN](#), [\[XYZ\]MINOR](#), [\[XYZ\]RANGE](#), [\[XYZ\]STYLE](#), [\[XYZ\]THICK](#), [\[XYZ\]TICKFORMAT](#), [\[XYZ\]TICKINTERVAL](#), [\[XYZ\]TICKLAYOUT](#), [\[XYZ\]TICKLEN](#),

[\[XYZ\]TICKNAME](#), [\[XYZ\]TICKS](#), [\[XYZ\]TICKUNITS](#), [\[XYZ\]TICKV](#),
[\[XYZ\]TICK_GET](#), [\[XYZ\]TITLE](#), [ZVALUE](#).

Examples

```
; Create a simple dataset:  
D = DIST(40)  
; Display the dataset as a light-source shaded surface:  
SHADE_SURF, D, TITLE = 'Shaded Surface'
```

Instead of light-source shading, an array of the same size as the elevation dataset can be used to color the surface. This technique creates four-dimensional displays.

```
; Create an array of shades to use:  
S = SIN(D)  
  
; Now create a new shaded surface that uses the array of shading  
; values instead of the light source:  
SHADE_SURF, D, SHADES = BYTSCL(S)
```

Note that the BYTSCL function is used to scale S into the range of bytes.

Version History

Introduced: Original

See Also

[ISURFACE](#), [POLYSHADE](#), [SET_SHADING](#), [SHADE_VOLUME](#), [SURFACE](#)

SHADE_SURF_IRR

The SHADE_SURF_IRR procedure creates a shaded surface representation of an irregularly gridded elevation dataset.

The data must be representable as an array of quadrilaterals. This routine should be used when the (*X*, *Y*, *Z*) arrays are too irregular to be drawn by SHADE_SURF, but are still semi-regular.

This routine is written in the IDL language. Its source code can be found in the file `shade_surf_irr.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SHADE_SURF_IRR, Z, X, Y [, AX=degrees] [, AZ=degrees] [, IMAGE=variable]
  [, PLIST=variable] [, /T3D]
```

Arguments

Z

An $n \times m$ array of elevations.

X

An $n \times m$ array containing the *X* location of each *Z* value.

Y

An $n \times m$ array containing the *Y* location of each *Z* value.

The grid described by *X* and *Y* must consist of quadrilaterals, must be semi-regular, and must be in “clockwise” order. Clockwise ordering means that:

```
  ;for all j
  x[i,j] <= x[i+1, j]
```

and

```
  ;for all i
  y[i,j] <= y[i, j+1]
```

Keywords

AX

The angle of rotation about the X axis. The default is 30 degrees.

AZ

The angle of rotation about the Z axis. The default is 30 degrees.

IMAGE

Set this keyword to a named variable that will contain the resulting shaded surface image. The variable is returned as a byte array of the same size as the currently selected graphics device.

PLIST

Set this keyword to a named variable that will contain the polygon list on return. This feature is useful when you want to make a number of images from the same set of vertices and polygons.

T3D

Set this keyword to indicate that the generalized transformation matrix in !P.T is to be used (in which case the keyword values for AX and AZ are ignored)

Examples

The following example creates a semi-regular data set in the proper format and displays the resulting irregular surface.

```

; Create some elevation data:
z = DIST(10,10)*100.0
; Create arrays to hold X and Y data:
x = FLTARR(10,10) & y = FLTARR(10,10)
; Ensure that X and Y arrays are in "clockwise" order:
FOR i = 0,9 DO x[0:9,i] = FINDGEN(10)
FOR j = 0,9 DO y[j,0:9] = FINDGEN(10)
; Make X and Y arrays irregular:
x = x + RANDOMU(seed,10,10)*0.49
y = y + RANDOMU(seed,10,10)*0.49
; Display the irregular surface:
SHADE_SURF_IRR, z, x, y

```

Version History

Introduced: Pre 4.0

See Also

[SHADE_SURF](#), [TRIGRID](#)

SHADE_VOLUME

Given a 3-D volume and a contour value, SHADE_VOLUME produces a list of vertices and polygons describing the contour surface. This surface can then be displayed as a shaded surface by the POLYSHADE procedure. Shading is obtained from either a single light-source model or from user-specified values.

SHADE_VOLUME computes the polygons that describe a three dimensional contour surface. Each volume element (voxel) is visited to find the polygons formed by the intersections of the contour surface and the voxel edges. The method used by SHADE_VOLUME is that of Klemp, McIrvin and Boyd, 1990: "PolyPaint—A Three-Dimensional Rendering Package," *American Meteorology Society Proceedings, Sixth International Conference on Interactive Information and Processing Systems*. This method is similar to the marching cubes algorithm described by Lorenson and Cline, 1987: "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics 21*, 163-169.

This routine is limited to processing datasets that will fit in memory.

Syntax

```
SHADE_VOLUME, Volume, Value, Vertex, Poly [, /LOW] [, SHADES=array]
    [, /VERBOSE] [, XRANGE=vector] [, YRANGE=vector] [, ZRANGE=vector]
```

Arguments

Volume

A three-dimensional array that contains the dataset to be contoured. If the *Volume* array is dimensioned (D_0 , D_1 , D_2), the resulting vertex coordinates are as follows:

$$0 < X < D_0 - 1; 0 < Y < D_1 - 1; 0 < Z < D_2 - 1.$$

If floating-point NaN values are present in *Volume*, then SHADE_VOLUME may generate inconsistent surfaces and may return NaN values in the *Vertex* argument. The surfaces generated by SHADE_VOLUME may also vary across platforms if NaN data is present in the *Volume* parameter.

Value

The scalar contour value. This value specifies the constant-density surface (also called an isosurface) to be rendered.

Vertex

The name of a variable to receive the vertex array. On output, this variable is set to a $(3, n)$ floating-point array, suitable for input to POLYSHADE.

Poly

A named variable to receive the polygon list, an n -element, longword array. This list describes the vertices of each polygon and is suitable for input to POLYSHADE. The vertices of each polygon are listed in counterclockwise order when observed from outside the surface. The vertex description of each polygon is a vector of the form: $[n, i_0, i_1, \dots, i_{n-1}]$ and the *Poly* array is the concatenation of the lists of each polygon. For example, when rendering a pyramid consisting of four triangles, *Poly* would contain 16 elements, made by concatenating four, four-element vectors of the form $[3, V_0, V_1, V_2]$. $V_0, V_1,$ and V_2 are the indices of the vertices describing each triangle.

Keywords

LOW

Set this keyword to display the low side of the contour surface (i.e., the contour surfaces enclose high data values). If this keyword is omitted or is 0, the high side of the contour surface is displayed and the contour encloses low data values. If this parameter is incorrectly specified, errors in shading will result.

SHADES

An optional array, converted to byte type before use, that contains the user-specified shading color index for each voxel. This array must have the same dimensions as *Volume*. On exit, this array is replaced by another array, that contains the shading value for each vertex, contained in *Vertex*.

Warning

When using the SHADES keyword on True Color devices, we recommend that decomposed color support be turned off, by setting `DECOMPOSED=0` for [DEVICE](#).

VERBOSE

Set this keyword to print a message indicating the number of polygons and vertices that are produced.

XRANGE

An optional two-element vector that contains the limits, over the first dimension, of the sub-volume to be considered.

YRANGE

An optional two-element vector that contains the limits, over the second dimension, of the sub-volume to be considered.

ZRANGE

An optional two-element vector containing the limits, over the third dimension, of the sub-volume to be considered.

Examples

The following procedure shades a volume passed as a parameter. It uses the `SCALE3` procedure to establish the viewing transformation. It then calls `SHADE_VOLUME` to produce the vertex and polygon lists, and `POLYSHADE` to draw the contour surface.

```

PRO SHOWVOLUME, vol, thresh, LOW = low
  ; Get the dimensions of the volume:
  s = SIZE(vol)
  ; Error, must be a 3-D array:
  IF s[0] NE 3 THEN MESSAGE, 'Error: vol must be a 3-D array'
  ; Establish the 3-D transformation and coordinate ranges:
  SCALE3, XRANGE=[0, S[1]], YRANGE=[0, S[2]], ZRANGE=[0, S[3]]
  ; Default = view high side of contour surface:
  IF N_ELEMENTS(low) EQ 0 THEN low = 0
  ; Produce vertices and polygons:
  SHADE_VOLUME, vol, thresh, v, p, LOW = low
  ; Produce image of surface and display:
  TV, POLYSHADE(v, p, /T3D)
END

```

Version History

Introduced: Pre 4.0

See Also

[INTERVAL_VOLUME](#), [IVOLUME](#), [POLYSHADE](#), [SHADE_SURF](#), [XVOLUME](#)

SHIFT

The SHIFT function shifts elements of vectors or arrays along any dimension by any number of elements. Positive shifts are to the right while left shifts are expressed as a negative number. All shifts are circular.

Elements shifted off one end wrap around and are shifted onto the other end. In the case of vectors the action of SHIFT can be expressed:

$$\text{Result}_{(i+s) \bmod n} = \text{Array}_i \text{ for } (0 \leq i < n)$$

where s is the amount of the shift, and n is the number of elements in the array.

Syntax

$$\text{Result} = \text{SHIFT}(\text{Array}, S_1, \dots, S_n)$$

Return Value

The result is a vector or array of the same structure and type as *Array*.

Arguments

Array

The array to be shifted.

S_i

The shift parameters. The S_i arguments can be either a single array containing the shift parameters for each dimension, or a sequence of up to eight scalar shift values. For arrays of more than one dimension, the parameter S_n specifies the shift applied to the n -th dimension. S_1 specifies the shift along the first dimension and so on. If only one shift parameter is present and the parameter is an array, the array is treated as a vector (i.e., the array is treated as having one-dimensional subscripts).

A shift specification of 0 means that no shift is to be performed along that dimension.

Keywords

None.

Examples

The following example demonstrates using SHIFT with a vector. by entering:

```
A = INDGEN(5)

; Print the original vector, the vector shifted one position to the
; right, and the vector shifted one position to the left:
PRINT, A, SHIFT(A, 1), SHIFT(A, -1)
```

IDL prints:

```
0  1  2  3  4
4  0  1  2  3
1  2  3  4  0
```

Notice how elements of the vector that shift off the end wrap around to the other end. This “wrap around” occurs when shifting arrays of any dimension.

Version History

Introduced: Original

See Also

[ISHFT](#)

SHMDEBUG

The SHMDEBUG function enables a debugging mode in which IDL prints an informational message (including a traceback) every time a variable created with the SHMVAR function loses its reference to the underlying memory segment created by. There are many reasons why a such a variable might lose its reference; some reasons have to do with the internal implementation of the IDL interpreter and are not obvious or visible to the IDL user.

Note

The SHMDEBUG debugging mode should be used for problem solving only, and should not be part of production code.

Syntax

Result = SHMDEBUG(*Enable*)

Return Value

SHMDEBUG returns the previous setting of the debugging state.

Arguments

Enable

Set this argument equal to a non-zero value to enable debugging, or to zero to disable debugging.

Keywords

None.

Examples

Create a memory segment, tie a variable to it, enable debugging, and then cause the variable to lose the reference:

```
old_debug = SHMDEBUG(1) ; Enable debug mode
SHMMAP, 'A', 100 ; 100 element floating vector
z = SHMVAR('A') ; Variable tied to segment
z[0] = FINDGEN(100) ; Does not lose reference
```

```
z = FINDGEN(100) ; Loses reference
% Variable released shared memory segment: A
% Released at: $MAIN$
```

The assignment `z[0] = FINDGEN(100)` explicitly uses subscripting to assign the `FINDGEN` value to the array. Under normal circumstances, using subscripting in this way on the left hand side of an assignment is inefficient and not recommended. In this case, however, it has the desirable side effect of causing the variable `Z` to maintain its connection to its existing underlying memory. In contrast, the second (normally more desirable) assignment without the subscript causes IDL to allocate different memory for the variable `Z`, with the side effect of losing the connection to the shared memory segment.

Version History

Introduced: 5.6

See Also

[SHMMAP](#), [SHMUNMAP](#), [SHMVAR](#)

SHMMAP

The SHMMAP procedure maps anonymous shared memory, or local disk files, into the memory address space of the currently executing IDL process. Mapped memory segments are associated with an IDL array specified by the user as part of the call to SHMMAP. The type and dimensions of the specified array determine the length of the memory segment.

The array can be of any type except pointer, object reference, or string. (Structure types are allowed as long as they do not contain any pointers, object references, or strings.) By default, the array type is single-precision floating-point; other types can be chosen by specifying the appropriate keyword.

Once such a memory segment exists, it can be tied to an actual IDL variable using the SHMVAR function, or unmapped using SHMUNMAP.

Why Use Mapped Memory?

- Shared memory is often used for interprocess communication. Any process that has a shared memory segment mapped into its address space is able to “see” any changes made by any other process that has access to the same segment. Shared memory is the default for SHMMAP, unless the FILENAME keyword is specified.
- Memory-mapped files allow you to treat the contents of a local disk file as if it were simple memory. Reads and writes to such memory are automatically written to the file by the operating system using its standard virtual memory mechanisms. Access to mapped files has the potential to be faster than standard Input/Output using Read/Write system calls because it does not go through the expensive system call interface, and because it does not require the operating system to copy data between user and kernel memory buffers when performing the I/O. However, it is not as general or flexible as the standard I/O mechanisms, and is therefore not a replacement for them.

Warning

Unlike most IDL functionality, incorrect use of SHMMAP can corrupt or even crash your IDL process. Proper use of these low level operating system features requires systems programming experience, and is not recommended for those without such experience. You should be familiar with the memory and file mapping features of your operating system and the terminology used to describe such features.

SHMMAP uses the facilities of the underlying operating system. Any of several alternatives may be used, as described in “Types Of Memory Segments” on page 1899. SHMMAP uses the following rules, in the specified order, to determine which method to use:

1. If the FILENAME keyword is present, SHMMAP creates a memory mapped file segment.
2. If the SYSV keyword is used under UNIX, a System V shared memory segment is created or attached. Use of the SYSV keyword under Windows will cause an error to be issued.
3. If none of the above options are specified, SHMMAP creates an anonymous shared memory segment. Under UNIX, this is done with Posix shared memory. Under Windows, the `CreateFileMapping()` system call is used.

Syntax

```
SHMMAP [ , SegmentName] [ ,  $D_1$ , ...,  $D_8$ ] [ , /BYTE] [ , /COMPLEX]
[ , /DCOMPLEX] [ , /DESTROY_SEGMENT] [ , DIMENSION=value]
[ , /DOUBLE] [ , FILENAME=value] [ , /FLOAT] [ , GET_NAME=value]
[ , GET_OS_HANDLE=value] [ , /INTEGER] [ , /L64] [ , /LONG]
[ , OFFSET=value] [ , OS_HANDLE=value] [ , /PRIVATE] [ , SIZE=value]
[ , /SYSV] [ , TEMPLATE=value] [ , TYPE=value] [ , /UINT] [ , /UL64]
[ , /ULONG]
```

Arguments

SegmentName

A scalar string supplying the name by which IDL will refer to the shared memory segment. This name is only used by IDL, and does not necessarily correspond to the name used for the shared memory segment by the underlying operating system. See the discussion of the `OS_HANDLE` keyword for more information on the underlying operating system name. If *SegmentName* is not specified, IDL will generate a unique name. The *SegmentName* can be obtained using the `GET_NAME` keyword.

D_i

The dimensions of the result. The D_i arguments can be either a single array containing the dimensions or a sequence of scalar dimensions. Up to eight dimensions can be specified.

Keywords

BYTE

Set this keyword to specify that the memory segment should be treated as a byte array.

COMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, single-precision floating-point array.

DCOMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, double-precision floating-point array.

DESTROY_SEGMENT

The UNIX anonymous shared memory mechanisms (Posix `shm_open()` and System V `shmget()`) create shared memory segments that are not removed from the operating system kernel until explicitly destroyed (or the system is rebooted). At any time, a client program can attach to such an existing segment, read or write to it, and then detach. This can be convenient in situations where the need for the shared memory is long lived, and programs that need it come and go. It also can create a problem, however, in that shared memory segments that are not explicitly destroyed can cause memory leaks in the operating system. Hence, it is important to properly destroy such segments when they are no longer required.

For UNIX anonymous shared memory (Posix or System V), the default behavior is for IDL to destroy any shared memory segments it created when the segments are unmapped, and not to destroy segments it did not create. The `DESTROY_SEGMENT` keyword is used to override this default: set `DESTROY_SEGMENT` to 1 (one) to indicate that IDL should destroy the segment when it is unmapped, or 0 (zero) to indicate that it should not destroy it. All such destruction occurs when the segment is unmapped (via the `SHMUNMAP` procedure) and not during the call to `SHMMAP`.

The `DESTROY_SEGMENT` keyword is ignored under the Windows operating system. Under UNIX, it is ignored for mapped files.

DIMENSION

Set this keyword equal to a vector of 1 to 8 elements specifying the dimensions of the result. Setting this keyword is equivalent to specifying an array via the *D* argument.

DOUBLE

Set this keyword to specify that the memory segment should be treated as a double-precision floating-point array.

FILENAME

By default, SHMMAP maps anonymous shared memory. Set the FILENAME keyword equal to a string containing the path name of a file to be mapped to create a memory-mapped file. A shared mapped file can serve as shared memory between unrelated processes. The primary difference between anonymous shared memory and mapped files is that mapped files require a file of the specified size to exist in the filesystem, whereas anonymous shared memory has no user-visible representation in the filesystem.

By default, files are mapped as shared, meaning that all processes that map the file will see any changes made. All changes are written back to the file by the operating system and become permanent. You must have write access to the file in order to map it as shared.

To change the default behavior, set the PRIVATE keyword. When a file is mapped privately, changes made to the file are *not* written back to the file by the operating system, and are not visible to any other processes. You do not need write access to a file in order to map it privately — read access is sufficient.

Note

The non-private form of file mapping corresponds to the MAP_SHARED flag to the UNIX `mmap()` function, or the PAGE_READWRITE to the Windows `CreateFileMapping()` system call.

FLOAT

Set this keyword to specify that the memory segment should be treated as a single-precision floating-point array.

GET_NAME

If *SegmentName* is not specified in a call to SHMMAP, IDL automatically generates a name. Set this keyword equal to a named variable that will receive the name assigned by IDL to the memory segment.

GET_OS_HANDLE

Set this keyword equal to a named variable that will receive the operating system name (or *handle*) for the memory segment. The meaning of the operating system handle depends on both the operating system and the type of memory segment used. See the description of the [OS_HANDLE](#) keyword for details.

INTEGER

Set this keyword to specify that the memory segment should be treated as an integer array.

L64

Set this keyword to specify that the memory segment should be treated as a 64-bit integer array.

LONG

Set this keyword to specify that the memory segment should be treated as a longword integer array.

OFFSET

If present and non-zero, this keyword specifies an offset (in bytes) from the start of the shared memory segment or memory mapped file that will be used as the base address for the IDL array associated with the memory segment.

Note

Most computer hardware is not able to access arbitrary data types at arbitrary memory addresses. Data must be properly aligned for its type or the program will crash with an alignment error (often called a *bus error*) when the data is accessed. The specific rules differ between machines, but in many cases the address of a data object must be evenly divisible by the size of that object. IDL will issue an error if you specify an offset that is not valid for the array specified.

Note

The actual memory mapping primitives provided by the underlying operating system require such offsets to be integer multiples of the virtual memory pagesize (sometimes called the *allocation granularity*) for the system. This value is typically a power of two such as 8K or 64K. In contrast, IDL allows arbitrary offsets as long as they satisfy the alignment constraints of the data type. This is implemented by mapping the page that contains the specified offset, and then adjusting the memory address to point at the specified byte within that page. In rounding your offset

request back to the nearest page boundary, IDL may map slightly more memory than your request would seem to require, but never more than a single page.

OS_HANDLE

Set this keyword equal to the name (or *handle*) used by the underlying operating system for the memory segment. If you do not specify the OS_HANDLE keyword, SHMMAP will under some circumstances provide a default value. The specific meaning and syntax of the OS_HANDLE depends on both the operating system and the form of memory used. See the following sections for operating-system specific behavior, and “[Types Of Memory Segments](#)” on page 1899 for behavior differences based on the form of memory used.

Posix (UNIX) Shared Memory

Use the OS_HANDLE keyword to supply a string value containing the system global name of the shared memory segment. Such names are expected to start with a slash (/) character, and not to contain any other slash characters. You can think of this as mimicking the syntax for a file in the root directory of the system, although no such file is created. See your system documentation for the `shm_open()` system call for specific details. If you do not supply the OS_HANDLE keyword, SHMMAP will create one for you by prepending a slash character to the value given by the *SegmentName* argument.

UNIX System V Shared Memory

Use the OS_HANDLE keyword to supply an integer value containing the system global identifier of an existing shared memory segment to attach to the process. If you do not supply the OS_HANDLE keyword, then SHMMAP creates a new memory segment. The identifier for this segment is available via the [GET_OS_HANDLE](#) keyword.

Windows Anonymous Shared Memory

Use the OS_HANDLE keyword to supply a global system name for the mapping object underlying the anonymous shared memory. If the OS_HANDLE keyword is not specified, SHMMAP uses the value of the *SegmentName* argument.

UNIX Memory Mapped Files

The OS_HANDLE keyword has no meaning for UNIX memory mapped files and is quietly ignored.

Windows Memory Mapped Files

Use the `OS_HANDLE` keyword to supply a global system name for the mapping object underlying the mapped file. Use of the `OS_HANDLE` will ensure that every process accessing the shared file will see a coherent view of its contents, and is thus recommended for Windows memory mapped files. However, if you do not supply the `OS_HANDLE` handle keyword for a memory mapped file, no global name is passed to the Windows operating system, and a unique mapping object for the file will be created.

PRIVATE

Set this keyword to specify that a private file mapping is required. In a private file mapping, any changes written to the mapped memory are visible only to the process that makes them, and such changes are not written back to the file. This keyword is ignored unless the `FILENAME` keyword is also present.

Mapping a file as shared requires that you have write access to the file, but a private mapping requires only read access. Use `PRIVATE` to map files for which you do not have write access, or when you want to ensure that the original file will not be altered by your process.

Note

Under UNIX, the private form of file mapping corresponds to the `MAP_PRIVATE` flag to the `mmap()` system call. Under Windows, the non-private form corresponds to the `PAGE_WRITECOPY` option to the Windows `CreateFileMapping()` system call. When your process alters data within a page of privately mapped memory, the operating system performs a *copy on write* operation in which the contents of that page are copied to a new memory page visible only to your process. This private memory usually comes from anonymous swap space or the system pagefile. Hence, private mapped files require more system resources than shared mappings.

It is possible for some processes to use private mappings to a given file while others use a public mapping to the same file. In such cases, the private mappings will see changes made by the public processes up until the moment the private process itself makes a change to the page. The pagesize granularity and timing issues between such processes can make such scenarios very difficult to control. RSI does not recommend combining simultaneous shared and private mappings to the same file.

SIZE

Set this keyword equal to a `size` vector specifying the type and dimensions to be associated with the memory segment. The format of a size vector is given in the description of the [SIZE](#) function.

SYSV

Under UNIX, the default form of anonymous memory is Posix shared memory, (`shm_open()` and `shm_unlink()`). Specify the `SYSV` keyword to use System V shared memory (`shmget()`, `shmctl()`, and `shmdt()`) instead. On systems where it is available, Posix shared memory is more flexible and has fewer limitations. System V shared memory is available on all UNIX implementations, and serves as an alternative when Posix memory does not exist, or when interfacing to exiting non-IDL software that uses System V shared memory. See “[Types Of Memory Segments](#)” on page 1899 for a full discussion.

TEMPLATE

Set this keyword equal to a variable of the type and dimensions to be associated with the memory segment.

TYPE

Set this keyword to specify the type code for the memory segment. See the description of the [SIZE](#) function for a list of IDL type codes.

UINT

Set this keyword to specify that the memory segment should be treated as a unsigned integer array.

ULONG

Set this keyword to specify that the memory segment should be treated as a unsigned longword integer array.

UL64

Set this keyword to specify that the memory segment should be treated as a unsigned 64-bit integer array.

Types Of Memory Segments

SHMMAP is a relatively direct interface to the shared memory and file mapping primitives provided by the underlying operating system. The SHMMAP interface attempts to minimize the differences between these primitives, and for simple shared memory use, it may not be necessary to fully understand the underlying mechanisms. For most purposes, however, it is necessary to understand the operating system primitives in order to understand how to use SHMMAP properly.

UNIX

In modern UNIX systems, the `mmap()` system call forms the primary basis for both file mapping and anonymous shared memory. The existence of System V shared memory, which is an older form of anonymous shared memory, adds some complexity to the situation.

UNIX Memory Mapped Files

To memory map a file under UNIX, you open the file using the `open()` system call, and then map it using `mmap()`. Once the file is mapped, you can close the file, and the mapping remains in place until explicitly unmapped, or until the process exits or calls `exec()` to run a different program.

If more than one process maps a file at the same time using the `MAP_SHARED` flag to `mmap()`, then those processes will be able to see each others' changes. Hence, memory mapped files are one form of shared memory. Although the requirement for a scratch file large enough to satisfy the mapping is inconvenient, limitations in System V shared memory have led many UNIX programmers to use memory mapped files in this way.

UNIX System V Shared Memory

Anonymous shared memory has traditionally been implemented via an API commonly referred to as System V IPC. The `shmget()` function is used to create a shared memory segment. The caller does not name the segment. Instead, the operating system assigns each such segment a unique integer ID when it is created. Once a shared memory segment exists, the `shmdt()` function can be used to map it into the address space of any process that knows the identifier. This segment persists in the OS kernel until it is explicitly destroyed via the `shmctl()` function, or until the system is rebooted. This is true even if there are no processes currently mapped to the segment. This can be convenient in situations where the need for the shared memory is long lived, and programs that need it come and go. It also can create a problem, however, since shared memory segments that are not explicitly destroyed

can cause memory leaks in the operating system. Hence, it is important to properly destroy such segments when they are no longer required.

System V shared memory has been part of UNIX for a long time. It is available on all UNIX platforms, and there is a large amount of existing code that uses it. There are, however, some limitations on its utility:

- Many systems place extremely small limits on the size allowed for such memory segments. These limits are often kernel parameters that can be adjusted by the system administrator. The details are highly system dependent. Consult your system documentation for details.
- The caller does not have the option of naming the shared memory segment. Instead, the operating system assigns an arbitrary number, which means that processes that want to map such a segment have to have a mechanism for finding the correct identifier to use before they can proceed. This, in turn, requires some additional form of interprocess communication.

RSI recommends the use of Posix shared memory instead of System V shared memory for those platforms that support it and applications that can use it. Under UNIX, SHMMAP defaults to Posix shared memory to implement anonymous shared memory. To use System V shared memory, you must specify the SYSV keyword. See the [Examples](#) section below for an example of using System V shared memory.

Posix Shared Memory

Posix shared memory is a newer alternative for anonymous shared memory. It is part of the UNIX98 standard, and although not all current UNIX systems support it, it will in time be available on all UNIX systems. Posix shared memory uses the `shm_open()` and `ftruncate()` system calls to create a memory segment that can be accessed via a file descriptor. This descriptor is then used with the `mmap()` system call to map the memory segment in the usual manner. The primary difference between this, and simply using `mmap()` on a scratch file to implement shared memory is that no scratch file is required (the disk space comes from the system swap space). As with System V shared memory, Posix shared memory segments exist in the operating system until explicitly destroyed (using the `shm_unlink()` system call). Unlike System V shared memory, but like all the other forms, Posix shared memory allows the caller to supply the name of the segment. This simplifies the situation in which multiple processes want to map the same segment. One of them creates it, and the others simply map it, all of them using the same name to reference it.

Posix shared memory is the default for SHMMAP on all UNIX platforms — even those that do not yet support it. (To use System V shared memory instead, you must

specify the `SYSV` keyword.) There are several reasons for making Posix shared memory the default for all UNIX platforms:

- To remain UNIX compliant, all platforms will have to implement the UNIX98 standard. Most have, and the remainder are currently in the process of doing so. We believe that Posix shared memory will be available on all UNIX systems very soon.
- Having different defaults for different UNIX platforms would cause unnecessary confusion; the confusion would only increase as platforms added support for Posix shared memory, causing the platform's `SHMMAP` default to change with later IDL releases. Since in most cases you need to know the underlying mechanism in use, the default should be easy to determine, and should not change over time.
- In the long run, it is desirable for the best option to be the default.

Microsoft Windows

Under Microsoft Windows, the `CreateFileMapping()` system call forms the basis for shared memory as well as memory mapped files. To map a file, you open the file and then pass the handle for that file to `CreateFileMapping()`. To create a region of anonymous mapped memory instead of a mapped file, you pass a special file handle (`0xffffffff`) to `CreateFileMapping()`. In this case, the disk space used to back the shared memory is taken from the system pagefile.

`CreateFileMapping()` accepts an optional parameter (`lpname`), which if present, is used to give the resulting memory mapping object a system global name. If you specify such a name, and a mapping object with that name already exists, you will receive a handle to the existing mapping object. Otherwise, `CreateFileMapping()` creates a new mapping object for the file. Hence, to create anonymous (no file) shared memory between unrelated processes, IDL calls `CreateFileMapping()` with the special `0xffffffff` file handle, and specifies a global name for it.

A global name (supplied via the `OS_HANDLE` keyword) is the only name by which an anonymous shared memory segment can be referenced within the system. Global names are not required for memory mapped files, because each process can create a separate mapping object and use it to refer to the same file. Although this does allow the unrelated processes to see each others' changes, their views of the file will not be *coherent* (that is, identical). With coherent access, all processes see exactly the same memory at exactly the same time because they are all mapping the same physical page of memory. To get coherent access to a memory mapped file, every process should specify the `OS_HANDLE` keyword to ensure that they use the same mapping object. Coherence is only an issue when the contents of the file are altered; when using read-only access to a mapped file, you need not be concerned with this issue.

The Windows operating system automatically destroys a mapping object when the last process with an open handle to it closes that handle. Destruction of the mapping object may be the result of an explicit call to `CloseHandle()`, or may involve an implicit close that happens when the process exits. This differs from the UNIX behavior for anonymous shared memory, and consequently the benefits and disadvantages are reversed. The advantage is that it is not possible to forget to destroy a mapping object, and end up with the operating system holding memory that is no longer useful, but which cannot be freed. On the other hand, you must ensure that at least one open handle to the object is open at all times, or the system might free an object that you intended to use again.

Note

Under Windows, when attaching to an existing memory object by providing the global segment name, IDL is not able to verify that the memory segment returned by the operating system is large enough to satisfy the IDL array specified to `SHMMAP` for its type and size. If the segment is not large enough, the IDL program will crash with an illegal memory access exception when it attempts to access memory addresses beyond the end of the segment. Hence, the IDL user must ensure that such pre-existing memory segments are long enough for the specified IDL array.

Reference Counts And Memory Segment Lifecycle

You can see a list of all current memory segments created with `SHMMAP` by issuing the statement

```
HELP , /SHARED_MEMORY
```

To access a current segment, it must be tied to an IDL variable using the `SHMVAR` function. IDL maintains a reference count of the number of variables currently accessing each memory segment, and does not allow a memory segment to be removed from the IDL process as long as variables that reference it still exist.

`SHMMAP` will not allow you to create a new memory segment with the same *SegmentName* as an existing segment. You should therefore be careful to pick unique segment names. One way to ensure that segment names are unique is to not provide the *SegmentName* argument when calling `SHMMAP`. In this case, `SHMMAP` will automatically choose a unique name, which can be obtained using the `GET_NAME` keyword.

The `SHMUNMAP` procedure is used to remove a memory segment from the IDL session. In addition, it may remove the memory segment from the system. (Whether the memory segment is removed from the system depends on the type of segment,

and on the arguments used with SHMMAP when the segment was initially attached.) If no variables from the current IDL session are accessing the segment (that is, if the IDL-maintained reference count is 0), the segment is removed immediately. If variables in the current IDL session are still referencing the segment, the segment is marked for removal when the last such variable drops its reference. Once SHMMAP is called on a memory segment, no additional calls to SHMVAR are allowed for it within the current IDL session; this means that a segment marked by SHMUNMAP as *UnmapPending* cannot be used for new variables within the current IDL session.

Note

IDL has no way to determine whether a process other than itself is accessing a shared memory segment. As a result, it is possible for IDL to destroy a memory segment that is in use by another process. The specific details depend on the type of memory segment, and the options used with SHMMAP when the segment was loaded. When creating applications that use shared memory, you should ensure that all applications that use the segment (be they instances of IDL or any other application) communicate regarding their use of the shared memory and act in a manner that avoids this pitfall.

Examples

Example 1

Create a shared memory segment of 1000000 double-precision data elements, and then fill it with a DINDGEN ramp:

```
SHMMAP, 'MYSEG', /DOUBLE, 1000000
z = SHMVAR('MYSEG')
z[0] = DINDGEN(1000000)
```

Note

When using shared memory, using the explicit subscript of the variable (z, in this case) maintains the variable's connection with the shared memory segment. When not using shared memory, assignment without subscripting is more efficient and is recommended.

Example 2

Create the same shared memory segment as the previous example, but let IDL choose the segment name:

```
SHMMAP, /DOUBLE, DIMENSION=[1000000], GET_NAME=segname
```

```
z = SHMVAR(segname)
z[0] = DINDGEN(1000000)
```

Example 3

Create the same shared memory segment as the previous example, but use a temporary file, mapped into IDL's address space, instead of anonymous shared memory. The file needs to be the correct length for the data we will be mapping onto it. We satisfy this need while simultaneously initializing it with the DINDGEN vector by writing the vector to the file. The use of the OS_HANDLE keyword improves performance and correctness under Windows while being quietly ignored under UNIX:

```
filename = FILEPATH('idl_scratch', /TMP)
OPENW, unit, filename, /GET_LUN
WRITEU, unit, DINDGEN(1000000)
CLOSE, unit
SHMMAP, /DOUBLE, DIMENSION=[1000000], GET_NAME=segname, $
FILENAME=filename, OS_HANDLE='idl_scratch'
z = SHMVAR(segname)
```

Example 4

Create an anonymous shared memory segment using UNIX System V shared memory. Use of System V shared memory differs from the other methods in two ways:

- The system identifier for the segment is a number chosen by the system instead of a name selected by the user.
- With SYSV memory, you have to explicitly indicate whether the operation is a create operation (no OS_HANDLE keyword) or merely an attach to an existing segment (OS_HANDLE is present). The other methods create the segment as needed, and will automatically attach to a memory segment with the desired operating system handle if it already exists. The SHMMAP call does not explicitly have to specify that the segment should be created.

In this example, we will use the type and size of the existing `myvar` variable to determine the size of the memory:

```
SHMMAP, TEMPLATE=myvar, GET_NAME=segname, /SYSV, $
GET_OS_HANDLE=ohandle
```

In this case, the SYSV keyword forces the use of System V shared memory. The absence of the OS_HANDLE keyword tells SHMMAP to create the segment, instead of simply mapping an existing one. In a different IDL session running on the same

machine, if you knew the proper `OS_HANDLE` value for this segment, you could attach to the segment created above as follows:

```
SHMMAP, TEMPLATE=myvar, GET_NAME=segname, /SYSV, $  
OS_HANDLE=oshandle
```

In this case, the `OS_HANDLE` keyword tells `SHMMAP` the identifier of the memory segment, causing it to attach to the existing segment instead of creating a new one.

Version History

Introduced: 5.6

`FILENAME` and `PRIVATE` keywords added: 6.0

See Also

[SHMDEBUG](#), [SHMUNMAP](#), [SHMVAR](#)

SHMUNMAP

The SHMUNMAP procedure is used to remove a memory segment previously created by SHMMAP from the IDL session. In addition, it may remove the memory segment from the system. (Whether the memory segment is removed from the system depends on the type of segment, and on the arguments used with SHMMAP when the segment was initially attached.) If no variables from the current IDL session are accessing the segment (that is, if the IDL-maintained reference count is 0), the segment is removed immediately. If variables in the current IDL session are still referencing the segment, the segment is marked for removal when the last such variable drops its reference.

During this *UnmapPending* phase:

- The segment still exists in the system, so attempts to use SHMMAP to create a new segment with the same *SegmentName* will fail.
- Additional calls to SHMVAR to attach new variables to this segment will fail.

Note

IDL has no way to determine whether a process other than itself is accessing a shared memory segment. As a result, it is possible for IDL to destroy a memory segment that is in use by another process. The specific details depend on the type of memory segment, and the options used with SHMMAP when the segment was loaded. When creating applications that use shared memory, you should ensure that all applications that use the segment (be they instances of IDL or any other application) communicate regarding their use of the shared memory and act in a manner that avoids this pitfall.

Syntax

SHMUNMAP, *SegmentName*

Arguments

SegmentName

A scalar string containing the IDL name for the shared memory segment, as assigned by SHMMAP.

Keywords

None.

Examples

To destroy a memory segment previously created by SHMMAP with the segment name `myseg`:

```
SHMUNMAP, 'myseg'
```

Version History

Introduced: 5.6

See Also

[SHMDEBUG](#), [SHMMAP](#), [SHMVAR](#)

SHMVAR

The SHMVAR function creates an IDL array variable that uses the memory from a current mapped memory segment created by the SHMMAP procedure. Variables created by SHMVAR are used in much the same way as any other IDL variable, and provide the IDL user with the ability to alter the contents of anonymous shared memory or memory mapped files.

By default, the variable created by SHMVAR is given the type and dimensions that were specified to SHMMAP when the memory segment was created. However, this default can be changed by SHMVAR via a variety of keywords as well as via the D_i arguments. The created array can be of any type except for pointer, object reference, or string. Structure types are allowed as long as they do not contain any pointers, object references, or strings.

Syntax

```
Result = SHMVAR(SegmentName [,  $D_1$ , ...,  $D_8$ ] [, /BYTE] [, /COMPLEX]
  [, /DCOMPLEX] [, DIMENSION=value] [, /DOUBLE] [, /FLOAT]
  [, /INTEGER] [, /L64] [, /LONG] [, SIZE=value] [, TEMPLATE=value]
  [, TYPE=value] [, /UINT] [, /UL64] [, /ULONG] )
```

Return Value

An IDL array variable that uses memory from a the specified mapped memory segment.

Arguments

SegmentName

A scalar string supplying the IDL name for the shared memory segment, as assigned by SHMMAP.

D_i

The dimensions of the result. The D_i arguments can be either a single array containing the dimensions or a sequence of scalar dimensions. Up to eight dimensions can be specified. If no dimensions are specified, the parameters specified to SHMMAP are used.

Keywords

BYTE

Set this keyword to specify that the memory segment should be treated as a byte array.

COMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, single-precision floating-point array.

DCOMPLEX

Set this keyword to specify that the memory segment should be treated as a complex, double-precision floating-point array.

DIMENSION

Set this keyword equal to a vector of 1 to 8 elements specifying the dimensions of the result. This is equivalent to the array form of the D_i plain arguments. If no dimensions are specified, the parameters specified to SHMMAP are used.

DOUBLE

Set this keyword to specify that the memory segment should be treated as a double-precision floating-point array.

FLOAT

Set this keyword to specify that the memory segment should be treated as a single-precision floating-point array.

INTEGER

Set this keyword to specify that the memory segment should be treated as an integer array.

L64

Set this keyword to specify that the memory segment should be treated as a 64-bit integer array.

LONG

Set this keyword to specify that the memory segment should be treated as a longword integer array.

SIZE

Set this keyword equal to a size vector specifying the type and dimensions to be associated with the memory segment. The format of a size vector is given in the description of the [SIZE](#) function. If no dimensions are specified, the parameters specified to SHMMAP are used.

TEMPLATE

Set this keyword equal to a variable of the type and dimensions to be associated with the memory segment. If no dimensions are specified, the parameters specified to SHMMAP are used.

TYPE

Set this keyword to specify the type code for the memory segment. See the description of the [SIZE](#) function for a list of IDL type codes.

UINT

Set this keyword to specify that the memory segment should be treated as a unsigned integer array.

ULONG

Set this keyword to specify that the memory segment should be treated as a unsigned longword integer array.

UL64

Set this keyword to specify that the memory segment should be treated as a unsigned 64-bit integer array.

Examples

See the examples given for the [SHMMAP](#) procedure.

Version History

Introduced: 5.6

See Also

[SHMDEBUG](#), [SHMMAP](#), [SHMUNMAP](#)

SHOW3

The SHOW3 procedure combines an image, a surface plot of the image data, and a contour plot of the images data in a single tri-level display.

This routine is written in the IDL language. Its source code can be found in the file `show3.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SHOW3, Image [, X, Y] [, /INTERP] [, E_CONTOUR=structure]  
      [, E_SURFACE=structure] [, SSCALE=scale]
```

Arguments

Image

The two-dimensional array to display.

X

A vector containing the X values of each column of *Image*. If the X argument is omitted, columns have values 0, 1, ..., *n*columns-1.

Y

A vector containing the Y values of each row of *Image*. If the Y argument is omitted, rows have values 0, 1, ..., *n*rows-1.

Keywords

INTERP

Set this keyword to use bilinear interpolation on the pixel display. This technique is slightly slower, but for small images, it makes a better display.

E_CONTOUR

Set this keyword equal to an anonymous structure containing additional keyword parameters that are passed to the CONTOUR procedure. Tag names in the structure should be valid keyword arguments to CONTOUR, and the values associated with each tag should be valid keyword values.

E_SURFACE

Set this keyword equal to an anonymous structure containing additional keyword parameters that are passed to the SURFACE procedure. Tag names in the structure should be valid keyword arguments to SURFACE, and the values associated with each tag should be valid keyword values.

SSCALE

Reduction scale for surface. The default is 1. If this keyword is set to a value other than 1, the array size is reduced by this factor for the surface display. That is, the number of points used to draw the wire-mesh surface is reduced. If the array dimensions are not an integral multiple of SSCALE, the image is reduced to the next smaller multiple.

Examples

```
; Create a dataset:
A = BESELJ(SHIFT(DIST(30,20), 15, 10)/2.,0)

; Show it with default display:
SHOW3, A

; Specify X axis proportional to square root of values:
SHOW3, A, SQRT(FINDGEN(30))

; Label CONTOUR lines with double size characters, and include
;downhill tick marks:
SHOW3, A, E_CONTOUR={C_CHARSIZE:2, DOWN:1}

; Draw a surface with a skirt and scale Z axis from -2 to 2:
SHOW3, A, E_SURFACE={SKIRT:-1, ZRANGE:[-2,2]}
```

Version History

Introduced: Original

See Also

[CONTOUR](#), [ICONTOUR](#), [ISURFACE](#), [SURFACE](#)

SHOWFONT

The SHOWFONT procedure displays a TrueType or vector-drawn font (from the file `hershl.chr`, located in the `resource/fonts` subdirectory of the IDL distribution) on the current graphics device.

This routine is written in the IDL language. Its source code can be found in the file `showfont.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SHOWFONT, Font, Name [, /ENCAPSULATED] [, /TT_FONT]
```

Arguments

Font

The index number of the font (may range from 3 to 29) or, if the `TT_FONT` keyword is set, a string that contains the name of the TrueType font to display.

Name

A string that contains the text of a title to appear at the top of the font display.

Keywords

ENCAPSULATED

Set this keyword, if the current graphics device is “PS”, to make encapsulated PostScript output.

TT_FONT

If this keyword is set, the specified font will be interpreted as a TrueType font.

Examples

To create a display of the Helvetica italic TrueType font on the screen:

```
SHOWFONT, 'Helvetica Italic', 'Helvetica Italic', /TT_FONT
```

To create a display of Font 3 for PostScript:

```
; Set output to PostScript:
```

```
SET_PLOT, 'PS'  
  
; Specify the output filename. If we didn't specify this, the file  
; would be saved as idl.ps by default:  
DEVICE, FILENAME='font3.ps'  
  
;Display font 3:  
SHOWFONT, 3, 'Simplex Roman'  
  
; Close the new PS file:  
DEVICE, /CLOSE
```

Version History

Introduced: Pre 4.0

See Also

[EFONT](#), [PS_SHOW_FONTS](#)

SIMPLEX

The SIMPLEX function uses the simplex method to solve linear programming problems. Given a set of N independent variables X_i , where $i = 0, \dots, N$, the simplex method seeks to maximize the following function,

$$Z = a_1 X_1 + a_2 X_2 + \dots + a_N X_N$$

with the assumption that $X_i \geq 0$. The X_i are further constrained by the following equations:

$$b_{j1} X_1 + b_{j2} X_2 + \dots + b_{jN} X_N \leq c_j \quad j = 1, 2, \dots, M_1$$

$$b_{j1} X_1 + b_{j2} X_2 + \dots + b_{jN} X_N \geq c_j \quad j = M_1 + 1, M_1 + 2, \dots, M_1 + M_2$$

$$b_{j1} X_1 + b_{j2} X_2 + \dots + b_{jN} X_N = c_j \quad j = M_1 + M_2 + 1, M_1 + M_2 + 2, \dots, M$$

where $M = M_1 + M_2 + M_3$ is the total number of equations, and the constraint values c_j must all be positive.

To solve the above problem using the SIMPLEX function, the Z equation is rewritten as a vector:

$$Z_{\text{equation}} = [a_1 \ a_2 \ \dots \ a_N]$$

The constraint equations are rewritten as a matrix with $N+1$ columns and M rows, where all of the b coefficients have had their sign reversed:

$$\text{Constraints} = \begin{bmatrix} c_1 & -b_{11} & -b_{12} \dots -b_{1N} \\ c_2 & -b_{21} & -b_{22} \dots -b_{2N} \\ : & : & : \\ : & : & : \\ c_M & -b_{M1} & -b_{M2} \dots -b_{MN} \end{bmatrix}$$

Note

The constraint matrix must be organized so that the coefficients for the less-than (<) equations come first, followed by the coefficients of the greater-than (>) equations, and then the coefficients of the equal (=) equations.

The `SIMPLEX` function is based on the routine `simplex` described in section 10.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = SIMPLEX( Zequation, Constraints, M1, M2, M3
  [, Tableau [, Izrov [, Iposv] ] ] [, /DOUBLE] [, EPS = value]
  [, STATUS = variable] )
```

Return Value

The *Result* is a vector of $N+1$ elements containing the maximum Z value and the values of the N independent X variables (the optimal feasible vector):

$$\text{Result} = \left[Z_{\max} \ X_1 \ X_2 \ \dots \ X_N \right]$$

Arguments

Zequation

A vector containing the N coefficients of the Z_{equation} to be maximized.

Constraints

An array of $N+1$ columns by M rows containing the constraint values and coefficients for the constraint equations.

M1

An integer giving the number of less-than constraint equations contained in *Constraints*. $M1$ may be zero, indicating that there are no less than constraints.

M2

An integer giving the number of greater-than constraint equations contained in *Constraints*. $M2$ may be zero, indicating that there are no greater than constraints.

M3

An integer giving the number of equal-to constraint equations contained in *Constraints*. $M3$ may be zero, indicating that there are no equal to constraints. The total of $M1 + M2 + M3$ should equal M , the number of constraint equations.

Tableau

Set this optional argument to a named variable in which to return the output array from the simplex algorithm. For more detailed discussion about this argument, see the write-up in section 10.8 of *Numerical Recipes in C*.

Izrov

Set this optional argument to a named variable in which to return the output izrov variable from the simplex algorithm. For more detailed discussion about this argument, see the write-up in section 10.8 of *Numerical Recipes in C*.

Iposv

Set this optional argument to a named variable in which to return the output iposv variable from the simplex algorithm. For more detailed discussion about this argument, see the write-up in section 10.8 of *Numerical Recipes in C*.

Keywords

DOUBLE

Set this keyword to use double-precision for computations and to return a double-precision result. Set DOUBLE to 0 to use single-precision for computations and to return a single-precision result. The default is /DOUBLE if any of the inputs are double-precision, otherwise the default is 0.

EPS

Set this keyword to a number close to machine accuracy, which is used to test for convergence at each iteration. The default is 10^{-6} .

STATUS

Set this keyword to a named variable to receive the status of the operation. Possible status values are:

Value	Description
0	Successful completion.
1	The objective function is unbounded.

Table 3-104: SIMPLEX Function Status Values

Value	Description
2	No solution satisfies the given constraints.
3	The routine did not converge.

Table 3-104: SIMPLEX Function Status Values (Continued)

Examples

The following example is taken from *Numerical Recipes in C*.

Find the Z value which maximizes the equation $Z = X_1 + X_2 + 3 X_3 - 0.5 X_4$, with the following constraints:

$$X_1 + 2X_3 \leq 740$$

$$2X_2 - 7X_4 \leq 0$$

$$X_2 - X_3 + 2X_4 \geq 0.5$$

$$X_1 + X_2 + X_3 + X_4 = 9$$

To find the solution, enter the following code at the IDL command line:

```

; Set up the Zequation with the X coefficients.
Zequation = [1,1,3,-0.5]
; Set up the Constraints matrix.
Constraints = [ $
    [740, -1, 0, -2, 0], $
    [ 0, 0, -2, 0, 7], $
    [0.5, 0, -1, 1, -2], $
    [ 9, -1, -1, -1, -1] ]
; Number of less-than constraint equations.
m1 = 2
; Number of greater-than constraint equations.
m2 = 1
; Number of equal constraint equations.
m3 = 1
;
; Call the function.
result = SIMPLEX(Zequation, Constraints, m1, m2, m3)
;
; Print out the results.
PRINT, 'Maximum Z value is: ', result[0]
PRINT, 'X coefficients are: '
PRINT, result[1:*]

```

IDL prints:

```
Maximum Z value is:      17.0250
X coefficients are:
    0.000000      3.32500      4.72500      0.950000
```

Therefore, the optimal feasible vector is $X_1 = 0.0$, $X_2 = 3.325$, $X_3 = 4.725$, and $X_4 = 0.95$.

Version History

Introduced: 5.5

See Also

[AMOEBA](#), [DFPMIN](#), [POWELL](#)

SIN

The periodic function SIN returns the trigonometric sine of X.

Syntax

Result = SIN(*X*)

Return Value

Returns the double-precision floating-point, complex or single-precision floating-point value.

Arguments

X

The angle for which the sine is desired, specified in radians. If *X* is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. When applied to complex numbers:

$$\text{SIN}(x) = (\text{EXP}(i*x) + 1/\text{EXP}(i*x))/(2*i)$$

where *i* is defined as COMPLEX(0, 1).

If input argument *X* is an array, the result has the same structure, with each element containing the sine of the corresponding element of *X*.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To find the sine of 0.5 radians and print the result, enter:

```
PRINT, SIN(0.5)
```

The following example plots the SIN function between 0 and 2π with 100 intervals:

```
X = 2*!PI/100 * FINDGEN(100)  
PLOT, X, SIN(X)
```

Note

!PI is a read-only system variable that contains the single-precision value for π .

Version History

Introduced: Original

See Also

[ASIN](#), [SINH](#)

SINDGEN

The SINDGEN function returns a string array with the specified dimensions. Each element of the array is set to the string representation of the value of its one-dimensional subscript, using IDL's default formatting rules.

Syntax

$$\text{Result} = \text{SINDGEN}(D_1 [, \dots, D_8])$$

Return Value

Returns a string array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Examples

To create S, a six-element string vector with each element set to the string value of its subscript, enter:

```
S = SINDGEN(6)
```

Version History

Introduced: Original

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [L64INDGEN](#), [LINDGEN](#), [UINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

SINH

The SINH function returns the hyperbolic sine of X .

Syntax

$Result = \text{SINH}(X)$

Return Value

Returns the double-precision floating-point, complex or single-precision floating-point value.

Arguments

X

The angle for which the hyperbolic sine is desired, specified in radians. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. SINH is defined as:

$$\sinh x = (e^x - e^{-x}) / 2$$

If X is an array, the result has the same structure, with each element containing the hyperbolic sine of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To find the hyperbolic sine of each element in the array [.5, .2, .4] and print the result, enter:

```
PRINT, SINH([.5, .2, .4])
```

To plot the SINH function between 0 and 2π with 100 intervals, enter:

```
X = 2*!PI/100 * FINDGEN(100)  
PLOT, X, SINH(X)
```

Note

!PI is a read-only system variable that contains the single-precision value of π .

Version History

Introduced: Original

See Also

[ASIN](#), [SIN](#)

SIZE

The `SIZE` function returns size and type information for its argument if no keywords are set. If a keyword is set, `SIZE` returns the specified quantity.

Syntax

```
Result = SIZE( Expression [, /L64] [, /DIMENSIONS | , /FILE_LUN |
, /FILE_OFFSET | , /N_DIMENSIONS | , /N_ELEMENTS | , /SNAME, |
, /STRUCTURE | , /TNAME | , /TYPE] )
```

Return Value

If no keywords are set, `SIZE` returns a vector of integer type. The first element is equal to the number of dimensions of *Expression*. This value is zero if *Expression* is scalar or undefined. The next elements contain the size of each dimension, one element per dimension (none if *Expression* is scalar or undefined). After the dimension sizes, the last two elements contain the type code (zero if undefined) and the number of elements in *Expression*, respectively. The type codes are listed below.

If a keyword is set, `SIZE` returns the specified information.

IDL Type Codes and Names

The following table lists the IDL type codes and type names returned by the `SIZE` function:

Type Code	Type Name	Data Type
0	UNDEFINED	Undefined
1	BYTE	Byte
2	INT	Integer
3	LONG	Longword integer
4	FLOAT	Floating point
5	DOUBLE	Double-precision floating
6	COMPLEX	Complex floating

Table 3-105: IDL Type Codes and Names

Type Code	Type Name	Data Type
7	STRING	String
8	STRUCT	Structure
9	DCOMPLEX	Double-precision complex
10	POINTER	Pointer
11	OBJREF	Object reference
12	UINT	Unsigned Integer
13	ULONG	Unsigned Longword Integer
14	LONG64	64-bit Integer
15	ULONG64	Unsigned 64-bit Integer

Table 3-105: IDL Type Codes and Names (Continued)

Arguments

Expression

The expression for which size information is requested.

Keywords

With the exception of L64, the following keywords determine the return value of the SIZE function and are mutually exclusive — specify at most one of the following.

DIMENSIONS

Set this keyword to return the dimensions of *Expression*. If *Expression* is scalar, the result is a scalar containing a 0. For arrays, the result is an array containing the array dimensions. The result is a 32-bit integer when possible, and 64-bit integer if the number of elements in *Expression* requires it. Set L64 to force 64-bit integers to be returned in all cases. If *Expression* is undefined, IDL reports eight dimensions.

FILE_LUN

Set this keyword to return the file unit to which *Expression* is associated, if it is an IDL file variable, as created with the ASSOC function. If *Expression* is not a file variable, 0 is returned (0 is not a valid file unit for ASSOC).

FILE_OFFSET

Set this keyword to return the ASSOC file offset for *Expression*. In the case of ASSOC file variables, this is the value of the *Offset* argument specified to ASSOC when the variable was created. In the case of regular non-file variables, 0 is returned. The result will be 32-bit integer when possible, and 64-bit integer if the size of the offset requires it. Set [L64](#) to force a 64-bit integer to be returned in all cases.

L64

By default, the result of SIZE is 32-bit integer when possible, and 64-bit integer if the number of elements in *Expression* requires it. Set L64 to force 64-bit integers to be returned in all cases. In addition to affecting the default result, L64 also affects the output from the [DIMENSIONS](#), [FILE_OFFSET](#), [N_ELEMENTS](#), and [STRUCTURE](#) keywords.

Note

Only 64-bit versions of IDL are capable of creating variables requiring 64-bit SIZE output. Check the value of !VERSION.MEMORY_BITS to see if your IDL is 64-bit or not.

N_DIMENSIONS

Set this keyword to return the number of dimension in *Expression*, if it is an array. If *Expression* is scalar or undefined, 0 is returned.

N_ELEMENTS

Set this keyword to return the number of data elements in *Expression*. Setting this keyword is equivalent to using the N_ELEMENTS function. The result will be 32-bit integer when possible, and 64-bit integer if the number of elements in *Expression* requires it. Set L64 to force 64-bit integers to be returned in all cases. If *Expression* is undefined, 0 is returned.

SNAME

Set this keyword to return the name of the structure definition for *Expression*, if *Expression* is a named structure. If *Expression* is not a named structure, a null string is returned. The string returned by this keyword is the same as that returned by TAG_NAMES(/STRUCTURE_NAME).

STRUCTURE

Set this keyword to return all available information about *Expression* in a structure.

Note

Since the structure is a named structure, the size of its fields is fixed. The result is an IDL_SIZE (32-bit) structure when possible, and an IDL_SIZE64 structure otherwise. Set [L64](#) to force an IDL_SIZE64 structure to be returned in all cases.

The following are descriptions of the fields in the returned structure:

Field	Description
TYPE_NAME	Name of IDL type of <i>Expression</i>
STRUCTURE_NAME	If <i>Expression</i> is a named (not anonymous) structure, the name of the structure definition; otherwise, a null string
TYPE	Type code of <i>Expression</i>
FILE_LUN	If <i>Expression</i> is an IDL file variable, as created with the ASSOC function, the file unit to which it is associated; otherwise, 0
FILE_OFFSET	If <i>Expression</i> is an IDL file variable, as created with the ASSOC function, the offset specified to ASSOC when the variable was created; otherwise, 0
N_ELEMENTS	Number of data elements in <i>Expression</i>
N_DIMENSIONS	If <i>Expression</i> is an array, the number of dimensions; otherwise, 0
DIMENSIONS	An 8-element array containing the dimensions of <i>Expression</i>

Table 3-106: Structure Fields

TNAME

Set this keyword to return the IDL type of *Expression* as a string. See [“IDL Type Codes and Names”](#) on page 1926 for details.

TYPE

Set this keyword to return the IDL type code for *Expression*. See [“IDL Type Codes and Names”](#) on page 1926 for details. For an example illustrating how to determine

the type code of an expression, see “[Determining the Size/Type of an Array](#)” in Chapter 16 of the *Building IDL Applications* manual.

Examples

Print the size information for a 10 by 20 floating-point array by entering:

```
PRINT, SIZE(FINDGEN(10, 20))
```

IDL prints:

```
2 10 20 4 200
```

This IDL output indicates the array has 2 dimensions, equal to 10 and 20, a type code of 4, and 200 elements total.

Similarly, to print only the number of dimensions of the same array:

```
PRINT, SIZE(FINDGEN(10, 20), /N_DIMENSIONS)
```

IDL prints:

```
2
```

Version History

Introduced: Original

FILE_OFFSET and SNAME keywords added: 6.1

FILE_OFFSET and STRUCTURE_NAME fields added to STRUCTURE: 6.1

See Also

[IDL_Savefile](#), [JOURNAL](#), [RESOLVE_ALL](#), [RESTORE](#)

SKEWNESS

The SKEWNESS function computes the statistical skewness of an n -element vector. Skewness determines whether a distribution is symmetric about its maximum. Positive skewness indicates the distribution is skewed to the right, with a longer tail to the right of the distribution maximum. Negative skewness indicates the distribution is skewed to the left, with a longer tail to the left of the distribution maximum.

SKEWNESS calls the IDL function MOMENT.

Syntax

```
Result = SKEWNESS( X [, /DOUBLE] [, /NAN] )
```

Return Value

Returns the floating point or double precision statistical skewness. If the variance of the vector is zero, the skewness is not defined, and SKEWNESS returns !VALUES.F_NAN as the result.

Arguments

X

A numeric vector.

Keywords

DOUBLE

Set this keyword to force computations to be done in double-precision arithmetic.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Examples

```
; Define the n-element vector of sample data:  
x = [65, 63, 67, 64, 68, 62, 70, 66, 68, 67, 69, 71, 66, 65, 70]  
; Compute the skewness:  
result = SKEWNESS(x)  
PRINT, 'Skewness = ', result
```

IDL prints:

```
Skewness =      -0.0942851
```

Version History

Introduced: 5.1

See Also

[KURTOSIS](#), [MEAN](#), [MEANABSDEV](#), [MOMENT](#), [STDDEV](#), [VARIANCE](#)

SKIP_LUN

The SKIP_LUN procedure reads data in an open file and moves the file pointer. It is useful in situations where it is necessary to skip over a known amount of data in a file without the requirement of having the data available in an IDL variable. SKIP_LUN can skip over a fixed amount of data, specified in bytes or lines of text, or can skip over the remainder of the input file from the current position to end of file. Since SKIP_LUN actually performs an input operation to advance the file pointer, it is not as efficient as POINT_LUN for skipping over a fixed number of bytes in a disk file. For that reason, use of POINT_LUN is preferred when possible. SKIP_LUN is especially useful in situations such as:

- Skipping over a fixed number of lines of text. Since lines of text can have variable length, it can be difficult to use POINT_LUN to skip them.
- Skipping data from a file that is not a regular disk file (for example, data from an internet socket).

Syntax

```
SKIP_LUN, FromUnit, [, Num] [, /EOF] [, /LINES]  
        [, /TRANSFER_COUNT=variable]
```

Arguments

FromUnit

An integer that specifies the file unit for the file in which the file pointer is to be moved. Data in *FromUnit* is skipped, starting at the current position of the file pointer. The file pointer is advanced as data is read and skipped. The file specified by *FromUnit* must be open, and must not have been opened with the RAWIO keyword to OPEN.

Num

The amount of data to skip. This value is specified in bytes, unless the LINES keyword is specified, in which case it is taken to be the number of text lines. If *Num* is not specified, SKIP_LUN acts as if the EOF keyword has been set, and skips all data in *FromUnit* (the source file) from the current position of the file pointer to the end of the file.

If *Num* is specified and the source file comes to end of file before the specified amount of data is skipped, `SKIP_LUN` issues an end-of-file error. The `EOF` keyword alters this behavior.

Keywords

EOF

Set this keyword to ignore the value given by *Num* (if present) and instead skip all data from the current position of the file pointer in *FromUnit* and the end of the file.

Note

If `EOF` is set, no end-of-file error is issued even if the amount of data skipped does not match the amount specified by *Num*. The `TRANSFER_COUNT` keyword can be used with `EOF` to determine how much data was skipped.

LINES

Set this keyword to indicate that the *Num* argument specifies the number of lines of text to be skipped. By default, the *Num* argument specifies the number of bytes of data to skip.

TRANSFER_COUNT

Set this keyword equal to a named variable that will contain the amount of data skipped. If `LINES` is specified, this value is the number of lines of text. Otherwise, it is the number of bytes. `TRANSFER_COUNT` is primarily useful in conjunction with the `EOF` keyword. If `EOF` is not specified, `TRANSFER_COUNT` will be the same as the value specified for *Num*.

Examples

Skip the next 8 lines of text from a file:

```
SKIP_LUN, FromUnit, 8, /LINES
```

Skip the remainder of the data in a file, and use the `TRANSFER_COUNT` keyword to determine how much data was skipped:

```
SKIP_LUN, FromUnit, /EOF, TRANSFER_COUNT=n
```

Skip the remainder of the text lines in a file, and use the `TRANSFER_COUNT` keyword to determine how many lines were skipped:

```
SKIP_LUN, FromUnit, /EOF, /LINES, TRANSFER_COUNT=n
```

Version History

Introduced: 5.6

See Also

[CLOSE](#), [COPY_LUN](#), [EOF](#), [FILE_COPY](#), [FILE_LINK](#), [FILE_MOVE](#), [OPEN](#),
[POINT_LUN](#), [READ/READF](#), [WRITEU](#)

SLICER3

The IDL SLICER3 is a widget-based application to visualize three-dimensional datasets. This program supersedes the SLICER program.

This routine is written in the IDL language. Its source code can be found in the file `slicer3.pro` in the `lib` subdirectory of the IDL distribution.

Note

See [“The SLICER3 Graphical User Interface”](#) on page 1938 for details on working with volumetric data within this application.

Syntax

```
SLICER3 [, hData3D] [, DATA_NAMES=string/string_array] [, /DETACH]
      [, GROUP=widget_id] [, /MODAL]
```

Arguments

hData3D

A pointer to a three-dimensional data array, or an array of pointers to multiple three-dimensional arrays. If multiple arrays are specified, they all must have the same X, Y, and Z dimensions. If *hData3D* is not specified, SLICER3 creates a 2 x 2 x 2 array of byte data using the IDL BYTARR function. You can also load data interactively via the File menu of the SLICER3 application (see [“Examples”](#) on page 1953 for details).

Note

If data are loaded in this fashion, any data passed to SLICER3 via a pointer (or pointers) is deleted, and the pointers become invalid.

Keywords

DATA_NAMES

Set this keyword equal to a string array of names for the data. The names appear on the droplist widget for the current data. If the number of elements of DATA_NAMES is less than the number of elements in *hData3D* then default names will be generated for the unnamed data.

DETACH

Set this keyword to place the drawing area in a window that is detached from the SLICER3 control panel. The drawing area can only be detached if SLICER3 is not run as a modal application.

GROUP

Set this keyword equal to the Widget ID of an existing widget that serves as the “group leader” for the SLICER3 graphical user interface. When a group leader is destroyed, all widgets in the group are also destroyed. If SLICER3 is started from a widget application, then GROUP should *always* be specified.

MODAL

Set this keyword to block user interaction with all other widgets (and block the command line) until the SLICER3 exits. If SLICER3 is started from some other widget-based application, then it is usually advisable to run SLICER3 with the MODAL keyword set.

Note

SLICER3 modifies the current color table, as well as various elements of the plotting system (i.e., the “!X”, “!Y”, “!Z”, and “!P” system variables). If the MODAL keyword is set (recommended), then SLICER3 will, upon exit, restore these system variables (and the color tables) to the values they had when SLICER3 was started.

The SLICER3 Graphical User Interface

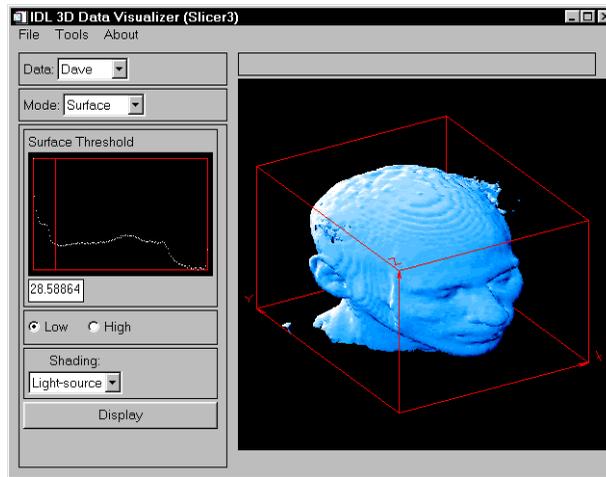


Figure 3-55: SLICER3 Graphical User Interface

The following options are available via SLICER3's graphical user interface.

File Menu

Load

Select this menu option to choose a file containing a 3-D array (or arrays) to load into SLICER3. The file must have been written in the format specified in the following table. For each data array in the file, the following values must be included. Note that the first six values are returned by the IDL SIZE function; see “Examples” on page 1953 for an example of how to create a data file suitable for SLICER3 with just a few IDL commands.

Data item	Data Type	Number of Bytes
Number of dimension in array. Note - This is always 3 for valid SLICER3 data.)	long	4

Table 3-107: SLICER3 Data File Structure

Data item	Data Type	Number of Bytes
Size of first dimension.	long	4
Size of second dimension.	long	4
Size of third dimension.	long	4
Data type (Must be type 1 through 5. See “SIZE” on page 1926 for a list of data types.)	long	4
Total number of elements (dimX, dimY, dimZ).	long	4
Number of characters in data name. (See “STRLEN” on page 2031 for the easiest way to determine this number.)	long	4
Data name	byte	strlen()
3-D data array.	varies	varies

Table 3-107: SLICER3 Data File Structure (Continued)

If multiple arrays are present in the file, they must all have the same dimensions.

Note

Files saved by the “Save Subset” operation (see below) are suitable for input via the “Load” operation.

Data files that are moved from one platform to another may not load as expected, due to byte ordering differences. See the [BYTEORDER](#) and [SWAP_ENDIAN](#) for details.

Save/Save Subset

SLICER3 must be in BLOCK mode to for this option to be available.

Select this menu option to save a subset of the 3-D data enclosed in the current block to the specified file. Subsets saved in this fashion are suitable for loading via the “Load” menu option. If multiple 3-D arrays are available when this option is selected, multiple subsets are saved to the file.

Save/Save Tiff Image

Select this menu option to save the contents of the current SLICER3 image window as a TIFF image in the specified file. When running in 8-bit mode, a “Class P” palette color TIFF file is created. In 24-bit mode, a “Class R” (interleaved by image) TIFF file is created.

Quit

Select this menu option to exit SLICER3.

Tools Menu

Erase

Select this menu option to erase the display window and delete all the objects in the display list.

Delete/...

As graphical objects are created, they are added to the display list. Select this menu option to delete a specific object from the list. When an object is deleted, the screen is redrawn with the remaining objects.

Colors/Reset Colors

Select this menu option to restore the original color scheme.

Colors/Differential Shading

Use this menu option to change the percentage of differential shading applied to the X, Y, and Z slices.

Colors/Slice/Block

Use this menu option to launch the XLOADCT application to modify the colors used for slices and blocks

Colors/Surface

Use this menu option to launch the XLOADCT application to modify the colors used for isosurfaces.

Colors/Projection

Use this menu option to launch the XLOADCT application to modify the colors used for projections.

Note

On some platforms, the selected colors may not become visible until after you exit the “XLOADCT” application.

Options

Select this menu option to display a panel that allows you to set:

- The axis visibility.
- The wire-frame cube visibility.
- The display window size.

Main Draw Window

Operations available in the Main Draw Window are dependent on the mode selected in the Mode pulldown menu. In general, when coordinate input is required from the user, it is performed by clicking a mouse button on the “surface” of the wire-frame cube that surrounds the data. This 3-D location is then used as the basis for whatever input is needed. In most cases, the “front” side of the cube is used. In a few cases, the coordinate input is on the “back” side of the cube.

Data Pulldown Menu

If multiple datasets are currently available in SLICER3, this menu allows you to select which data will be displayed in the Main Draw Window. Slices, blocks, iso-surfaces, etc. are created from the currently selected data. If only one dataset is loaded, this menu is inactive.

Mode Pulldown Menu

This menu is used to select the current mode of operation.

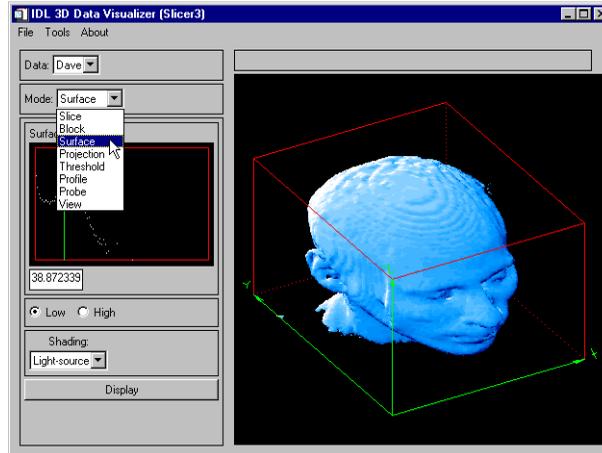


Figure 3-56: Mode Pulldown Menu

Slice Mode

To display a slice, click and drag the left mouse button on the wire-frame cube. When the button is released, a slice through the data will be drawn at that location.

Draw Radio Button

When in Draw mode, new slices will be merged into the current Z-buffer contents.

Expose Radio Button

When in Expose mode, new slices will be drawn in front of everything else.

Orthogonal Radio Button

When in Orthogonal mode, use the left mouse button in the main draw window to position and draw an orthogonal slicing plane. Clicking the right mouse button in the main draw window (or any mouse button in the small window) will toggle the slicing plane orientation.

X/Y/Z Radio Buttons

- X: This sets the orthogonal slicing plane orientation to be perpendicular to the X axis.
- Y: This sets the orthogonal slicing plane orientation to be perpendicular to the Y axis.
- Z: This sets the orthogonal slicing plane orientation to be perpendicular to the Z axis.

Oblique Radio Button

Clicking any mouse button in the small window will reset the oblique slicing plane to its default orientation.

Normal Radio Button

When in this mode, click and drag the left mouse button in the big window to set the surface normal for the oblique slicing plane.

Center Radio Button

When in this mode, click and drag the left mouse button in the big window to set the center point for the surface normal.

Display Button

Clicking this button will cause an oblique slicing plane to be drawn.

Block Mode

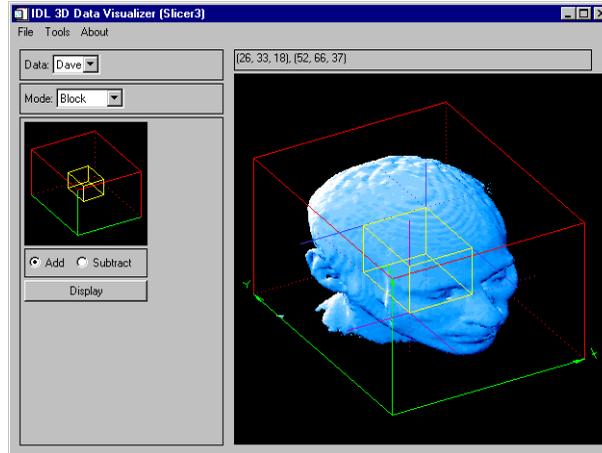


Figure 3-57: Block Mode

When in Block mode, use the left mouse button in the main draw window to set the location for the “purple” corner of the block. Use the right mouse button to locate the opposite “blue” corner of the block. When in Block mode, the “Save Subset” operation under the main “File” menu is available.

Add

When in this mode, the block will be “added” to the current Z-buffer contents.

Subtract

When in this mode, the block will be “subtracted” from the current Z-buffer contents. Subtract mode is only effective when the block intersects some other object in the display (such as an iso-surface).

Display Button

Clicking this button will cause the block to be drawn.

Surface Mode

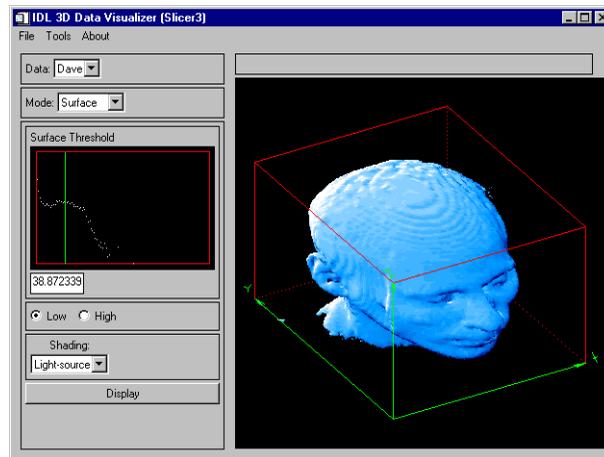


Figure 3-58: Surface Mode

An iso-surface is like a contour line on a contour map. On one side of the line, the elevation is higher than the contour level, and on the other side of the line, the elevation is lower than the contour level. An iso-surface, however, is a 3-D surface that passes through the data such that the data values on one side of the surface are higher than the threshold value, and on the other side of the surface, the data values are lower than the threshold value.

When in Surface mode, a logarithmic histogram plot of the data is displayed in the small draw window. Click and drag a mouse button on this plot to set the iso-surface threshold value. This value is also shown in the text widget below the plot. The threshold value may also be set by typing a new value in this text widget. The histogram plot is affected by the current threshold settings. (See Threshold mode, below).

Low

Selecting this mode will cause the iso-surface polygon facing to face towards the lower data values. Usually, this is the mode to use when the iso-surface is desired to surround high data values.

High

Selecting this mode will cause the iso-surface polygon facing to face towards the higher data values. Usually, this is the mode to use when the iso-surface is desired to surround low data values.

Shading pulldown menu

Iso-surfaces are normally rendered with light-source shading. If multiple datasets are currently loaded, then this menu allows the selection of a different 3-D array for the source of the iso-surface shading values. If only one dataset is currently loaded, then this menu is inactive.

Display Button

Clicking this button will cause the iso-surface to be created and drawn. Iso-surfaces often consist of tens of thousands of polygons, and can sometimes take considerable time to create and render.

Projection Mode

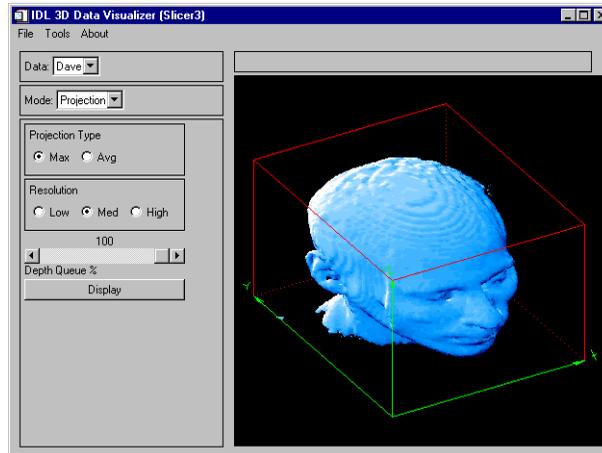


Figure 3-59: Projection Mode

A “voxel” projection of a 3-D array is the projection of the data values within that array onto a viewing plane. This is similar to taking an X-ray image of a 3-D object.

Max

Select this mode for a Maximum intensity projection.

Avg

Select this mode for an Average intensity projection.

Low

Select this mode for a Low resolution projection.

Med

Select this mode for a Medium resolution projection.

High

Select this mode for a High resolution projection.

Depth Queue % Slider

Use the slider to set the depth queue percent. A value of 50, for example, indicates that the farthest part of the projection will be 50% as bright as the closest part of the projection.

Display Button

Clicking this button will cause the projection to be calculated and drawn. Projections can sometimes take considerable time to display. Higher resolution projections take more computation time.

Threshold Mode

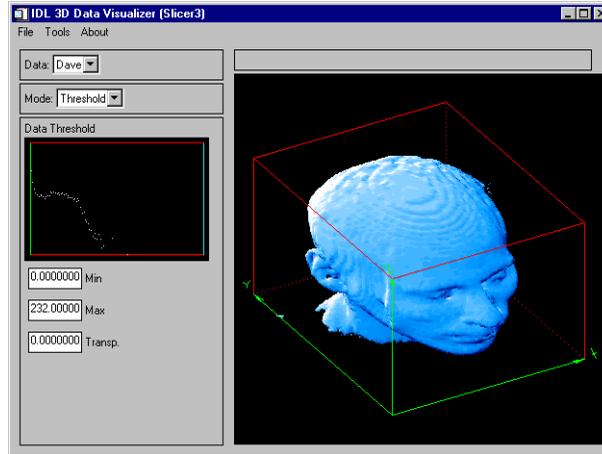


Figure 3-60: Threshold Mode

When in Threshold mode, a logarithmic histogram plot of the data is displayed in the small draw window. Click and drag the left mouse button on this plot to set the minimum and maximum threshold values. To expand a narrow range of data values into the full range of available colors, set the threshold range before displaying slices, blocks, or projections. The threshold settings also affect the histogram plot in “Surface” mode. The minimum and maximum threshold values are also shown in the text widgets below the histogram plot.

Click and drag the right mouse button on the histogram plot to set the transparency threshold. Portions of any slice, block, or projection that are less than the transparency value are not drawn (clear). Iso-surfaces are not affected by the transparency threshold. The transparency threshold value is also shown in a text widget below the histogram plot.

Min

In this text widget, a minimum threshold value can be entered.

Max

In this text widget, a maximum threshold value can be entered.

Transp.

In this text widget, a transparency threshold value can be entered.

Profile Mode

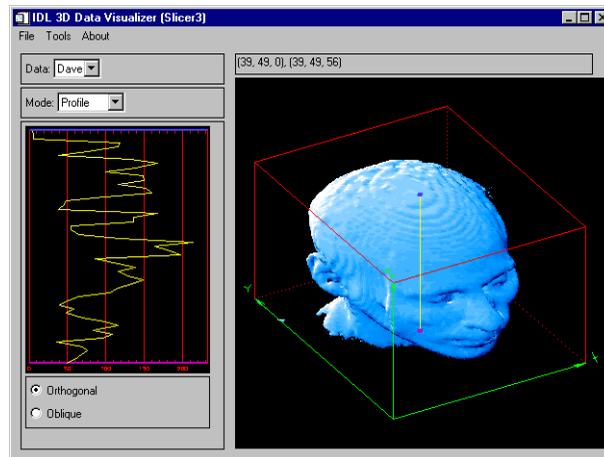


Figure 3-61: Profile Mode

In Profile mode, a plot is displayed showing the data values along a line. This line is also shown superimposed on the data in the main draw window. The bottom of the plot corresponds to the “purple” end of the line, and the top of the plot corresponds to the “blue” end of the line.

Orthogonal

Click and drag the left mouse button to position the profile line, based upon a point on the “front” faces of the wire-frame cube. Click and drag the right mouse button to position the profile line, based upon a point on the “back” faces of the wire-frame cube. As the profile line is moved, The profile plot is dynamically updated.

Oblique

Click and drag the left mouse button to position the “purple” end of the profile line on one of the “front” faces of the wire-frame cube. Click and drag the right mouse button to position the “blue” end of the profile line on one of the “back” faces of the wire-frame cube. As the profile line is moved, The profile plot is dynamically updated.

Probe Mode

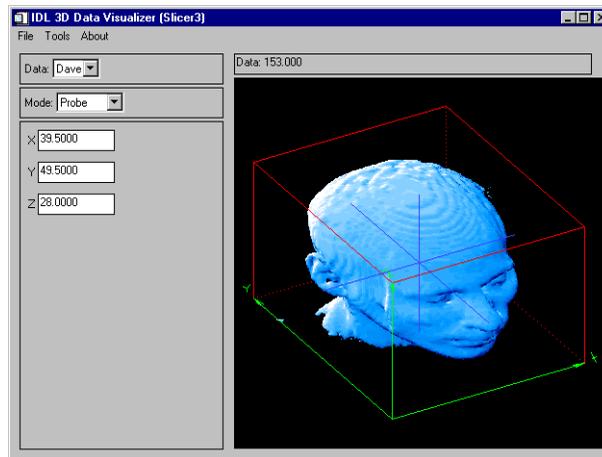


Figure 3-62: Probe Mode

In Probe mode, click and drag a mouse button over an object in the main draw window. The actual X-Y-Z location within the data volume is displayed in the three text widgets. Also, the data value at that 3-D location is displayed in the status window, above the main draw window. If the cursor is inside the wire-frame cube, but not on any object, then the status window displays “No data value”, and the three text widgets are empty. If the cursor is outside the wire-frame cube, then the status window and text widgets are empty.

X

Use this text widget to enter the X coordinate for the probe.

Y

Use this text widget to enter the Y coordinate for the probe.

Z

Use this text widget to enter the Z coordinate for the probe.

View Mode

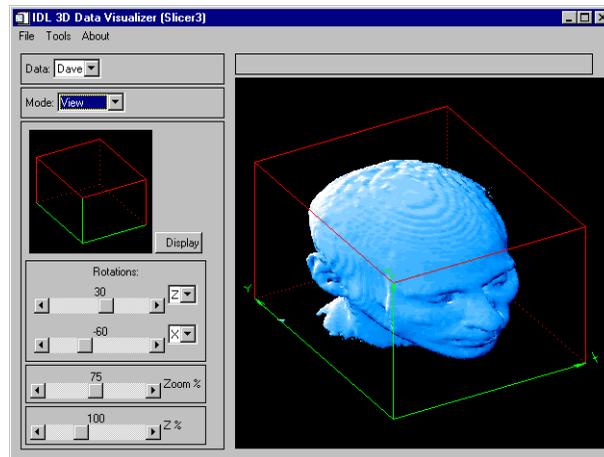


Figure 3-63: View Mode

In view mode, a small window shows the orientation of the data cube in the current view. As view parameters are changed, this window is dynamically updated. The main draw window is then updated when the user clicks on “Display”, or exits View mode.

Display

Clicking on this button will cause the objects in the main view window to be drawn in the new view. If any view parameters have been changed since the last time the main view was updated, the main view will be automatically redrawn when the user exits View mode.

1st Rotation

Use this slider to set the angle of the first view rotation (in degrees). The droplist widget adjacent to the slider indicates which axis this rotation is about.

2nd Rotation

Use this slider to set the angle of the second view rotation (in degrees). The droplist widget adjacent to the slider indicates which axis this rotation is about.

Zoom % Slider

Use this slider to set the zoom factor percent. Depending upon the view rotations, SLICER3 may override this setting to ensure that all eight corners of the data cube are within the window.

Z % Slider

Use this slider to set a scale factor for the Z axis (to compensate for the data's aspect ratio).

Operational Details

The SLICER3 procedure has the following side effects:

- SLICER3 sets the position for the light source and enables back-facing polygons to be drawn (see the IDL “SET_SHADING” command).
- SLICER3 overwrites the existing contents of the Z-buffer. Upon exiting SLICER3, the Z-buffer contents are the same as what was last displayed by SLICER3.
- On 24-bit displays, SLICER3 sets the device to non-decomposed color mode (DEVICE, DECOMPOSED=0).
- SLICER3 breaks the color table into 6 “bands”, based upon the number of available colors (where `max_color=!D.N_COLORS` on 8-bit displays, and `max_color=256` on 24-bit displays and `nColor = (max_color - 9) / 5`):

Band Start index	Band End index	Used For
0	$nColor-1$	X Slices.
$nColor$	$(2*nColor)-1$	Y Slices.
$2*nColor$	$(3*nColor)-1$	Z Slices.
$3*nColor$	$(4*nColor)-1$	Iso-surfaces
$4*nColor$	$(5*nColor)-1$	Projections

Table 3-108: SLICER3 Band Start/End

Annotation colors are the last “band”, and they are set up as shown in the table:

Color index	Color
max_color - 1	White
max_color - 2	Yellow
max_color - 3	Cyan
max_color - 4	Purple
max_color - 5	Red
max_color - 6	Green
max_color - 7	Blue
max_color - 8	Black

Table 3-109: SLICER3 Color Bands

On 24-bit displays, you can often improve performance by running SLICER3 in 8-bit mode. This can be accomplished (on some platforms) by entering the following command at the start of the IDL session (before any windows are created):

```
Device, Pseudo_Color=8
```

Examples

The following IDL commands open a data file from the IDL distribution and load it into SLICER3:

```
; Choose a data file:
file=FILEPATH('head.dat', SUBDIR=['examples', 'data'])

; Open the data file:
OPENR, UNIT, file, /GET_LUN

; Create an array to hold the data:
data = BYTARR(80, 100, 57, /NOZERO)

; Read the data into the array:
READU, UNIT, data

; Close the data file:
CLOSE, UNIT
```

```

; Create a pointer to the data array:
hData = PTR_NEW(data, /NO_COPY)

; Load the data into SLICER3:
SLICER3, hdata, DATA_NAMES='Dave'

```

Note

If data are loaded via the File menu after SLICER3 is launched with a pointer argument (as shown above), the pointer becomes invalid. You can use an IDL statement like the following to “clean up” after calling SLICER3 in this fashion:

```
if PTR_VALID(hdata) then PTR_FREE, hdata
```

Because we did not launch SLICER3 with the MODAL keyword, the last contents of the main draw window still reside in IDL’s Z-buffer. To retrieve this image after exiting SLICER3, use the following IDL statements:

```

; Save the current graphics device:
current_device = !D.Name

; Change to the Z-buffer device:
SET_PLOT, 'Z'

; Read the image from the Z-buffer:
image_buffer = TVRD()

; Return to the original graphics device:
SET_PLOT, current_device

; Display the image:
TV, image_buffer

```

The following IDL commands manually create a data save file suitable for dynamic loading into SLICER3. Note that if you load data into SLICER3 as shown above, you can also create save files by switching to BLOCK mode and using the Save Subset menu option.

```

; Store some 3-D data in a variable called data_1:
data_1 = INDGEN(20,30,40)

; Store some 3-D data in a variable called data_2:
data_2 = FINDGEN(20,30,40)

; Define the names for the datasets. Their names will appear in the
; "Data" pulldown menu in SLICER3:
data_1_name = 'Test Data 1'
data_2_name = 'Data 2'

```

```
; Select a data file name:
dataFile = DIALOG_PICKFILE()

; Write the file:
GET_LUN, lun
OPENW, lun, dataFile
WRITEU, lun, SIZE(data_1)
WRITEU, lun, STRLEN(data_1_name)
WRITEU, lun, BYTE(data_1_name)
WRITEU, lun, data_1
WRITEU, lun, SIZE(data_2)
WRITEU, lun, STRLEN(data_2_name)
WRITEU, lun, BYTE(data_2_name)
WRITEU, lun, data_2
CLOSE, lun
FREE_LUN, lun
```

Version History

Introduced: 5.0

See Also

[GRID3](#), [EXTRACT_SLICE](#), [IVOLUME](#), [SHADE_VOLUME](#), [XVOLUME](#)

SLIDE_IMAGE

The SLIDE_IMAGE procedure creates a scrolling graphics window for examining large images. By default, 2 draw widgets are used. The draw widget on the left shows a reduced version of the complete image, while the draw widget on the right displays the actual image with scrollbars that allow sliding the visible window.

This routine is written in the IDL language. Its source code can be found in the file `slide_image.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SLIDE_IMAGE [, Image] [, /BLOCK] [, CONGRID=0]
  [, FULL_WINDOW=variable] [, GROUP=widget_id] [, /ORDER]
  [, /REGISTER] [, RETAIN={0 | 1 | 2}] [, SLIDE_WINDOW=variable]
  [, SHOW_FULL=0] [, TITLE=string] [, TOP_ID=variable] [, XSIZE=width]
  [, XVISIBLE=width] [, YSIZE=height] [, YVISIBLE=height]
```

Arguments

Image

A 2-D image array to be displayed. If this argument is not specified, no image is displayed. The FULL_WINDOW and SCROLL_WINDOW keywords can be used to obtain the window numbers of the two draw widgets so they can be drawn into at a later time.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the NO_BLOCK keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have SLIDE_IMAGE block, any earlier calls to XMANAGER must have been called

with the `NO_BLOCK` keyword. See the documentation for the [NO_BLOCK](#) keyword to `XMANAGER` for an example.

CONGRID

Normally, the image is processed with the `CONGRID` procedure before it is written to the fully visible window on the left. Specifying `CONGIRD=0` will force the image to be drawn as is.

FULL_WINDOW

Set this keyword to a named variable that will contain the IDL window number of the fully visible window. This window number can be used with the `WSET` procedure to draw to the scrolling window at a later point.

GROUP

Set this keyword to the widget ID of the widget that calls `SLIDE_IMAGE`. If set, the death of the caller results in the death of `SLIDE_IMAGE`.

ORDER

This keyword is passed directly to the `TV` procedure to control the order in which the images are drawn. Usually, images are drawn from the bottom up. Set this keyword to a non-zero value to draw images from the top down.

REGISTER

Set this keyword to create a “Done” button for `SLIDE_IMAGE` and register the widgets with the `XMANAGER` procedure.

The basic widgets used in this procedure do not generate widget events, so it is not necessary to process events in an event loop. The default is therefore to simply create the widgets and return. Hence, when `REGISTER` is not set, `SLIDE_IMAGE` can be displayed and the user can still type commands at the IDL command prompt.

RETAIN

This keyword is passed directly to the `WIDGET_DRAW` function. `RETAIN` specifies how backing store should be handled for the window. Valid values are:

- 0 — Specifies no backing store. In this case, it is recommended that the `REGISTER` keyword be set so that expose and scroll events are handled.

- 1 — Requests that the server or window system provide backing store. This is the default if `RETAIN` is not specified.
- 2 — Specifies that IDL provide backing store directly.

See “[Backing Store](#)” on page 4286 for details.

SLIDE_WINDOW

Set this keyword to a named variable that will contain the IDL window number of the sliding window. This window number can be used with the `WSET` procedure to draw to the scrolling window at a later time.

SHOW_FULL

Set this keyword to zero to show the entire image at full resolution in one scrolling graphics window. By default, `SHOW_FULL` is set, displaying two draw widgets.

TITLE

Set this keyword to the title to be used for the `SLIDE_IMAGE` widget. If this keyword is not specified, “Slide Image” is used.

TOP_ID

Set this keyword to a named variable that will contain the top widget ID of the `SLIDE_IMAGE` hierarchy. This ID can be used to kill the hierarchy as shown below:

```
SLIDE_IMAGE, TOP_ID=base, ...
WIDGET_CONTROL, /DESTROY, base
```

XSIZE

Set this keyword to the maximum width of the image that can be displayed by the scrolling window. This keyword should not be confused with the visible size of the image, controlled by the `XVISIBLE` keyword. If `XSIZE` is not specified, the width of *Image* is used. If *Image* is not specified, 256 is used.

XVISIBLE

Set this keyword to the width of the viewport on the scrolling window. If this keyword is not specified, 256 is used.

YSIZE

Set this keyword to the maximum height of the image that can be displayed by the scrolling window. This keyword should not be confused with the visible size of the

image, controlled by the `YVISIBLE` keyword. If `YSIZE` is not present the height of *Image* is used. If *Image* is not specified, 256 is used.

YVISIBLE

Set this keyword to the height of the viewport on the scrolling window. If this keyword is not present, 256 is used.

Examples

Open an image from the IDL distribution and load it into `SLIDE_IMAGE`:

```
; Create a variable to hold the image:
image = BYTARR(768,512)

OPENR, unit, FILEPATH('nyny.dat', SUBDIR=['examples','data']),
/GET_LUN
READU, unit, image
CLOSE, unit

; Scale the image into byte range of the display:
image = BYTSCL(image)

; Display the image:
SLIDE_IMAGE, image
```

Version History

Introduced: Pre 4.0

See Also

[IIMAGE](#), [TV](#), [TVSCL](#), [WIDGET_DRAW](#), [WINDOW](#)

SMOOTH

The SMOOTH function returns a copy of *Array* smoothed with a boxcar average of the specified width. The result has the same type and dimensions as *Array*. The algorithm used by SMOOTH is:

$$R_i = \begin{cases} \frac{1}{w} \sum_{j=0}^{w-1} A_{i+j-w/2}, & i = \frac{(w-1)}{2}, \dots, N - \frac{(w+1)}{2} \\ A_i, & \text{otherwise} \end{cases}$$

where N is the number of elements in A .

Syntax

Result = SMOOTH(*Array*, *Width* [, /EDGE_TRUNCATE] [, MISSING=*value*] [, /NAN])

Return Value

Returns the smoothed array, which has the same dimensions as the input array.

Arguments

Array

The array to be smoothed. *Array* can have any number of dimensions.

Width

The width of the smoothing window. *Width* can either be a scalar or a vector with length equal to the number of dimensions of *Array*. If *Width* is a scalar then the same width is applied for each dimension that has length greater than 1 (dimensions of length 1 are skipped). If *Width* is a vector, then each element of *Width* is used to specify the smoothing width for each dimension of *Array*. Values for *Width* must be smaller than the corresponding *Array* dimension. If a *Width* value is even, then $Width+1$ will be used instead. The value of *Width* does not affect the running time of SMOOTH to a great extent.

Note

A Width value of zero or 1 implies no smoothing. However, if the NAN keyword is set, then any NaN values within the Array will be treated as missing data and will be replaced.

Tip

For a multi-dimensional array, set widths to 1 within the Width vector for dimensions that you don't want smoothed.

Keywords

EDGE_TRUNCATE

Set this keyword to apply the smoothing function to all points. If the neighborhood around a point includes a point outside the array, the nearest edge point is used to compute the smoothed result. If EDGE_TRUNCATE is not set, the end points are copied from the original array to the result with no smoothing.

For example, when smoothing an n -element vector with a three point wide smoothing window, the first point of the result R_0 is equal to A_0 if EDGE_TRUNCATE is not set, but is equal to $(A_0+A_0+A_1)/3$ if the keyword is set. In the same manner, point R_{n-1} is set to A_{n-1} if EDGE_TRUNCATE is not set, or to $(A_{n-2}+A_{n-1}+A_{n-1})/3$ if it is.

Note

Normally, two-dimensional floating-point arrays are smoothed in one pass. If the EDGE_TRUNCATE keyword is specified for a two-dimensional floating-point array, the result is obtained in two passes, first for all of the rows, and second for all of the columns. Therefore, the results for points in the interior of the array may differ slightly when the EDGE_TRUNCATE keyword is set. This difference will be most pronounced if the array contains NaN values.

MISSING

The value to return for elements that contain no valid points within the kernel. The default is the IEEE floating-point value NaN. This keyword is only used if the NAN keyword is set.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the

Building IDL Applications manual for more information on IEEE floating-point values.) In the *Result*, missing elements are replaced by the smoothed value of all other valid points within the smoothing window. If all points within the window are missing, then the result at that point is given by the MISSING keyword.

Note

SMOOTH should never be called without the NAN keyword if the input array may possibly contain *NaN* values.

Examples

Create and display a simple image by entering:

```
WINDOW, XSIZE=800, YSIZE=400
D = SIN(DIST(256)/3)
TVSCL, D
```

Now display the same dataset smoothed with a width of 9 in each dimension by entering:

```
TVSCL, SMOOTH(D, 9), 256, 0
```

Now smooth only in the vertical direction with a width of 15:

```
TVSCL, SMOOTH(D, [1, 15]), 512, 0
```

Version History

Introduced: Original

See Also

[DIGITAL_FILTER](#), [LEEFILT](#), [MEDIAN](#), [TS_DIFF](#), [TS_FCAST](#), [TS_SMOOTH](#)

SOBEL

The SOBEL function returns an approximation to the Sobel edge enhancement operator for images,

$$G_{jk} = |G_x| + |G_y|$$

$$G_Y = F_{j-1,k-1} + 2F_{j,k-1} + F_{j+1,k-1} - (F_{j-1,k+1} + 2F_{j,k+1} + F_{j+1,k+1})$$

$$G_X = F_{j+1,k+1} + 2F_{j+1,k} + F_{j+1,k-1} - (F_{j-1,k+1} + 2F_{j-1,k} + F_{j-1,k-1})$$

where (j, k) are the coordinates of each pixel F_{jk} in the *Image*. This is equivalent to a convolution using the masks,

$$\text{X mask} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \text{Y mask} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

All of the edge points in the result are set to zero.

Syntax

Result = SOBEL(*Image*)

Return Value

SOBEL returns a two-dimensional array of the same size as *Image*. If *Image* is of type byte or integer then the result is of integer type, otherwise the result is of the same type as *Image*.

Note

To avoid overflow for integer types, the computation is done using the next larger signed type and the result is transformed back to the correct type. Values larger than the maximum for that integer type are truncated. For example, for integers the function is computed using type long, and on output, values larger than 32767 are set equal to 32767.

Arguments

Image

The two-dimensional array containing the image to which edge enhancement is applied.

Keywords

None.

Examples

If the variable `myimage` contains a two-dimensional image array, a Sobel sharpened version of `myimage` can be displayed with the command:

```
TVSCL, SOBEL(myimage)
```

Version History

Introduced: Original

See Also

[ROBERTS](#)

SOCKET

The SOCKET procedure, supported on UNIX and Microsoft Windows platforms, opens a client-side TCP/IP Internet socket as an IDL file unit. Such files can be used in the standard manner with any of IDL's Input/Output routines.

Tip

RSI recommends that you don't use the EOF procedure as a way to check to see if a socket is empty. It is recommended that you structure your communication across the socket so that using EOF is not necessary to know when the communication is complete.

Syntax

```
SOCKET, Unit, Host, Port [, CONNECT_TIMEOUT=value] [, ERROR=variable]  
    [, GET_LUN] [, RAWIO] [, READ_TIMEOUT=value] [, SWAP_ENDIAN]  
    [, SWAP_IF_BIG_ENDIAN] [, SWAP_IF_LITTLE_ENDIAN]  
    [, WIDTH=value] [, WRITE_TIMEOUT=value]
```

UNIX-Only Keywords: [, **STDIO**]

Arguments

Unit

The unit number to associate with the opened socket.

Host

The name of the host to which the socket is connected. This can be either a standard Internet host name (e.g. `ftp.RSInc.com`) or a dot-separated numeric address (e.g. `192.5.156.21`).

Port

The port to which the socket is connected on the remote machine. If this is a well-known port (as contained in the `/etc/services` file on a UNIX host), then you can specify its name (e.g. `daytime`); otherwise, specify a number.

Keywords

CONNECT_TIMEOUT

Set this keyword to the number of seconds to wait before giving up and issuing an error to shorten the connect timeout from the system-supplied default. Most experts recommend that you not specify an explicit timeout, and instead use your operating system defaults.

Note

Although you can use `CONNECT_TIMEOUT` to shorten the timeout, you cannot increase it past the system-supplied default.

ERROR

A named variable in which to place the error status. If an error occurs in the attempt to open File, IDL normally takes the error handling action defined by the `ON_ERROR` and/or `ON_IOERROR` procedures. `SOCKET` always returns to the caller without generating an error message when `ERROR` is present. A nonzero error status indicates that an error occurred. The error message can then be found in the system variable `!ERROR_STATE.MSG`.

GET_LUN

Set this keyword to use the `GET_LUN` procedure to set the value of *Unit* before the file is opened. Instead of using the two statements:

```
GET_LUN, Unit
OPENR, Unit, 'data.dat'
```

you can use the single statement:

```
OPENR, Unit, 'data.dat', /GET LUN
```

RAWIO

Set this keyword to disable all use of the standard operating system I/O for the file, in favor of direct calls to the operating system. This allows direct access to devices, such as tape drives, that are difficult or impossible to use effectively through the standard I/O. Using this keyword has the following implications:

- No formatted or associated (`ASSOC`) I/O is allowed on the file. Only `READU` and `WRITEU` are allowed.

- Normally, attempting to read more data than is available from a file causes the unfilled space to be set to zero and an error to be issued. This does not happen with files opened with RAWIO. When using RAWIO, the programmer must check the transfer count, either via the TRANSFER_COUNT keywords to READU and WRITEU, or the FSTAT function.
- The EOF and POINT_LUN functions cannot be used with a file opened with RAWIO.
- Each call to READU or WRITEU maps directly to UNIX read(2) and write(2) system calls. The programmer must read the UNIX system documentation for these calls and documentation on the target device to determine if there are any special rules for I/O to that device. For example, the size of data that can be transferred to many cartridge tape drives is often forced to be a multiple of 512 bytes.

READ_TIMEOUT

Set this keyword to the number of seconds to wait for data to arrive before giving up and issuing an error. By default, IDL blocks indefinitely until the data arrives. Typically, this option is unnecessary on a local network, but it is useful with networks that are slow or unreliable.

STDIO

This keyword is only available on UNIX platforms.

Under UNIX, forces the file to be opened via the standard C I/O library (stdio) rather than any other more native OS API that might usually be used. This is primarily of interest to those who intend to access the file from external code, and is not necessary for most uses.

Note

Under Windows, the STDIO feature is not possible. Requesting it causes IDL to throw an error.

SWAP_ENDIAN

Set this keyword to swap byte ordering for multi-byte data when performing binary I/O on the specified file. This is useful when accessing files also used by another system with byte ordering different than that of the current host.

SWAP_IF_BIG_ENDIAN

Setting this keyword is equivalent to setting `SWAP_ENDIAN`; it only takes effect if the current system has big endian byte ordering. This keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

Setting this keyword is equivalent to setting `SWAP_ENDIAN`; it only takes effect if the current system has little endian byte ordering. This keyword does not refer to the byte ordering of the input data, but to the computer hardware.

WIDTH

The desired output width. When using the defaults for formatted output, IDL uses the following rules to determine where to break lines:

- If the output file is a terminal, the terminal width is used.
- Otherwise, a default of 80 columns is used.

The `WIDTH` keyword allows the user to override this default.

WRITE_TIMEOUT

Set this keyword to the number of seconds to wait to send data before giving up and issuing an error. By default, IDL blocks indefinitely until it is possible to send the data. Typically, this option is unnecessary on a local network, but it is useful with networks that are slow or unreliable.

Examples

Most UNIX systems maintain a daytime server on the daytime port (port 13). These servers send a 1 line response when connected to, containing the current time of day.

```
; To obtain the current time from the host bullwinkle:
SOCKET, 1, 'bullwinkle', 'daytime'
date= '
READF, 1, date
CLOSE, 1
PRINT, date
```

IDL prints:

```
Wed Sep 15 17:20:27 1999
```

Version History

Introduced: 5.3

SWAP_IF_BIG_ENDIAN, SWAP_IF_LITTLE_ENDIAN keywords added: 5.6

SORT

The SORT function returns a vector of subscripts that allow access to the elements of *Array* in ascending order.

Syntax

Result = SORT(*Array* [, /L64])

Return Value

The result is always a vector of integer type with the same number of elements as *Array*.

Note

If *Array* contains any identical elements, the order in which the identical elements are sorted is arbitrary and may vary between operating systems.

Arguments

Array

The array to be sorted. *Array* can be any basic type of vector or array. String arrays are sorted using the ASCII collating sequence. Complex arrays are sorted by their magnitude. Array values which are Not A Number (NaN) are moved to the end of the resulting array.

Keywords

L64

By default, the result of SORT is 32-bit integer when possible, and 64-bit integer if the number of elements being sorted requires it. Set L64 to force 64-bit integers to be returned in all cases.

Note

Only 64-bit versions of IDL are capable of creating variables requiring a 64-bit sort. Check the value of !VERSION.MEMORY_BITS to see if your IDL is 64-bit or not.

Examples

Example 1

```
A = [4, 3, 7, 1, 2]
PRINT, 'SORT(A) = ', SORT(A)

; Display the elements of A in sorted order:
PRINT, 'Elements of A in sorted order: ', A[SORT(A)]

; Display the elements of A in descending order:
PRINT, 'Elements of A in descending order: ', A[REVERSE(SORT(A))]
```

IDL prints:

```
SORT(A) = 3 4 1 0 2
Elements of A in sorted order: 1 2 3 4 7
Elements of A in descending order: 7 4 3 2 1
```

SORT(A) returns “3 4 1 0 2” because:

$$A[3] < A[4] < A[1] < A[0] < A[2]$$

Example 2

Sorting NaN Values

When sorting data including Not A Number (NaN) values, the NaN entries are moved to the end of the resulting array. For example:

```
values = [ 500, !VALUES.F_NAN, -500 ]
PRINT, SORT(values)
```

IDL prints:

```
2          0          1
```

Version History

Introduced: Original

See Also

[REVERSE](#), [UNIQ](#), [WHERE](#)

SPAWN

The SPAWN procedure spawns a child process to execute a command or series of commands. The result of calling SPAWN depends on the platform on which it is being used:

- Under UNIX, the shell used (if any) is obtained from the SHELL environment variable. The NOSHELL keyword can be used to execute a command directly as a child process without starting a shell process.
- Under Windows, a Command Shell is opened. The NOSHELL keyword can be used to execute the specified command directly without starting an intermediate command interpreter shell.

Note

See [“Using SPAWN Without a Shell Under UNIX”](#) on page 1977 for notes on executing commands without using a shell process under UNIX. See [“Execution Directory under Microsoft Windows”](#) on page 1978 for a caution regarding Windows network paths.

If SPAWN is called without arguments, an interactive command interpreter process is started, in which you can enter one or more operating system commands.

By default, IDL waits for the child process started by SPAWN to finish before it continues. It is possible to have IDL instead continue execution in parallel with the child process. The syntax for this depends on the operating system on your system:

- Under UNIX, include an ampersand (&) at the end of your shell command.
- Under Windows, specify the NOWAIT keyword to SPAWN.

Note

For more information on using SPAWN, see the *External Development Guide*.

Syntax

```
SPAWN [, Command [, Result] [, ErrResult] ]
```

Keywords (all platforms): [, [COUNT=variable](#)] [, [EXIT_STATUS=variable](#)]
 [, [/NOSHELL](#)] [, [/NULL_STDIN](#)] [, [PID=variable](#)] [, [/STDERR](#)]

UNIX-Only Keywords: [, [/NOTTYRESET](#)] [, [/SH](#)]
 [, [UNIT=variable](#) { *Command required, Result and ErrResult not allowed*}]

Windows-Only Keywords: [, /HIDE] [, /LOG_OUTPUT] [, /NOWAIT]

Arguments

Command

A string containing the commands to be executed.

If *Command* is present, it must be specified as follows:

- On UNIX, *Command* is expected to be scalar unless used in conjunction with the NOSHELL keyword, in which case *Command* is expected to be a string array where each element is passed to the child process as a separate argument.
- On Windows, *Command* can be a scalar string or string array. If it is a string array, SPAWN glues together each element of the string array, with each element separated by whitespace.

If *Command* is not present, SPAWN starts an interactive command interpreter process, which you can use to enter one or more operating system commands. While you use the command interpreter process, IDL is suspended. Under Windows, an interactive command shell window is created for this purpose. UNIX spawn does not create a separate window, but simply runs on the user's current tty, using the default shell (as specified by the SHELL environment variable). The SH keyword can be used to force use of the Bourne shell (/bin/sh).

Note

Using SPAWN in this manner is equivalent to using the IDL \$ command. The difference between these two is that \$ can only be used interactively while SPAWN can be used interactively or in IDL programs

Result

A named variable in which to place the output from the child process. Each line of output becomes a single array element. If *Result* is not present, the output from the child shell process goes to the standard output (usually the terminal).

Note

Result will be a null string if the NOWAIT keyword is specified (Windows) or if *Command* ends in an ampersand (&) (UNIX).

ErrResult

A named variable in which to place the error output (stderr) from the child process. Each line of output becomes a single array element. If *ErrResult* is not present, the error output from the child shell process goes to the standard error file.

Note

See the [STDERR](#) keyword for another error stream option.

Note

ErrResult will be a null string if the NOWAIT keyword is specified (Windows). If *Command* ends in an ampersand (&) (UNIX), *ErrResult* may contain an error string generated by the shell.

Keywords

COUNT

If *Result* is present and this keyword is also specified, COUNT specifies a named variable into which the number of lines of output is placed. This value gives the number of elements placed into *Result*.

EXIT_STATUS

Set this keyword to a named variable in which the exit status for the child process is returned. The meaning of this value is operating system dependent:

- Under UNIX, it is the value passed by the child to `exit(2)`, and is analogous to the value returned by `$?` under most UNIX shells. If the UNIT keyword is used, this keyword always returns 0. In this case, use the EXIT_STATUS keyword to FREE_LUN or CLOSE to determine the final exit status of the process.
- Under Windows, it is the value returned by the Win32 `GetExitCodeProcess()` system function. If the NOWAIT keyword is set, EXIT_STATUS returns 0.

HIDE

This keyword is only available on Windows platforms.

If HIDE is set, the command interpreter shell window is minimized to prevent the user from seeing it.

LOG_OUTPUT

This keyword is only available on Windows platforms.

Normally, IDL starts a command interpreter shell, and output from the child process is displayed in the command interpreter's window. If LOG_OUTPUT is set, the command interpreter window is minimized (as with HIDE) and all output is diverted to the IDLDE log window. If the *Result* or *ErrResult* arguments are present, they take precedence over LOG_OUTPUT.

NOSHELL

Set this keyword to specify that *Command* should execute directly as a child process without an intervening shell process.

- UNIX – When using the NOSHELL keyword under UNIX, you must specify *Command* as a string array in which the first element is the name of the command to execute and the following arguments are the arguments to be passed to the command. See [“Using SPAWN Without a Shell Under UNIX”](#) on page 1977 for notes on executing commands without using a shell process.
- Windows – Use this keyword to start the specified *Command* directly, without the use of an intervening command shell. This is useful for Windows programs that do not require a console, such as Notepad.

Many common DOS commands (e.g. DIR) are not distinct programs, and are instead implemented as part of the command interpreter. Specifying NOSHELL with such commands results in the command not being found. In such cases, the HIDE keyword might be useful.

NOTTYRESET

This keyword is only available on UNIX platforms.

Some UNIX systems drop characters when the tty mode is switched between normal and raw modes. IDL switches between these modes when reading command input and when using the GET_KBRD function. On such systems, IDL avoids losing characters by delaying the switch back to normal mode until it is truly needed. This method has the benefit of avoiding the large number of mode changes that would otherwise be necessary. Routines that cause output to be sent to the standard output (e.g., I/O operations, user interaction and SPAWN) ensure that the tty is in its normal mode before performing their operations.

If the NOTTYRESET keyword is set, SPAWN does not switch the tty back to normal mode before launching the child process assuming instead that the child will not send

output to the tty. Use this keyword to avoid characters being dropped in a loop of the form:

```
WHILE (GET_KBRD(0) NE 'q') SPAWN, command
```

This keyword has no effect on systems that don't suffer from dropped characters.

NOWAIT

This keyword is only available on Windows platforms.

If this keyword is set, the IDL process continues executing in parallel with the subprocess. Normally, the IDL process suspends execution until the subprocess completes.

NULL_STDIN

If set, the null device is connected to the standard input of the child process. The null device is either `/dev/null` (under UNIX) or `NUL` (under Windows).

PID

A named variable into which the Process Identification number of the child process is stored.

SH

This keyword is only available on UNIX platforms.

Set this keyword to force the use of the Bourne shell (`/bin/sh`). Usually, the shell used is determined by the `SHELL` environment variable.

STDERR

If set, the child's error output (`stderr`) is combined with the standard output and returned in *Result*. `STDERR` and the *ErrResult* argument are mutually exclusive. You should use one or the other, but not both.

UNIT

This keyword is only available on UNIX platforms.

If `UNIT` is present, `SPAWN` creates a child process in the usual manner, but instead of waiting for the specified command to finish, it attaches a bidirectional pipe between the child process and IDL. From the IDL session, the pipe appears as a logical file unit. The other end of the pipe is attached to the child process standard input and

output. The UNIT keyword specifies a named variable into which the number of the file unit is stored.

Once the child process is started, the IDL session can communicate with it through the usual input/output facilities. After the child process has done its task, the CLOSE procedure can be used to kill the process and close the pipe. Since SPAWN uses GET_LUN to allocate the file unit, FREE_LUN should be used to free the unit.

If UNIT is present, *Command* must be present, and neither *Result* or *ErrResult* are allowed.

Obsolete Keywords

The following keywords are obsolete:

- FORCE
- MACCREATOR
- NOCLISYM
- NOLOGNAM
- NOTIFY

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Using SPAWN Without a Shell Under UNIX

When a Unix program is run, its name and arguments are provided to it as an array of strings, one string per argument. The first string is the name of the program, and the remainder (if any) are the arguments. C programmers will recognize this as the standard (argc, argv) arguments passed to the main() function when the program is run. When you execute a command via a Unix shell, one of the operations that the shell carries out for you is to split the command and arguments apart on whitespace boundaries (blanks and tabs) to create this array of arguments. It then runs the program for you, using one of the Unix system exec() functions.

By default, SPAWN creates a shell process and passes the command to this shell instead of simply creating a child process to directly execute the command. Use of a shell is the default because the shell provides useful facilities such as wildcard expansion and argument processing (described above). Although this is usually desirable, it has the drawback of being slower than necessary, and of using an additional process for the shell.

When SPAWN is called with the NOSHELL keyword set, the command is executed as a direct child process, avoiding the extra overhead of starting a shell. This is faster,

but since there is no shell, you must specify the arguments in the standard form required by Unix programs. When you specify the NOSHELL keyword, the Command argument should be a string array. The first element of the array is the name of the command to use, and the following elements contain the arguments.

For example, consider the command,

```
SPAWN, 'ps ax'
```

that uses the UNIX *ps* command to show running processes on the computer. To issue this command without a shell, you would write it as follows:

```
SPAWN, ['ps', 'ax'], /NOSHELL
```

Execution Directory under Microsoft Windows

SPAWN attempts to use IDL's current working directory as the current directory for the spawned process. However, Microsoft Windows does not support the specification of a UNC path as the current directory for a Command Shell. Issuing a SPAWN command when IDL's current working directory is set to a UNC path will cause Windows to generate an error that looks something like:

```
CMD.EXE was started with '\\host\dir' as the current directory
path. UNC paths are not supported. Defaulting to Windows
directory.
```

If your application requires that you be able to use SPAWN when IDL's current working directory is set to a directory on a Windows network, consider mapping the UNC path to a Windows drive letter and setting that to be IDL's working directory.

Examples

Example 1: Interactive use of SPAWN

To simply spawn a shell process from within IDL, enter the command:

```
SPAWN
```

To execute the UNIX *ls* command and return to the IDL prompt, enter:

```
SPAWN, 'ls'
```

To execute the UNIX *ls* command and store the result in the IDL string variable *listing*, enter:

```
SPAWN, 'ls', listing
```

Example 2: Noninteractive use of SPAWN

It is sometimes useful to create a temporary scratch file, removing the file when it is no longer needed. SPAWN could be used as shown below to manage the removal of the scratch file.

```

OPENW, UNIT, 'scratch.dat', /GET_LUN

;...IDL commands go here.

;Deallocate the file unit and close the file.
FREE_LUN, UNIT

;Use the !VERSION system variable to determine the proper file
;deletion command for the current operating system.
CASE !VERSION.OS OF
  'Windows': CMD = 'DEL'
  ELSE: CMD = 'rm'
ENDCASE

;Delete the file using SPAWN.
SPAWN, CMD + ' scratch.dat'

END

```

Note

The DELETE keyword to the [OPEN](#) procedure or the [FILE_DELETE](#) procedure more efficiently handles this job. The above example should serve only to demonstrate use of the SPAWN procedure.

Version History

Introduced: Original

See Also

“Dollar Sign (\$)” on page 4407, [Chapter 2](#), “Using SPAWN and UNIX Pipes” in the *External Development Guide* manual

SPH_4PNT

Given four 3-dimensional points, the SPH_4PNT procedure returns the center and radius necessary to define the unique sphere passing through those points.

This routine is written in the IDL language. Its source code can be found in the file `sph_4pnt.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SPH_4PNT, X, Y, Z, Xc, Yc, Zc, R [, /DOUBLE]
```

Arguments

X, Y, Z

4-element floating-point or double-precision vectors containing the X, Y, and Z coordinates of the points.

Xc, Yc, Zc

Named variables that will contain the sphere's center X, Y, and Z coordinates.

R

A named variable that will contain the sphere's radius.

Keywords

DOUBLE

Set this keyword to force computations to be done in double-precision arithmetic.

Examples

Find the center and radius of the unique sphere passing through the points: (1, 1, 0), (2, 1, 2), (1, 0, 3), (1, 0, 1):

```
    ; Define the floating-point vectors containing the x, y and z
    ; coordinates of the points:
    X = [1, 2, 1, 1] + 0.0
    Y = [1, 1, 0, 0] + 0.0
    Z = [0, 2, 3, 1] + 0.0
```

```
    ; Compute sphere's center and radius:  
    SPH_4PNT, X, Y, Z, Xc, Yc, Zc, R  
  
    ; Print the results:  
    PRINT, Xc, Yc, Zc, R
```

IDL prints:

```
    -0.500000      2.00000      2.00000      2.69258
```

Version History

Introduced: Pre 4.0

See Also

[CIR_3PNT](#), [PNT_LINE](#)

SPH_SCAT

The SPH_SCAT function performs spherical gridding. Scattered samples on the surface of a sphere are interpolated to a regular grid. This routine is a convenient interface to the spherical gridding and interpolation provided by TRIANGULATE and TRIGRID.

This routine is written in the IDL language. Its source code can be found in the file `sph_scat.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = SPH_SCAT( Lon, Lat, F [, BOUNDS=[lonmin, latmin, lonmax, latmax]]
  [, BOUT=variable] [, GOUT=variable] [, GS=[lonspacing, latspacing]]
  [, NLON=value] [, NLAT=value] )
```

Return Value

Returns a regularly-interpolated grid of the function.

Arguments

Lon

A vector of sample longitudes, in degrees.

Note

Lon, *Lat*, and *F* must all have the same number of points.

Lat

A vector of sample latitudes, in degrees.

F

A vector of data values which are functions of Lon and Lat. F_i represents a value at (Lon_i, Lat_i) .

Keywords

BOUNDS

Set this keyword to a four-element vector containing the grid limits in longitude and latitude of the output grid. The four elements are: [Lon_{min} Lat_{min} Lon_{max} Lat_{max}]. If this keyword is not set, the grid limits are set to the extent of Lon and Lat .

Note

To cover all longitudes, you must explicitly specify the values for the BOUNDS keyword.

BOUT

Set this keyword to a named variable that, on return, contains a four-element vector (similar to BOUNDS) that describes the actual extent of the regular grid.

GOUT

Set this keyword to a named variable that, on return, contains a two-element vector (similar to GS) that describes the actual grid spacing.

GS

Set this keyword to a two-element vector that specifies the spacing between grid points in longitude (the first element) and latitude (the second element).

If this keyword is not set, the default value is based on the extents of Lon and Lat . The default longitude spacing is $(Lon_{max} - Lon_{min})/(NLON-1)$. The default latitude spacing is $(Lat_{max} - Lat_{min})/(NLAT-1)$. If NLON and NLAT are not set, the default grid size of 26 by 26 is used for NLON and NLAT.

NLON

The output grid size in the longitude direction. The default value is 26.

Note

NLON need not be specified if the size can be inferred from GS and BOUNDS.

NLAT

The output grid size in the latitude direction. The default value is 26.

Note

NLAT need not be specified if the size can be inferred from GS and BOUNDS.

Examples

```
; Create some random longitude points:
lon = RANDOMU(seed, 50) * 360. -180.

; Create some random latitude points:
lat = RANDOMU(seed, 50) * 180. -90.

; Make a function to fit:
z = SIN(lat*!DTOR)
c = COS(lat*!DTOR)
x = COS(lon*!DTOR) * c
y = SIN(lon*!DTOR) * c

; The finished dependent variable:
f = SIN(x+y) * SIN(x*z)
; Interpolate the data and return the result in variable r:
r = SPH_SCAT(lon, lat, f, BOUNDS=[0, -90, 350, 85], GS=[10,5])
```

Version History

Introduced: 4.0

See Also

[TRIANGULATE](#), [TRIGRID](#)

SPHER_HARM

The SPHER_HARM function returns the value of the spherical harmonic $Y_{lm}(\theta, \phi)$, $-l \leq m \leq l$, $l \geq 0$, which is a function of two coordinates on a spherical surface.

The spherical harmonics are related to the associated Legendre polynomial by:

$$Y_{lm}(\theta, \phi) = \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_l^m(\cos \theta) e^{im\phi}$$

For negative m the following relation is used:

$$Y_{l,-m}(\theta, \phi) = (-1)^m Y_{lm}^*(\theta, \phi)$$

where * represents the complex conjugate.

This routine is written in the IDL language. Its source code can be found in the file `spher_harm.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = SPHER_HARM(*Theta*, *Phi*, *L*, *M*, [, /DOUBLE])

Return Value

SPHER_HARM returns a complex scalar or array containing the value of the spherical harmonic function. The return value has the same dimensions as the input arguments *Theta* and *Phi*. If one argument (*Theta* or *Phi*) is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

If either *Theta* or *Phi* are double-precision or if the DOUBLE keyword is set, the result is double-precision complex, otherwise the result is single-precision complex.

Arguments

Theta

The value of the polar (colatitudinal) coordinate θ at which $Y_{lm}(\theta, \phi)$ is evaluated. *Theta* can be either a scalar or an array.

Phi

The value of the azimuthal (longitudinal) coordinate ϕ at which $Y_{lm}(\theta, \phi)$ is evaluated. *Phi* can be either a scalar or an array.

L

A scalar integer, $L \geq 0$, specifying the order l of $Y_{lm}(\theta, \phi)$. If L is of type float, it will be truncated.

M

A scalar integer, $-L \leq M \leq L$, specifying the azimuthal order m of $Y_{lm}(\theta, \phi)$. If M is of type float, it will be truncated.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

This example visualizes the electron probability density for the hydrogen atom in state 3d0. (Feynman, Leighton, and Sands, 1965: The Feynman Lectures on Physics, Calif. Inst. Tech, Ch. 19):

```

; Define a data cube (N x N x N)
n = 41L
a = 60*FINDGEN(n)/(n-1) - 29.999 ; [-1,+1]
x = REBIN(a, n, n, n) ; X-coordinates of cube
y = REBIN(REFORM(a,1,n), n, n, n) ; Y-coordinates
z = REBIN(REFORM(a,1,1,n), n, n, n); Z-coordinates

; Convert from rectangular (x,y,z) to spherical (phi, theta, r)
spherCoord = CV_COORD(FROM_RECT= $
  TRANSPOSE([[x[*]], [y[*]], [z[*]]]), /TO_SPHERE)
phi = REFORM(spherCoord[0,*], n, n, n)
theta = REFORM(!PI/2 - spherCoord[1,*], n, n, n)
r = REFORM(spherCoord[2,*], n, n, n)

; Find electron probability density for hydrogen atom in state 3d0
; Angular component
L = 2 ; state "d" is electron spin L=2
M = 0 ; Z-component of spin is zero
angularState = SPHER_HARM(theta, phi, L, M)

```

```
; Radial component for state n=3, L=2
radialFunction = EXP(-r/2)*(r^2)
waveFunction = angularState*radialFunction
probabilityDensity = ABS(waveFunction)^2

SHADE_VOLUME, probabilityDensity, $
  0.1*MEAN(probabilityDensity), vertex, poly
oPolygon = OBJ_NEW('IDLgrPolygon', vertex, $
  POLYGON=poly, COLOR=[180,180,180])
XOBJVIEW, oPolygon
```

The results are shown in the following figure (rotated in XOBJVIEW for clarity):

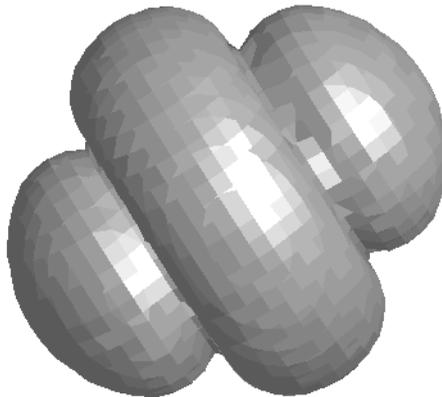


Figure 3-64: SPHER_HARM Example of Hydrogen Atom (object rotated in XOBJVIEW for clarity)

Version History

Introduced: 5.4

See Also

[LEGENDRE](#), [LAGUERRE](#)

SPL_INIT

The SPL_INIT function is called to establish the type of interpolating spline for a tabulated set of functional values X_i , $Y_i = F(X_i)$.

It is important to realize that SPL_INIT should be called only *once* to process an entire tabulated function in arrays X and Y . Once this has been done, values of the interpolated function for any value of X can be obtained by calls (as many as desired) to the separate function SPL_INTERP.

SPL_INIT is based on the routine spline described in section 3.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = SPL_INIT( X, Y [, /DOUBLE] [, YP0=value] [, YPN_1=value] )
```

Return Value

Returns the values of the 2nd derivative of the interpolating function at the points X_i .

Arguments

X

An n -element input vector that specifies the tabulate points in ascending order.

Y

An n -element input vector that specifies the values of the tabulated function $F(X_i)$ corresponding to X_i .

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

YP0

The first derivative of the interpolating function at the point X_0 . If YP0 is omitted, the second derivative at the boundary is set to zero, resulting in a “natural spline.”

YPN_1

The first derivative of the interpolating function at the point X_{n-1} . If YPN_1 is omitted, the second derivative at the boundary is set to zero, resulting in a “natural spline.”

Examples

Example 1

```
X = (FINDGEN(21)/20.) * 2.0*!PI
Y = SIN(X)
PRINT, SPL_INIT(X, Y, YP0 = -1.1, YPN_1 = 0.0)
```

IDL Prints:

```
23.1552    -6.51599    1.06983    -1.26115    -0.839544    -1.04023
-0.950336  -0.817987   -0.592022  -0.311726   2.31192e-05  0.311634
 0.592347   0.816783   0.954825   1.02348    0.902068    1.02781
-0.198994   3.26597   -11.0260
```

Example 2

```
PRINT, SPL_INIT(X, Y, YP0 = -1.1)
```

IDL prints:

```
23.1552    -6.51599    1.06983    -1.26115    -0.839544    -1.04023
-0.950336  -0.817988   -0.592020  -0.311732   4.41521e-05  0.311555
 0.592640   0.815690   0.958905   1.00825    0.958905    0.815692
 0.592635   0.311567   0.00000
```

Version History

Introduced: 4.0

See Also

[SPL_INTERP](#), [SPLINE](#), [SPLINE_P](#)

SPL_INTERP

Given the arrays X and Y , which tabulate a function (with the X_i in ascending order), and given the array Y_2 , which is the output from SPL_INIT, and given an input value of X_2 , the SPL_INTERP function returns a cubic-spline interpolated value for the given value of X_2 .

SPL_INTERP is based on the routine `splint` described in section 3.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = SPL_INTERP( X, Y, Y2, X2 [, /DOUBLE] )
```

Return Value

Returns either single- or double-precision floating result of the same structure as X_2 .

Arguments

X

An input array that specifies the tabulated points in ascending order.

Y

An input array that specifies the values of the tabulate function corresponding to X_i .

Y2

The output from SPL_INIT for the specified X and Y .

X2

The input value for which an interpolated value is desired. X can be scalar or an array of values. The result of SPL_INTERP will have the same structure.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To create a spline interpolation over a tabulated set of data, $[X_i, Y_i]$, first create the tabulated data. In this example, X_i will be in the range $[0.0, 2\pi]$ and Y_i in the range $[\sin(0.0), \sin(2\pi)]$.

```
X = (FINDGEN(21)/20.0) * 2.0 * !PI
Y = SIN(X)

; Calculate interpolating cubic spline:
Y2 = SPL_INIT(X, Y)

; Define the X values P at which we desire interpolated Y values:
X2= FINDGEN(11)/11.0 * !PI

; Calculate the interpolated Y values corresponding to X2[i]:
result = SPL_INTERP(X, Y, Y2, X2)

PRINT, result
```

IDL prints:

```
0.00000 0.281733 0.540638 0.755739 0.909613 0.989796
0.989796 0.909613 0.755739 0.540638 0.281733
```

The exact solution vector is $\sin(X_2)$.

To interpolate a line in the XY plane, see [SPLINE_P](#).

Version History

Introduced: 4.0

See Also

[SPL_INIT](#), [SPLINE](#), [SPLINE_P](#)

SPLINE

The `SPLINE` function performs cubic spline interpolation.

This routine is written in the IDL language. Its source code can be found in the file `spline.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = `SPLINE`(*X*, *Y*, *T* [, *Sigma*] [, `/DOUBLE`])

Return Value

Returns the result of the cubic spline interpolation. If *X*, *Y*, or *T* are of type double-precision, then the computations will be done using double-precision and a double-precision result will be returned. Otherwise, single-precision will be used.

Arguments

X

The abscissa vector. Values *must* be monotonically increasing. *X* must have three or more points.

Y

The vector of ordinate values corresponding to *X*. *Y* should have the same number of elements as *X*.

T

The vector of abscissa values for which the ordinate is desired. The values of *T* *must* be monotonically increasing.

Sigma

The amount of “tension” that is applied to the curve. The default value is 1.0. If sigma is close to 0, (e.g., .01), then effectively there is a cubic spline fit. If sigma is large, (e.g., greater than 10), then the fit will be like a polynomial interpolation.

Keywords

DOUBLE

Set this keyword to perform computations using double-precision arithmetic and to return a double-precision result.

Examples

The commands below show a typical use of SPLINE:

```
; X values of original function:  
X = [2.,3.,4.]  
  
; Make a quadratic  
Y = (X-3)^2  
;Values for interpolated points:  
T = FINDGEN(20)/10.+2  
  
; Do the interpolation:  
Z = SPLINE(X,Y,T)
```

Version History

Introduced: Original

DOUBLE keyword added: 6.1

See Also

[SPL_INIT](#), [SPLINE_P](#)

SPLINE_P

The `SPLINE_P` procedure performs parametric cubic spline interpolation with relaxed or clamped end conditions.

This routine is both more general and faster than the `SPLINE` function. One call to `SPLINE_P` is equivalent to two calls to `SPLINE`, as both the `X` and `Y` are interpolated with splines. It is suited for interpolating between randomly placed points, and the abscissa values need not be monotonic. In addition, the end conditions may be optionally specified via tangents.

This routine is written in the IDL language. Its source code can be found in the file `spline_p.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SPLINE_P, X, Y, Xr, Yr [, /DOUBLE] [, INTERVAL=value] [, TANO=[X0, Y0]]
      [, TANI=[Xn-1, Yn-1]]
```

Arguments

X

The abscissa vector. `X` should be floating-point or double-precision.

Y

The vector of ordinate values corresponding to `X`. `Y` should be floating-point or double-precision.

Neither `X` or `Y` need be monotonic.

Xr

A named variable that will contain the abscissa values of the interpolated function. If `X` or `Y` is double-precision, then the computations will be done using double-precision and a double-precision result will be returned. Otherwise, single-precision will be used.

Yr

A named variable that will contain the ordinate values of the interpolated function. If `X` or `Y` is double-precision, then the computations will be done using double-

precision and a double-precision result will be returned. Otherwise, single-precision will be used.

Keywords

DOUBLE

Set this keyword to perform computations using double-precision arithmetic and to return a double-precision result.

INTERVAL

Set this keyword equal to the desired interval in XY space between interpolants. If omitted, approximately 8 interpolants per XY segment will result.

TAN0

The tangent to the spline curve at $X[0]$, $Y[0]$. If omitted, the tangent is calculated to make the curvature of the result zero at the beginning. TAN0 is a two element vector, containing the X and Y components of the tangent.

TAN1

The tangent to the spline curve at $X[n-1]$, $Y[n-1]$. If omitted, the tangent is calculated to make the curvature of the result zero at the end. TAN1 is a two element vector, containing the X and Y components of the tangent.

Examples

The commands below show a typical use of `SPLINE_P`:

```
; Abscissas for square with a vertical diagonal:
X = [0.,1,0,-1,0]

; Ordinates:
Y = [0.,1,2,1,0]

; Interpolate with relaxed end conditions:
SPLINE_P, X, Y, XR, YR

; Show it:
PLOT, XR, YR
```

As above, but with setting both the beginning and end tangents:

```
SPLINE_P, X, Y, XR, YR, TAN0=[1,0], TAN1=[1,0]
```

This yields approximately 32 interpolants.

As above, but with setting the interval to 0.05, making more interpolants, closer together:

```
SPLINE_P, X, Y, XR, YR, TAN0=[1,0], TAN1=[1,0], INTERVAL=0.05
```

This yields 116 interpolants and looks close to a circle.

Version History

Introduced: Pre 4.0

DOUBLE keyword added: 6.1

See Also

[SPL_INIT](#), [SPLINE](#)

SPRSAB

The SPRSAB function performs matrix multiplication on two row-indexed sparse arrays created by SPRSIN. The routine computes all components of the matrix products, but only stores those values whose absolute magnitude exceeds the threshold value.

SPRSAB is based on the routine `sprstm` described in section 2.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission. The difference between the two routines is that SPRSAB performs the matrix multiplication $A*B$ rather than $A*B^T$.

Syntax

Result = SPRSAB(*A*, *B* [, /DOUBLE] [, THRESHOLD=*value*])

Return Value

The result is a row-indexed sparse array.

Arguments

A, B

Row-indexed sparse arrays created by the SPRSIN function.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

THRESHOLD

Use this keyword to set the criterion for deciding the absolute magnitude of the elements to be retained in sparse storage mode. For single-precision calculations, the default value is 1.0×10^{-7} . For double-precision calculations, the default is 1.0×10^{-14} .

Examples

```

; Begin by creating two arrays:
A = [[ 5.0,  0.0, 0.0,  1.0], $
      [ 3.0, -2.0, 0.0,  1.0], $
      [ 4.0, -1.0, 0.0,  2.0], $
      [ 0.0,  3.0, 3.0,  1.0]]
B = [[ 1.0,  2.0, 3.0,  1.0], $
      [ 3.0, -3.0, 0.0,  1.0], $
      [-1.0,  3.0, 1.0,  2.0], $
      [ 0.0,  3.0, 3.0,  1.0]]

; Convert the arrays to sparse array format before multiplying.
; The variable SPARSE holds the result in sparse array form:
sparse = SPRSAB(SPRSIN(A), SPRSIN(B))

; Restore the sparse array structure to full storage mode:
result = FULSTR(sparse)

; Print the result:
PRINT, 'result:'
PRINT, result

; Check this result by multiplying the original arrays:
exact = B # A
PRINT, 'exact:'
PRINT, exact

```

IDL prints:

```

result:
  5.00000    13.0000    18.0000     6.00000
 -3.00000    15.0000    12.0000     2.00000
  1.00000    17.0000    18.0000     5.00000
  6.00000     3.00000     6.00000    10.0000
exact:
  5.00000    13.0000    18.0000     6.00000
 -3.00000    15.0000    12.0000     2.00000
  1.00000    17.0000    18.0000     5.00000
  6.00000     3.00000     6.00000    10.0000

```

Version History

Introduced: 4.0

See Also

[FULSTR](#), [LINBCG](#), [SPRSAX](#), [SPRSIN](#), [SPRSTP](#), [READ_SPR](#), [WRITE_SPR](#)

SPRSAX

The SPRSAX function takes a row-indexed sparse array created by the SPRSIN function and multiplies it by an n -element vector to its right.

SPRSAX is based on the routine `spr sax` described in section 2.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

Result = SPRSAX(*A*, *X* [, /DOUBLE])

Return Value

Returns a n -element vector.

Arguments

A

A row-indexed sparse array created by the SPRSIN function.

X

An n -element right hand vector.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```
; Begin by creating an array A:
A = [[ 5.0,  0.0, 0.0], $
      [ 3.0, -2.0, 0.0], $
      [ 4.0, -1.0, 0.0]]

; Define the right-hand vector:
X = [1.0, 2.0, -1.0]
```

```
    ; Convert to sparse format, then multiply by X:  
    result = SPRSAX(SPRSIN(A),X)
```

```
    ; Print the result:  
    PRINT, result
```

IDL prints:

```
    5.00000    -1.00000    2.00000
```

Version History

Introduced: 4.0

See Also

[FULSTR](#), [LINBCG](#), [SPRSAB](#), [SPRSIN](#), [SPRSTP](#), [READ_SPR](#), [WRITE_SPR](#)

SPRSIN

The SPRSIN function converts an array, or list of subscripts and values, into a row-index sparse storage mode, retaining only elements with an absolute magnitude greater than or equal to the specified threshold. The list form is much more efficient than the array form if the density of the matrix is low.

SPRSIN is based on the routine `sprsin` described in section 2.7 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = SPRSIN( A [, /COLUMN] [, /DOUBLE] [, THRESHOLD=value] )
```

or

```
Result = SPRSIN(Columns, Rows, Values, N [, /DOUBLE] [, THRESHOLD=value])
```

Return Value

The result is a row-indexed sparse array contained in structure form. The structure consists of two linear sparse storage vectors: SA, a vector of array values, and IJA, a vector of subscripts to the SA vector. The length of these vectors is equal to 1 plus the number of diagonal elements of the array, plus the number of off-diagonal elements with an absolute magnitude greater than or equal to the threshold value. Diagonal elements of the array are always retained even if their absolute magnitude is less than the specified threshold.

Arguments

A

An n by n array of any type except string or complex.

Columns

A vector containing the column subscripts of the non-zero elements. Values must be in the range of 0 to (N-1).

Rows

A vector, of the same length as Column, containing the row subscripts of the non-zero elements. Values must be in the range of 0 to (N-1).

Values

A vector, of the same length as Column, containing the values of the non-zero elements.

N

The size of the resulting sparse matrix.

Keywords

COLUMN

Set this keyword if the input array *A* is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors). This keyword is not allowed in the list form of the call.

DOUBLE

Set this keyword to convert the sparse array to double-precision.

THRESHOLD

Use this keyword to set the criterion for deciding the absolute magnitude of the elements to be retained in sparse storage mode. For single-precision calculations, the default value is 1.0×10^{-7} . For double-precision values, the default is 1.0×10^{-14} .

Examples

Example1

Suppose we wish to convert the following array to sparse storage format:

```
A = [[ 5.0, -0.2, 0.1], $
      [ 3.0, -2.0, 0.3], $
      [ 4.0, -1.0, 0.0]]
```

```
; Convert to sparse storage mode. All elements of the array A that
; have absolute values less than THRESH are set to zero.
sparse = SPRSIN(A, THRESH = 0.5)
```

The variable SPARSE now contains a representation of A in structure form. See the description of FULSTR for an example that restores such a structure to full storage mode.

Example2

This example demonstrates how to use the list form of the call to SPRSIN. The following line of code creates a sparse matrix, equivalent to a 100 by 100 identity matrix, i.e. all diagonal elements are set to 1, all other elements are zero:

```
I100 = SPRSIN(LINDGEN(100), LINDGEN(100), REPLICATE(1.0,100), 100)
```

Version History

Introduced: 4.0

See Also

[FULSTR](#), [LINBCG](#), [SPRSAB](#), [SPRSAX](#), [SPRSTP](#), [READ_SPR](#), [WRITE_SPR](#)

SPRSTP

The SPRSTP function constructs the transpose of a sparse matrix.

Syntax

Result = SPRSTP(*A*)

Return Value

Returns the sparse matrix of the given sparse array.

Arguments

A

A row-indexed sparse array created by the SPRSIN function.

Keywords

None

Examples

This example creates a 100 by 100 pseudo-random sparse matrix, with 1000 non-zero elements, and then computes the product of the matrix and its transpose:

```
n = 100                                ;Dimensions of matrix
m = 1000                               ;Number of non-zero elements
a = SPRSIN(RANDOMU(seed, m)*n, RANDOMU(seed, m)*n, $
           RANDOMU(seed, m),n)
b = SPRSAB(a, SPRSTP(a)) ;Transpose and create the product
```

Version History

Introduced: 4.0

See Also

[FULSTR](#), [LINBCG](#), [SPRSAB](#), [SPRSAX](#), [SPRSIN](#), [READ_SPR](#), [WRITE_SPR](#)

SQRT

The SQRT function computes the square root of X .

Syntax

$Result = \text{SQRT}(X)$

Return Value

Returns the square root of X .

Arguments

X

The value for which the square root is desired. If X is double-precision floating-point or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. When applied to complex numbers, $z = x+iy$:

$$z^{1/2} = \left[\frac{1}{2}(r + x) \right]^{1/2} \pm i \left[\frac{1}{2}(r - x) \right]^{1/2}$$

$$r = \sqrt{x^2 + y^2}$$

The ambiguous sign is taken to be the same as the sign of y . The result has the same structure as X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To find the square root of 145 and store the result in variable S, enter:

```
S = SQRT(145)
```

Version History

Introduced: Original

See Also

[“Exponentiation”](#) in Chapter 2 of the *Building IDL Applications* manual.

STANDARDIZE

The STANDARDIZE function computes standardized variables from an array of m variables (columns) and n observations (rows).

This routine is written in the IDL language. Its source code can be found in the file `standardize.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = STANDARDIZE( A [, /DOUBLE] )
```

Return Value

The result is an m -column, n -row array where all columns have a mean of zero and a variance of one.

Arguments

A

An m -column, n -row single- or double-precision floating-point array.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```
; Define an array with 4 variables and 20 observations:
array = $
      [[19.5, 43.1, 29.1, 11.9], $
       [24.7, 49.8, 28.2, 22.8], $
       [30.7, 51.9, 37.0, 18.7], $
       [29.8, 54.3, 31.1, 20.1], $
       [19.1, 42.2, 30.9, 12.9], $
       [25.6, 53.9, 23.7, 21.7], $
       [31.4, 58.5, 27.6, 27.1], $
       [27.9, 52.1, 30.6, 25.4], $
       [22.1, 49.9, 23.2, 21.3], $
       [25.5, 53.5, 24.8, 19.3], $
```

```

[31.1, 56.6, 30.0, 25.4], $
[30.4, 56.7, 28.3, 27.2], $
[18.7, 46.5, 23.0, 11.7], $
[19.7, 44.2, 28.6, 17.8], $
[14.6, 42.7, 21.3, 12.8], $
[29.5, 54.4, 30.1, 23.9], $
[27.7, 55.3, 25.7, 22.6], $
[30.2, 58.6, 24.6, 25.4], $
[22.7, 48.2, 27.1, 14.8], $
[25.2, 51.0, 27.5, 21.1]]

```

```

; Compute the mean and variance of each variable using the MOMENT
; function. The skewness and kurtosis are also computed:
FOR K = 0, 3 DO PRINT, MOMENT(array[K,*])

```

```

; Compute the standardized variables:
result = STANDARDIZE(array)

```

```

; Compute the mean and variance of each standardized variable using
; the MOMENT function. The skewness and kurtosis are also computed:
FOR K = 0, 3 DO PRINT, MOMENT(result[K,*])

```

IDL prints:

```

25.3050      25.2331      -0.454763      -1.10028
51.1700      27.4012      -0.356958      -1.19516
27.6200      13.3017       0.420289       0.104912
20.1950      26.0731      -0.363277      -1.24886

-7.67130e-07    1.00000      -0.454761      -1.10028
-3.65451e-07    1.00000      -0.356958      -1.19516
-1.66707e-07    1.00000       0.420290       0.104913
 4.21703e-07    1.00000      -0.363278      -1.24886

```

Version History

Introduced: 5.0

See Also

[MOMENT](#)

STDDEV

The STDDEV function computes the standard deviation of an n -element vector.

Syntax

$$\text{Result} = \text{STDDEV}(X [, /\text{DOUBLE}] [, /\text{NAN}])$$

Return Value

The result is the standard deviation of the given vector.

Arguments

X

A numeric vector.

Keywords

DOUBLE

If this keyword is set, computations are performed in double precision arithmetic.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Examples

```
; Define the n-element vector of sample data:
x = [65, 63, 67, 64, 68, 62, 70, 66, 68, 67, 69, 71, 66, 65, 70]

; Compute the standard deviation:
result = STDDEV(x)

PRINT, result
```

IDL prints:

2.65832

Version History

Introduced: 5.1

See Also

[KURTOSIS](#), [MEAN](#), [MEANABSDEV](#), [MOMENT](#), [SKEWNESS](#), [VARIANCE](#)

STOP

The STOP procedure stops the execution of a running program or batch file. Control reverts to the interactive mode.

Syntax

```
STOP [, Expr1, ..., Exprn]
```

Arguments

*Expr*_{*j*}

One or more expressions whose value is printed. If no parameters are present, a brief message describing where the STOP was encountered is printed.

Keywords

None.

Examples

Suppose that you want to stop the execution of a procedure and print the values of the variables A, B, C and NUM. At the appropriate location in your procedure include the command:

```
STOP, A, B, C, NUM
```

To continue execution of the procedure (if possible) enter the IDL executive command:

```
.CONT
```

Version History

Introduced: Original

See Also

[BREAKPOINT](#), [EXIT](#), [WAIT](#)

STRARR

The STRARR function returns a string array containing zero-length strings.

Syntax

$$Result = STRARR(D_1 [, \dots, D_8])$$

Return Value

The result is a string array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Examples

To create S, a 20-element string vector, enter:

```
S = STRARR(20)
```

Version History

Introduced: Original

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#), [LON64ARR](#), [LONARR](#), [MAKE_ARRAY](#), [UINTARR](#), [ULON64ARR](#), [ULONARR](#)

STRCMP

The STRCMP function performs string comparisons between its two String arguments, returning True (1) for those that match and False (0) for those that do not. Normally, the IDL equality operator (EQ) is used for such comparisons, but STRCMP can optionally perform case-insensitive comparisons and can be limited to compare only the first N characters of the two strings, both of which require extra steps using the EQ operator.

Syntax

Result = STRCMP(*String1*, *String2* [, *N*] [, /FOLD_CASE])

Return Value

If all of the arguments are scalar, the result is scalar. If one of the arguments is an array, the result is an integer with the same structure. If more than one argument is an array, the result has the structure of the smallest array. Each element of the result contains True (1) if the corresponding elements of String1 and String2 are the same, and False (0) otherwise.

Arguments

String1, String2

The strings to be compared.

N

Normally String1 and String2 are compared in their entirety. If N is specified, the comparison is made on at most the first N characters of each string.

Keywords

FOLD_CASE

String comparison is normally a case sensitive operation. Set FOLD_CASE to perform case insensitive comparisons instead.

Examples

Compare two strings in a case-insensitive manner, considering only the first 3 characters:

```
Result = STRCMP('Moose', 'moo', 3, /FOLD_CASE)  
PRINT, Result
```

IDL prints:

```
1
```

Version History

Introduced: 5.3

See Also

[STREGEX](#), [STRJOIN](#), [STRMATCH](#), [STRMID](#), [STRPOS](#), [STRSPLIT](#)

“[Non-Printing Characters](#)” in Chapter 6 of the *Building IDL Applications* manual

STRCOMPRESS

The STRCOMPRESS function returns a copy of *String* with all whitespace (blanks and tabs) compressed to a single space or completely removed.

Syntax

```
Result = STRCOMPRESS( String [, /REMOVE_ALL ] )
```

Return Value

The result is a compressed string.

Arguments

String

The string to be compressed. If not of type string, it is converted using IDL's default formatting rules. If *String* is an array, the result is an array with the same structure—each element contains a compressed copy of the corresponding element of *String*.

Keywords

REMOVE_ALL

Set this keyword to remove *all* whitespace. Normally, all whitespace is compressed to a *single* space.

Examples

```
; Create a string variable S:  
S = 'This is a string with spaces in it.'  
  
; Print S with all of the whitespace removed:  
PRINT, STRCOMPRESS(S, /REMOVE_ALL)
```

IDL Output

```
Thisisastringwithspacesinit.
```

Version History

Introduced: Original

See Also

[STRTRIM](#)

STREAMLINE

The STREAMLINE procedure generates the visualization graphics from a path. The output is a polygonal ribbon which is tangent to a vector field along its length. The ribbon is generated by placing a line at each vertex in the direction specified by each normal value multiplied by the anisotropy factor. The input normal array is not normalized before use, making it possible to vary the ribbon width as well.

Syntax

```
STREAMLINE, Verts, Conn, Normals, Outverts, Outconn [, ANISOTROPY=array]  
[, SIZE=vector] [, PROFILE=array]
```

Arguments

Verts

Input array of path vertices ($[3, n]$ array).

Conn

Input path connectivity array in IDLgrPolyline POLYLINES keyword format. There is one set of line segments in this array for each streamline.

Normals

Normal estimate at each input vertex ($[3, n]$ array).

Outverts

Output vertices ($[3 \times n]$ float array). Useful if the routine is to be used with Direct Graphics or the user wants to manipulate the data directly.

Outconn

Output polygonal connectivity array to match the output vertices.

Keywords

ANISOTROPY

Set this input keyword to a three-element array describing the distance between grid points in each dimension. The default value is $[1.0, 1.0, 1.0]$

SIZE

Set this keyword to a vector of values (one for each path point). These values are used to specify the width of the ribbon or the size of profile at each point along its path. This keyword is generally used to convey additional data parameters along the streamline.

PROFILE

Set this keyword to an array of two-dimensional points which are treated as the cross section of the ribbon instead of a line segment. If the first and last points in the array are the same, a closed profile is generated. The profile is placed at each path vertex in the plane perpendicular to the line connecting each path vertex with the vertex normal defining the up direction. This allows for the generation of streamtubes and other geometries.

Version History

Introduced: 5.3

STREGEX

The STREGEX function performs regular expression matching against the strings contained in `StringExpression`. STREGEX can perform either a simple boolean True/False evaluation of whether a match occurred, or it can return the position and offset within the strings for each match. The regular expressions accepted by this routine, which correspond to “Posix Extended Regular Expressions”, are similar to those used by such UNIX tools as `egrep`, `lex`, `awk`, and `Perl`.

For more information about regular expressions, see “[Learning About Regular Expressions](#)” in Chapter 6 of the *Building IDL Applications* manual.

STREGEX is based on the `regex` package written by Henry Spencer, modified by RSI only to the extent required to integrate it into IDL. This package is freely available at <ftp://zoo.toronto.edu/pub/regex.shar>.

Syntax

```
Result = STREGEX( StringExpression, RegularExpression [, /BOOLEAN |  
                , /EXTRACT | , LENGTH=variable [, /SUBEXPR]] [, /FOLD_CASE] )
```

Return Value

By default, STREGEX returns the position and length of the matched string within `StringExpression`. If no match is found, -1 is returned for both of these. Optionally, it can return a Boolean True/False result of the match, or the matched strings.

Arguments

StringExpression

String to be matched.

RegularExpression

A scalar string containing the regular expression to match. See “[Learning About Regular Expressions](#)” in Chapter 6 of the *Building IDL Applications* manual for a description of the meta characters that can be used in a regular expression.

Keywords

BOOLEAN

Normally, STREGEX returns the position of the first character in StringExpression that matches RegularExpression. Setting BOOLEAN modifies this behavior to simply return a True/False value indicating if a match occurred or not.

EXTRACT

Normally, STREGEX returns the position of the first character in StringExpression that matches RegularExpression. Setting EXTRACT modifies this behavior to simply return the matched substrings. The EXTRACT keyword cannot be used with either BOOLEAN or LENGTH.

FOLD_CASE

Regular expression matching is normally a case-sensitive operation. Set FOLD_CASE to perform case-insensitive matching instead.

LENGTH

If present, specifies a variable to receive the lengths of the matches. Together with this result of this function, which contains the starting points of the matches in StringExpression, LENGTH can be used with the STRMID function to extract the matched substrings. The LENGTH keyword cannot be used with either BOOLEAN or EXTRACT.

SUBEXPR

By default, STREGEX only reports the overall match. Setting SUBEXPR causes it to report the overall match as well as any subexpression matches. A subexpression is any part of a regular expression written within parentheses. For example, the regular expression '(a)(b)(c+)' has 3 subexpressions, whereas the functionally equivalent 'abc+' has none. The SUBEXPR keyword cannot be used with BOOLEAN.

If a subexpression participated in the match several times, the reported substring is the last one it matched. Note, as an example in particular, that when the regular expression '(b*)+' matches 'bbb', the parenthesized subexpression matches the three 'b's and then an infinite number of empty strings following the last 'b', so the reported substring is one of the empties. This occurs because the '*' matches *zero or more* instances of the character that precedes it.

In order to return multiple positions and lengths for each input, the result from SUBEXPR has a new first dimension added compared to StringExpression.

Examples

Example 1

To match a string starting with an “a”, followed by a “b”, followed by 1 or more “c”:

```
pos = STREGEX('aaabccc', 'abc+', length=len)
PRINT, STRMID('aaabccc', pos, len)
```

IDL Prints:

```
abccc
```

To perform the same match, and also find the locations of the three parts:

```
pos = STREGEX('aaabccc', '(a)(b)(c+)', length=len, /SUBEXPR)
print, STRMID('aaabccc', pos, len)
```

IDL Prints:

```
abccc a b ccc
```

Or more simply:

```
print, STREGEX('aaabccc', '(a)(b)(c+)', /SUBEXPR, /EXTRACT)
```

IDL Prints:

```
abccc a b ccc
```

Example 2

This example searches a string array for words of any length beginning with “f” and ending with “t” without the letter “o” in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'affluent']
PRINT, STREGEX(str, '^f[^o]*t$', /EXTRACT, /FOLD_CASE)
```

This statement results in:

```
Feet FAST ferret
```

Note the following about this example:

- Unlike the * wildcard character used by STRMATCH, the * meta character used by STREGEX applies to the item directly on its left, which in this case is [^o], meaning “any character except the letter ‘o’”. Therefore, [^o]* means “zero or more characters that are not ‘o’”, whereas the following statement would find only words whose second character is not “o”:

```
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

- The anchors (^ and \$) tell STREGEX to find only words that begin with “f” and end with “t”. If we left out the ^ anchor in the above example, STREGEX would also return “ffluent” (a substring of “affluent”). Similarly, if we left out the \$ anchor, STREGEX would also return “fat” (a substring of “fate”).

Version History

Introduced: 5.3

See Also

[STRCMP](#), [STRJOIN](#), [STRMATCH](#), [STRMID](#), [STRPOS](#), [STRSPLIT](#)

“[Non-Printing Characters](#)” in Chapter 6 of the *Building IDL Applications* manual

STRETCH

The STRETCH procedure stretches the image display color tables so the full range runs from one color index to another. The modified colortable is loaded, but the COLORS common block is not changed. The original colortable can be restored by calling STRETCH with no arguments. A colortable must be loaded before STRETCH can be called.

This routine is written in the IDL language. Its source code can be found in the file `stretch.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
STRETCH [, Low, High [, Gamma]] [, /CHOP]
```

Arguments

Low

The lowest pixel value to use. If this parameter is omitted, 0 is assumed. Appropriate values range from 0 to the number of available colors-1. If no parameters are supplied, the original color tables are restored.

High

The highest pixel value to use. If this parameter is omitted, the number of colors-1 is assumed. Appropriate values range from 0 to the number of available colors-1.

Gamma

An optional Gamma correction factor. If this value is omitted, 1.0 is assumed. Gamma correction works by raising the color indices to the *Gamma* power, assuming they are scaled into the range 0 to 1.

Keywords

CHOP

Set this keyword to set color indices above the upper threshold to color index 0. Normally, values above the upper threshold are set to the maximum color index.

Examples

Load the STD GAMMA-II color table by entering:

```
LOADCT, 5
```

Create and display an image by entering:

```
TVSCL, DIST(300)
```

Now adjust the color table with STRETCH. Make the entire color table fit in the range 0 to 70 by entering:

```
STRETCH, 0, 70
```

Notice that pixel values above 70 are now colored white. Restore the original color table by entering:

```
STRETCH
```

Version History

Introduced: Original

See Also

[GAMMA_CT](#), [H_EQ_CT](#), [MULTI](#), [XLOADCT](#)

STRING

The `STRING` function returns its arguments converted to string type. It is similar to the `PRINT` procedure, except that its output is placed in a string rather than being output to the terminal. The case in which a single expression of type byte is specified without the `FORMAT` keyword is special—see the “[Differences Between `STRING` and `PRINT`](#)” on page 2027 for details.

Note

Applying the `STRING` function to a byte array containing a null (zero) value will result in the resulting string being truncated at that position.

Syntax

```
Result = STRING( Expression1, ..., Expressionn [, AM_PM=[string, string]]
  [, DAYS_OF_WEEK=string_array{7 names}] [, FORMAT=value]
  [, MONTHS=string_array{12 names}] [, /PRINT] )
```

Return Value

Returns a string containing the specified expression.

Arguments

Expression_n

The expressions to be converted to string type.

Note

If you supply a comma-separated list of expressions without specifying a `FORMAT`, and the combined length of the expressions is greater than the current width of your tty or command log window, `STRING` will create a string array with one element for each expression, rather than concatenating all expressions into a single string. In this case, you can either specify a `FORMAT`, or use the string concatenation operator, “+”.

Keywords

AM_PM

Supplies a string array of 2 names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the FORMAT keyword.

DAYS_OF_WEEK

Supplies a string array of 7 names to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the FORMAT keyword.

FORMAT

A format string to be used in formatting the expressions. See [“Using Explicitly Formatted Input/Output”](#) in Chapter 11 of the *Building IDL Applications* manual.

MONTHS

Supplies a string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the FORMAT keyword.

PRINT

Set this keyword to specify that any special case processing should be ignored and that STRING should behave exactly as the PRINT procedure would.

Differences Between STRING and PRINT

The behavior of STRING differs from the behavior of the PRINT procedure in the following ways (unless the PRINT keyword is set):

- When called with a single non-byte argument and no format specification, STRING returns a result that has the same dimensions as the original argument. For example, the statement:

```
HELP, STRING(INDGEN(5))
```

gives the result:

```
<Expression> STRING = Array[5]
```

while:

```
HELP, STRING(INDGEN(5), /PRINT)
```

results in:

```
<Expression> STRING = ' 0 1 2 3 4'
```

- If called with a single argument of byte type and the **FORMAT** keyword is not used, **STRING** simply stores the unmodified values of each byte element in the result. This result is a string containing the byte values from the original argument. Thus, the result has one less dimension than the original argument. For example, a 2-dimensional byte array becomes a vector of strings, a byte vector becomes a scalar string. However, a byte scalar also becomes a string scalar. For example, the statement:

```
PRINT, STRING([72B, 101B, 108B, 108B, 111B])
```

produces the output:

```
Hello
```

because the argument to **STRING**, is a byte vector. Its first element is a 72B which is the ASCII code for “H”, the second is 101B which is an ASCII “e”, and so forth.

- If both the **FORMAT** and **PRINT** keywords are not present and **STRING** is called with more than one argument, and the last argument is a scalar string starting with the characters “\$(” or “(”, this final argument is taken to be the format specification, just as if it had been specified via the **FORMAT** keyword. This feature is maintained for compatibility with version 1 of VMS IDL.

Examples

To convert the contents of variable **A** to string type and store the result in the variable **B**, enter:

```
B = STRING(A)
```

Version History

Introduced: Original

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#), [LONG64](#), [UINT](#), [ULONG](#), [ULONG64](#)

STRJOIN

The STRJOIN function collapses a one-dimensional string array into a single merged string or two-dimensional string array into a one-dimensional array of merged strings. The strings in the removed first dimension are concatenated into a single string using the string in *Delimiter* to separate them.

Syntax

```
Result = STRJOIN( String [, Delimiter] [, /SINGLE] )
```

Return Value

Returns the merged strings.

Arguments

String

A string array to be collapsed into merged strings.

Delimiter

The separator string to use between the joined strings. If Delimiter is not specified, an empty string is used.

Keywords

SINGLE

If SINGLE is set, the entire String is joined into a single scalar string result.

Examples

Replace all the blanks in a sentence with colons:

```
str = 'Out, damned spot! Out I say!'
print, (STRJOIN(STRSPLIT(str, /EXTRACT), ':'))
```

IDL prints:

```
Out,:damned:spot!:Out:I:say!
```

Version History

Introduced: 5.3

See Also

[STRCMP](#), [STREGEX](#), [STRMATCH](#), [STRMID](#), [STRPOS](#), [STRSPLIT](#)

STRLEN

The STRLEN function returns the length of its string-type argument. If the argument is not a string, it is first converted to string type.

Syntax

Result = STRLEN(*Expression*)

Return Value

Returns the string length.

Arguments

Expression

The expression for which the string length is desired. If this parameter is not a string, it is converted using IDL's default formatting rules in order to determine the length. The result is a long integer. If *Expression* is an array, the result is a long integer array with the same structure, where each element contains the length of the corresponding *Expression* element.

Keywords

None.

Examples

To find the length of the string "IDL is fun" and print the result, enter:

```
PRINT, STRLEN('IDL is fun')
```

IDL prints:

```
10
```

Version History

Introduced: Original

STRLOWCASE

The STRLOWCASE function returns a copy of *String* converted to lowercase characters. Only uppercase characters are modified—lowercase and non-alphabetic characters are copied without change.

Syntax

Result = STRLOWCASE(*String*)

Return Value

Returns a string composed of lowercase characters.

Arguments

String

The string to be converted. If this argument is not a string, it is converted using IDL's default formatting rules. If *String* is an array, the result is an array with the same structure—each element contains a lower case copy of the corresponding element of *String*.

Keywords

None.

Examples

To convert the string “IDL is fun” to all lowercase characters and print the result, enter:

```
PRINT, STRLOWCASE('IDL is fun')
```

IDL prints:

```
idl is fun
```

Version History

Introduced: Original

See Also

[STRUPCASE](#)

STRMATCH

The STRMATCH function compares its search string, which can contain wildcard characters, against the input string expression. The result is an array with the same structure as the input string expression. Those elements that match the corresponding input string are set to True (1), and those that do not match are set to False (0).

The wildcards understood by STRMATCH are similar to those used by the standard UNIX shell:

Wildcard Character	Description
*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters. A pair of characters separated by “-” matches any character lexically between the pair, inclusive. If the first character following the opening [is a !, any character not enclosed is matched. To prevent one of these characters from acting as a wildcard, it can be quoted by preceding it with a backslash character (e.g. “*” matches the asterisk character). Quoting any other character (including \ itself) is equivalent to the character (e.g. “\a” is the same as “a”).

Table 3-110: Wildcard Characters used by STRMATCH

Syntax

Result = STRMATCH(*String*, *SearchString* [, /FOLD_CASE])

Return Value

Returns 1 if the pattern specified by *SearchString* exists in *String*, or 0 otherwise. If the *String* argument contains an array of strings, the result is an array of 1s and 0s with the same number of elements as *String*, indicating which elements contain *SearchString*.

Arguments

String

A scalar string or string array to be searched.

SearchString

A scalar string containing the pattern to search for in *String*. The pattern string can contain wildcard characters as discussed above.

Keywords

FOLD_CASE

The comparison is usually case sensitive. Setting the FOLD_CASE keyword causes a case insensitive match to be done instead.

Examples

Example 1

Find all 4-letter words in a string array that begin with “f” or “F” and end with “t” or “T”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f??t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST fort
```

Example 2

Find words of any length that begin with “f” and end with “t”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet FAST ferret fort
```

Example 3

Find 4-letter words beginning with “f” and ending with “t”, with any combination of “o” and “e” in between:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[eo][eo]t', /FOLD_CASE) EQ 1)]
```

This results in:

```
foot Feet
```

Example 4

Find all words beginning with “f” and ending with “t” whose second character is not the letter “o”:

```
str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
PRINT, str[WHERE(STRMATCH(str, 'f[!o]*t', /FOLD_CASE) EQ 1)]
```

This results in:

```
Feet FAST ferret
```

Version History

Introduced: 5.3

See Also

[STRCMP](#), [STRJOIN](#), [STREGEX](#), [STRMID](#), [STRPOS](#), [STRSPLIT](#)

“Non-Printing Characters” in Chapter 6 of the *Building IDL Applications* manual

STRMESSAGE

The STRMESSAGE function returns the text of the error message specified by *Err*. This function is especially useful in conjunction with the CODE field of the !ERROR_STATE system variable which always contains the error number of the last error. The MSG field of the !ERROR_STATE system variable contains the text of the last error message.

Syntax

Result = STRMESSAGE(*Err* [, /BLOCK | , /CODE | , /NAME])

Return Value

Returns the error message text.

Arguments

Err

The error number or text. Programs must not make the assumption that certain error numbers are always related to certain error messages—the actual correspondence changes over time as IDL is modified.

Keywords

BLOCK

Set this keyword to return the name of the message block that defines *Err*. If this keyword is specified, *Err* must be an error code.

CODE

Set this keyword to return the error code for the error message specified in *Err*. If this keyword is specified, *Err* must be an error name.

NAME

Set this keyword to return a string containing the error message that goes with *Err*. If this keyword is specified, *Err* must be an error code.

Examples

Print the error message associated with error number 4 by entering:

```
PRINT, STRMESSAGE(4)
```

Version History

Introduced: Original

See Also

[MESSAGE](#)

STRMID

The STRMID function extracts one or more substrings from a string expression. Each extracted string is the result of removing characters.

Syntax

Result = STRMID(*Expression*, *First_Character* [, *Length*] [, /REVERSE_OFFSET])

Return Value

The result of the function is a string of *Length* characters taken from *Expression*, starting at character position *First_Character*.

The form of *First_Character* and *Length* control how they are applied to *Expression*. Either argument can be a scalar or an array:

- If a scalar value is supplied for *First_Character* and *Length*, then those values are applied to all elements of *Expression*. The result has the same structure and number of elements as *Expression*.

If *First_Character* or *Length* is an array, the size of their first dimension determines how many substrings are extracted from each element of *Expression*. We call this the “stride”. If both are arrays, they must have the same stride. If *First_Character* or *Length* do not contain enough elements to process *Expression*, STRMID automatically loops back to the beginning as necessary. Excess values are ignored. If the stride is 1, the result will have the same structure and number of elements as *Expression*. If it is greater than 1, the result will have an additional dimension, with the new first dimension having the same size as the stride.

Arguments

Expression

The expression from which the substrings are to be extracted. If this argument is not a string, it is converted using IDL's default formatting rules.

First_Character

The starting position within *Expression* at which the substring starts. The first character position is 0.

Length

The length of the substring. If there are not enough characters left in the main string to obtain *Length* characters, the substring is truncated. If *Length* is not supplied, STRMID extracts all characters from the specified start position to the end of the string.

Keywords

REVERSE_OFFSET

Specifies that *First_Character* should be counted from the end of the string backwards. This allows simple extraction of strings from the end.

Examples

If the variable B contains the string “IDL is fun”, the substring “is” can be extracted and stored in the variable C with the command:

```
C = STRMID(B, 4, 2)
```

Version History

Introduced: Original

See Also

[STRPOS](#), [STRPUT](#), [STRTRIM](#)

STRPOS

The STRPOS function finds the first occurrence of a substring within an object string.

Syntax

```
Result = STRPOS( Expression, Search_String [, Pos] [, /REVERSE_OFFSET]  
                [, /REVERSE_SEARCH] )
```

Return Value

If *Search_String* occurs in *Expression*, STRPOS returns the character position of the match, otherwise it returns -1.

Arguments

Expression

The expression in which to search for the substring. If this argument is not a string, it is converted using IDL's default formatting rules. If *Expression* is an array, the result is an array with the same structure, where each element contains the position of the substring within the corresponding element *Expression*. If *Expression* is the null string, STRPOS returns the value -1.

Search_String

The substring to be searched for within *Expression*. If this argument is not a string, it is converted using IDL's default formatting rules. If *Search_String* is the null string, STRPOS returns the smaller of *Pos* or one less than the length of *Expression*.

Pos

The character position at which the search is begun. If *Pos* is omitted and the REVERSE_SEARCH keyword is not set, the search begins at the first character (character position 0). If REVERSE_SEARCH is set, the default is to start at the last character in the string. If *Pos* is less than zero, zero is used for the starting position.

Keywords

REVERSE_OFFSET

Normally, the value of *Pos* is used as an offset from the beginning of the expression towards the end. Set `REVERSE_OFFSET` to use it as an offset from the last character of the string moving towards the beginning. This keyword makes it easy to position the starting point of the search at a fixed offset from the end of the string.

REVERSE_SEARCH

`STRPOS` usually starts at *Pos* and moves toward the end of the string looking for a match. If `REVERSE_SEARCH` is set, the search instead moves towards the beginning of the string.

Examples

Example 1

Find the position of the string “fun” within the string “IDL is fun” and print the result by entering:

```
PRINT, STRPOS('IDL is fun', 'fun')
```

IDL prints:

```
7
```

Example 2

The `REVERSE_SEARCH` keyword to the `STRPOS` function makes it easy to find the last occurrence of a substring within a string. In the following example, we search for the last occurrence of the letter “I” (or “i”) in a sentence:

```
sentence = 'IDL is fun.'  
sentence = STRUPCASE(sentence)  
lasti = STRPOS(sentence, 'I', /REVERSE_SEARCH)  
PRINT, lasti
```

This results in:

```
4
```

Note that although `REVERSE_SEARCH` tells `STRPOS` to begin searching from the end of the string, the `STRPOS` function still returns the position of the search string from the beginning of the string (where 0 is the position of the first character).

Version History

Introduced: Original

See Also

[STRMID](#), [STRPUT](#), [STRTRIM](#)

“[Non-Printing Characters](#)” in Chapter 6 of the *Building IDL Applications* manual

STRPUT

The STRPUT procedure inserts the contents of one string into another. The source string, *Source*, is inserted into the destination string, *Destination*, starting at the given position, *Position*. Characters in *Destination* before the starting position and after the starting position plus the length of *Source* remain unchanged. The length of the destination string is not changed. If the insertion extends past the end of the destination, it is clipped at the end.

Syntax

STRPUT, *Destination*, *Source* [, *Position*]

Arguments

Destination

The named string variable into which *Source* is inserted. *Destination* must be a named variable of type string. If it is an array, *Source* is inserted into every element of the array.

Source

A scalar string to be inserted into *Destination*. If this argument is not a string, it is converted using IDL's default formatting rules.

Position

The character position at which the insertion is begun. If *Position* is omitted, the insertion begins at the first character (character position 0). If *Position* is less than zero, zero is used for the initial position.

Keywords

None.

Examples

If the variable A contains the string "IBM is fun", the substring "IBM" can be overwritten with the string "IDL" by entering:

```
STRPUT, A, 'IDL', 0
```

The following commands demonstrate the clipping of output that extends past the end of the destination string:

```
STRPUT, A, 'FUNNY', 7  
PRINT, A
```

IDL prints:

```
IDL is FUN
```

Version History

Introduced: Original

See Also

[STRMID](#), [STRPOS](#), [STRTRIM](#)

STRSPLIT

The STRSPLIT function splits its input *String* argument into separate substrings, according to the specified delimiter or regular expression. By default, the position of the substrings is returned. The EXTRACT keyword can be used to cause STRSPLIT to return an array containing the substrings.

Syntax

```
Result = STRSPLIT( String [, Pattern] [, COUNT=variable] [, ESCAPE=string |
, /REGEX [, /FOLD_CASE]] [, /EXTRACT | , LENGTH=variable]
[, /PRESERVE_NULL] )
```

Return Value

Returns an array containing either the positions of the substrings or the substrings themselves (if the EXTRACT keyword is specified).

Arguments

String

A scalar string to be split into substrings.

Pattern

A scalar string that can contain one of two types of information:

- One or more *single characters*, each of which is considered to be a separator. *String* will be split when any of the characters is detected. For example, if *Pattern* is " , " *String* will be split whenever *either* a space or a comma is detected. In this case, IDL performs a simple string search for the specified characters. This method is simple and fast.
- If the REGEX keyword is specified, *Pattern* is considered to be a single regular expression (as implemented by the [STREGEX](#) function). This method is slower and more complex, but can handle extremely complicated *Pattern* strings.

In either case, if the EXTRACT keyword is specified, the separator characters are *not* included in the result.

Note

Pattern is an optional argument. If it is not specified, STRSPLIT defaults to splitting on spans of whitespace (space or tab characters) in *String*.

Keywords

COUNT

Set this keyword to a named variable that will contain the number of matched substrings returned by STRSPLIT. This value will be 0 if either of the *String* or *Pattern* arguments is null. Otherwise, it will contain the number of elements in the *Result* array.

ESCAPE

When doing simple pattern matching, the ESCAPE keyword can be used to specify any characters that should be considered to be “escape” characters. Preceding any character with an escape character prevents STRSPLIT from treating it as a separator character even if it is found in *Pattern*.

Note that if the EXTRACT keyword is set, STRSPLIT will automatically remove the escape characters from the resulting substrings. If EXTRACT is not specified, STRSPLIT cannot perform this editing, and the returned position and offsets will include the escape characters.

For example:

```
print, STRSPLIT('a\b,c', ' ', ESCAPE='\', /EXTRACT)
```

IDL prints:

```
a,b c
```

ESCAPE cannot be specified with the FOLD_CASE or REGEX keywords.

EXTRACT

By default, STRSPLIT returns an array of character offsets into *String* that indicate where the substrings are located. These offsets, along with the lengths available from the LENGTH keyword can be used later with STRMID to extract the substrings. Set EXTRACT to bypass this step, and cause STRSPLIT to return the substrings. EXTRACT cannot be specified with the LENGTH keyword.

FOLD_CASE

Indicates that the regular expression matching should be done in a case-insensitive fashion. FOLD_CASE can only be specified if the REGEX keyword is set, and cannot be used with the ESCAPE keyword.

LENGTH

Set this keyword to a named variable to receive the lengths of the substrings. Together with this result of this function, LENGTH can be used with the STRMID function to extract the matched substrings. The LENGTH keyword cannot be used with the EXTRACT keyword.

PRESERVE_NULL

Normally, STRSPLIT will not return null length substrings unless there are no non-null values to report, in which case STRSPLIT will return a single null string. Set PRESERVE_NULL to cause all null substrings to be returned.

REGEX

For complex splitting tasks, the REGEX keyword can be specified. In this case, *Pattern* is taken to be a regular expression to be matched against *String* to locate the separators. If REGEX is specified and *Pattern* is not, the default *Pattern* is the regular expression:

```
'[ ' + STRING(9B) + ' ]+'
```

which means “any series of one or more space or tab characters” (9B is the byte value of the ASCII TAB character).

Note that the default *Pattern* contains a space after the [character.

The REGEX keyword cannot be used with the ESCAPE keyword.

Note

If *Pattern* specifies a single multi-character separator pattern (as contrasted with a string of two or more individual separator characters), you *must* specify the REGEX keyword.

Examples

Example 1

To split a string on spans of whitespace and replace them with hyphens:

```
Str = 'STRSPLIT chops up strings.'
print, STRJOIN(STRSPLIT(Str, /EXTRACT), '-')
```

IDL prints:

```
STRSPLIT-chops-up-strings.
```

Example 2

As an example of a more complex splitting task that can be handled with the simple character-matching mode of STRSPLIT, consider a sentence describing different colored ampersand characters. For unknown reasons, the author used commas to separate all the words, and used ampersands or backslashes to escape the commas that actually appear in the sentence (which therefore should not be treated as separators). The unprocessed string looks like:

```
Str = 'There,was,a,red,&&&, ,a,yellow,&&\, ,and,a,blue,\&.'
```

We use STRSPLIT to break this line apart, and STRJOIN to reassemble it as a standard blank-separated sentence:

```
S = STRSPLIT(Str, ',', ESCAPE='&\', /EXTRACT)
PRINT, STRJOIN(S, ' ')
```

IDL prints:

```
There was a red &, a yellow &, and a blue &.
```

Example 3

Strings separated by multi-character delimiters cannot be split using the simple character matching mode of STRSPLIT. Such delimiters require the use of a regular expression. For instance, consider splitting the following string on double ampersand boundaries.

```
str = 'red&&blue&&yellow&&odds&ends'
```

The desired result of such splitting would be four strings, with the values 'red', 'blue', 'yellow', and 'odds&ends'. You might be tempted to use STRSPLIT as follows:

```
PRINT, STRSPLIT(str, '&&', /EXTRACT)
```

which causes IDL to print:

```
red blue yellow odds ends
```

IDL split the string on single ampersand boundaries, yielding 5 strings instead of the desired 4. When using the simple character matching mode of STRSPLIT, the characters in the Pattern argument specify a set of possible single character delimiters. The order of these characters is unimportant, and specifying a character more than once has no effect (the extras are ignored).

To properly split the above string using a regular expression:

```
print, strsplit(str, '&&', /EXTRACT, /REGEX)
```

producing the desired IDL output:

```
red blue yellow odds&ends
```

Example 4

Finally, suppose you had a complicated string, in which every token was preceded by the count of characters in that token, with the count enclosed in angle brackets:

```
str = '<4>What<1>a<7>tangled<3>web<2>we<6>weave.'
```

This is too complex to handle with simple character matching, but can be easily handled using the regular expression '<[0-9]+>' to match the separators. This regular expression can be read as “an opening angle bracket, followed by one or more numeric characters between 0 and 9, followed by a closing angle bracket.” The STRJOIN function is used to glue the resulting substrings back together:

```
S = STRSPLIT(str, '<[0-9]+>', /EXTRACT, /REGEX)
PRINT, STRJOIN(S, ' ')
```

IDL prints:

```
What a tangled web we weave.
```

Version History

Introduced: 5.3

COUNT keyword added: 6.0

See Also

[STRCMP](#), [STRJOIN](#), [STRMATCH](#), [STREGEX](#), [STRMID](#), [STRPOS](#)

STRTRIM

The STRTRIM function removes leading and/or trailing blank from the input *String*.

Syntax

Result = STRTRIM(*String* [, *Flag*])

Return Value

Returns a copy of string with the specified blank spaces removed.

Arguments

String

The string to have leading and/or trailing blanks removed. If this argument is not a string, it is converted using IDL's default formatting rules. If it is an array, the result is an array with the same structure where each element contains a trimmed copy of the corresponding element of *String*.

Flag

A value that controls the action of STRTRIM. If *Flag* is zero or not present, trailing blanks are removed. Leading blanks are removed if it is equal to 1. Both are removed if it is equal to 2.

Examples

Converting variables to string type often results in undesirable leading blanks. For example, the following command converts the integer 56 to string type:

```
C = STRING(56)
```

Entering the command:

```
HELP, C
```

IDL prints:

```
C          STRING = '      56'
```

which shows that there are six leading spaces before the characters 5 and 6. To remove these leading blanks, enter the command:

```
C = STRTRIM(C, 1)
```

To confirm that the blanks were removed, enter:

```
HELP, C
```

IDL prints:

```
C          STRING = '56'
```

Version History

Introduced: Original

See Also

[STRMID](#), [STRPOS](#), [STRPUT](#), [STRSPLIT](#)

STRUCT_ASSIGN

The IDL “=” operator is unable to assign a structure value to a structure of a different type. The STRUCT_ASSIGN procedure performs “relaxed structure assignment,” which is a field-by-field copy of a structure to another structure. Fields are copied according to the following rules:

1. Any fields found in the destination structure that are not found in the source structure are “zeroed” (set to zero, the empty string, or a null pointer or object reference depending on the type of field).
2. Any fields in the source structure that are not found in the destination structure are quietly ignored.
3. Any fields that are found in both the source and destination structures are copied one at a time. If necessary, type conversion is done to make their types agree. If a field in the source structure has fewer data elements than the corresponding field in the destination structure, then the “extra” elements in the field in the destination structure are zeroed. If a field in the source structure has more elements than the corresponding field in the destination structure, the extra elements are quietly ignored.

Relaxed structure assignment is especially useful when restoring structures from disk files into an environment where the structure definition has changed. See the description of the RELAXED_STRUCTURE_ASSIGNMENT keyword to the RESTORE procedure for additional details. “Relaxed Structure Assignment” in Chapter 9 of the *Building IDL Applications* manual provides a more in-depth discussion of the structure-definition process.

Syntax

```
STRUCT_ASSIGN, Source, Destination [, /NOZERO] [, /VERBOSE]
```

Arguments

Source

A named variable or element of an array containing a structure, the contents of which will be assigned to the structure specified by the *Destination* argument. *Source* can be an object reference if STRUCT_ASSIGN is called inside an object method.

Destination

A named variable containing a structure into which the contents of the structure specified by the *Source* argument will be inserted. *Destination* can be an object reference if `STRUCT_ASSIGN` is called inside an object method.

Keywords

NOZERO

Normally, any fields found in the destination structure that are not found in the source structure are zeroed. Set `NOZERO` to prevent this action and leave the original contents of such fields unchanged.

VERBOSE

Set this keyword to cause `STRUCT_ASSIGN` to issue informational messages about any incompatibilities that prevent data from being copied.

Examples

The following example creates two anonymous structures, then uses `STRUCT_ASSIGN` to insert the contents of the first into the second:

```
source = { a: FINDGEN(4), b: 12 }
dest = { a: INDGEN(2), c: 20 }
STRUCT_ASSIGN, /VERBOSE, source, dest
```

IDL prints:

```
% STRUCT_ASSIGN: <Anonymous> tag A is longer than destination.
                  The end will be clipped.
% STRUCT_ASSIGN: Destination lacks <Anonymous> tag B. Not copied.
```

After assignment, `dest` contains a two-element integer array [0, 1] in its field A and the integer 0 in its field C. Since `dest` does not have a field B, field B from `source` is not copied.

Version History

Introduced: 5.1

STRUCT_HIDE

The IDL HELP procedure displays information on all known structures or object classes when used with the STRUCTURES or OBJECTS keywords. Although this is usually the desired behavior, authors of large vertical applications or library routines may wish to prevent IDL from displaying information on structures or objects that are not part of their public interface, but which exist solely in support of the internal implementation. The STRUCT_HIDE procedure is used to mark such structures or objects as hidden. Items so marked are not displayed by HELP, /STRUCTURE unless the user sets the FULL keyword, but are otherwise unaltered.

Note

STRUCT_HIDE is primarily intended for use with named structures or objects. Although it can be safely used with anonymous structures, there is no visible benefit to doing so as anonymous structures are hidden by default.

Tip

Authors of objects will often place a call to STRUCT_HIDE in the __DEFINE procedure that defines the structure.

Syntax

```
STRUCT_HIDE, Arg1 [, Arg2, ..., Argn]
```

Arguments

Arg₁, ..., Arg_n

If an argument is a variable of one of the following types, its underlying structure and/or object definition is marked as being hidden from the HELP procedure's default output:

- Structure
- Pointer that refers to a heap variable of structure type
- Object Reference

Any arguments that are not one of these types are quietly ignored. No change is made to the value of any argument.

Keywords

None

Examples

This example shows how a structure can be hidden if an application designer doesn't want end-users to be able to see it, but variables are not hidden. To create a named structure called `bullwinkle` and prevent it from appearing in the `HELP` procedure's default output, do the following.

```
; Define a variable containing the named structure:
tmp = { bullwinkle, moose:1, squirrel:0 }
; IDL returns BULLWINKLE in addition to the other system variables.
HELP, /STRUCTURE, /BRIEF
; Next, specifically hide the structure using
; the STRUCT_HIDE procedure.
STRUCT_HIDE, tmp
; This time IDL returns the system variables but
; not the BULLWINKLE structure.
HELP, /STRUCTURE, /BRIEF
; IDL returns the variable tmp showing that it is
; a named structure called BULLWINKLE.
HELP, tmp
```

Version History

Introduced: 5.3

See Also

[COMPILE_OPT](#)

STRUPCASE

The STRUPCASE function returns a copy of *String* converted to upper case. Only lowercase characters are modified—uppercase and non-alphabetic characters are copied without change.

Syntax

Result = STRUPCASE(*String*)

Return Value

Returns a string composed of uppercase characters.

Arguments

String

The string to be converted. If this argument is not a string, it is converted using IDL's default formatting rules. If it is an array, the result is an array with the same structure where each element contains an uppercase copy of the corresponding element of *String*.

Keywords

None.

Examples

To print an uppercase version of the string “IDL is fun”, enter:

```
PRINT, STRUPCASE('IDL is fun')
```

IDL prints:

```
IDL IS FUN
```

Version History

Introduced: Original

See Also

[STRLOWCASE](#)

SURFACE

The SURFACE procedure draws a wire-mesh representation of a two-dimensional array projected into two dimensions, with hidden lines removed.

Restrictions

If the (X, Y) grid is not regular or nearly regular, errors in hidden line removal occur. The TRIGRID and TRIANGULATE routines can be used to interpolate irregularly-gridded data points to a regular grid before plotting.

If the T3D keyword is set, the 3-D to 2-D transformation matrix contained in !P.T must project the Z axis to a line parallel to the device Y axis, or errors will occur.

The surface lines may blend together when drawing large arrays, especially on low or medium resolution displays. Use the REBIN or CONGRID procedure to resample the array to a lower resolution before plotting.

Syntax

```
SURFACE, Z [, X, Y] [, AX=degrees] [, AZ=degrees] [, BOTTOM=index]
  [, /HORIZONTAL] [, /LEGO] [, /LOWER_ONLY | /UPPER_ONLY]
  [, MAX_VALUE=value] [, MIN_VALUE=value] [, /SAVE] [, SHADES=array]
  [, SKIRT=value] [, /XLOG] [, /YLOG] [, ZAXIS={1 | 2 | 3 | 4}] [, /ZLOG]
```

Graphics Keywords: Accepts all graphics keywords accepted by PLOT except for: PSYM, SYMSIZE. See “[Graphics Keywords Accepted](#)” on page 2062.

Arguments

Z

The two-dimensional array to be displayed. If X and Y are provided, the surface is plotted as a function of the (X, Y) locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of Z .

This argument is converted to double-precision floating-point before plotting. Plots created with SURFACE are limited to the range and precision of double-precision floating-point values.

X

A vector or two-dimensional array specifying the X coordinates of the grid. If this argument is a vector, each element of X specifies the X coordinate for a column of Z

(e.g., $X[0]$ specifies the X coordinate for $Z[0, *]$). If X is a two-dimensional array, each element of X specifies the X coordinate of the corresponding point in Z (X_{ij} specifies the X coordinate for Z_{ij}).

This argument is converted to double-precision floating-point before plotting.

Y

A vector or two-dimensional array specifying the Y coordinates of the grid. If this argument is a vector, each element of Y specifies the Y coordinate for a row of Z (e.g., $Y[0]$ specifies the Y coordinate for $Z[* , 0]$). If Y is a two-dimensional array, each element of Y specifies the Y coordinate of the corresponding point in Z (Y_{ij} specifies the Y coordinate for Z_{ij}).

This argument is converted to double-precision floating-point before plotting.

Keywords

AX

This keyword specifies the angle of rotation, about the X axis, in degrees towards the viewer. This keyword is effective only if !P.T3D is not set. If !P.T3D is set, the three-dimensional to two-dimensional transformation used by SURFACE is taken from the 4 by 4 array !P.T.

The surface represented by the two-dimensional array is first rotated, AZ (see below) degrees about the Z axis, then by AX degrees about the X axis, tilting the surface towards the viewer ($AX > 0$), or away from the viewer.

The AX and AZ keyword parameters default to +30 degrees if omitted and !P.T3D is 0.

The three-dimensional to two-dimensional transformation represented by AX and AZ, can be saved in !P.T by including the SAVE keyword.

AZ

This keyword specifies the counterclockwise angle of rotation about the Z axis. This keyword is effective only if !P.T3D is not set. The order of rotation is AZ first, then AX.

BOTTOM

The color index used to draw the bottom surface. If not specified, the bottom is drawn with the same color as the top.

HORIZONTAL

A keyword flag which if set causes SURFACE to only draw lines across the plot perpendicular to the line of sight. The default is for SURFACE to draw both across the plot and from front to back.

LEGO

Set this keyword to produce stacked histogram-style plots. Each data value is rendered as a box covering the XY extent of the cell and with a height proportional to the Z value.

If the X and Y arguments are specified, only N_x-1 columns and N_y-1 rows are drawn. (This means that the last row and column of array data are not displayed.) The rectangular area covered by $Z[i, j]$ is given by $X[i]$, $X[i+1]$, $Y[j]$, and $Y[j+1]$.

LOWER_ONLY

Set this keyword to draw only the lower surface of the object. By default, both surfaces are drawn.

MAX_VALUE

The maximum value to be plotted. If this keyword is present, data values greater than the value of MAX_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

MIN_VALUE

The minimum value to be plotted. If this keyword is present, data values less than the value of MIN_VALUE are treated as missing and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. (See [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

SAVE

Set this keyword to save the 3-D to 2-D transformation matrix established by SURFACE in the system variable field !P.T. Use this keyword when combining the output of SURFACE with additional output from other routines in the same plot.

When used with `AXIS`, the `SAVE` keyword parameter saves the scaling parameters established by the call in the appropriate axis system variable, `!X`, `!Y`, or `!Z`. This causes subsequent overplots to be scaled to the new axis.

For example, to display a two-dimensional array using `SURFACE`, and to then superimpose contours over the surface (this example assumes that `!P.T3D` is zero, its default value.), enter the following commands:

```
; Make a surface plot and save the transformation:
SURFACE, Z, /SAVE

; Make contours, don't erase, use the 3-D to 2-D transform placed
; in !P.T by SURFACE:
CONTOUR, Z, /NOERASE, /T3D
```

To display a surface and to then display a flat contour plot, registered above the surface:

```
; Make the surface, save transform:
SURFACE, Z, /SAVE

; Now display a flat contour plot, at the maximum Z value
; (normalized coordinates):
CONTOUR, Z, /NOERASE, /T3D, ZVALUE=1.0
```

You can display the contour plot below the surface with by using a `ZVALUE` of 0.0.

SHADES

This keyword allows user-specified coloring of the mesh surfaces. Set this keyword to an array that specifies the color index of the lines emanating from each data point toward the top and right.

Warning

When using the `SHADES` keyword on True Color devices, we recommend that decomposed color support be turned off, by setting `DEVICE, DECOMPOSED=0`. See [“DEVICE”](#) on page 537 and [“DECOMPOSED”](#) on page 4257.

SKIRT

This keyword represents a `Z`-value at which to draw a skirt around the array. The `Z` value is expressed in data units. The default is no skirt.

If the skirt is drawn, each point on the four edges of the surface is connected to a point on the skirt which has the given `Z` value, and the same `X` and `Y` values as the edge point. In addition, each point on the skirt is connected to its neighbor.

UPPER_ONLY

Set this keyword to draw only the upper surface of the object. By default, both surfaces are drawn.

XLOG

Set this keyword to specify a logarithmic X axis.

YLOG

Set this keyword to specify a logarithmic Y axis.

ZAXIS

This keyword specifies the placement of the Z axis for the SURFACE plot.

By default, SURFACE draws the Z axis at the upper left corner of the axis box. To suppress the Z axis, use ZAXIS=-1 in the call. The position of the Z axis is determined from the value of ZAXIS as follows: 1 = lower right, 2 = lower left, 3 = upper left, and 4 = upper right.

ZLOG

Set this keyword to specify a logarithmic Z axis.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#), for the description of graphics and plotting keywords not listed above. [BACKGROUND](#), [CHARSIZE](#), [CHARTHICK](#), [CLIP](#), [COLOR](#), [DATA](#), [DEVICE](#), [FONT](#), [LINESTYLE](#), [NOCLIP](#), [NODATA](#), [NOERASE](#), [NORMAL](#), [POSITION](#), [SUBTITLE](#), [T3D](#), [THICK](#), [TICKLEN](#), [TITLE](#), [\[XYZ\]CHARSIZE](#), [\[XYZ\]GRIDSTYLE](#), [\[XYZ\]MARGIN](#), [\[XYZ\]MINOR](#), [\[XYZ\]RANGE](#), [\[XYZ\]STYLE](#), [\[XYZ\]THICK](#), [\[XYZ\]TICKFORMAT](#), [\[XYZ\]TICKINTERVAL](#), [\[XYZ\]TICKLAYOUT](#), [\[XYZ\]TICKLEN](#), [\[XYZ\]TICKNAME](#), [\[XYZ\]TICKS](#), [\[XYZ\]TICKUNITS](#), [\[XYZ\]TICKV](#), [\[XYZ\]TICK_GET](#), [\[XYZ\]TITLE](#), [ZVALUE](#).

Examples

```
; Create a simple dataset to display:
D = DIST(30)

; Plot a simple wire-mesh surface representation of D:
SURFACE, D
```

```
    ; Create a wire-mesh plot of D with a title and a "skirt" around  
    ; the edges of the dataset at Z=0:  
SURFACE, D, SKIRT=0.0, TITLE = 'Surface Plot', CHAR.SIZE = 2
```

Version History

Introduced: Original

See Also

[CONTOUR](#), [ISURFACE](#), [SHADE_SURF](#)

SURFR

The SURFR procedure sets up 3-D transformations. This procedure duplicates the rotation, translation, and scaling features of the SURFACE routine, but does not display any data. The resulting transformations are stored in the !P.T system variable.

This routine is written in the IDL language. Its source code can be found in the file `surfr.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SURFR [, AX=degrees] [, AZ=degrees]
```

Arguments

None.

Keywords

AX

This keyword specifies the angle of rotation about the X axis in degrees towards the viewer. The default is 30 degrees.

First rotated, AZ (see below) degrees about the Z axis, then by AX degrees about the X axis, tilting the surface towards the viewer ($AX > 0$), or away from the viewer.

AZ

This keyword specifies the counterclockwise angle of rotation about the Z axis. The default is 30 degrees.

Version History

Introduced: Original

See Also

[SCALE3](#), [SCALE3D](#), [T3D](#)

SVDC

The SVDC procedure computes the Singular Value Decomposition (SVD) of a square ($n \times n$) or non-square ($n \times m$) array as the product of orthogonal and diagonal arrays. SVD is a very powerful tool for the solution of linear systems, and is often used when a solution cannot be determined by other numerical algorithms.

The SVD of an ($n \times m$) non-square array A is computed as the product of an ($n \times m$) column orthogonal array U , an ($n \times n$) diagonal array SV , composed of the singular values, and the transpose of an ($n \times n$) orthogonal array V : $A = U \, SV \, V^T$

SVDC is based on the routine `svdcmp` described in section 2.6 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the `LA_SVD` procedure.

Syntax

SVDC, A , W , U , V [, /COLUMN] [, /DOUBLE] [, ITMAX=*value*]

Arguments

A

The square ($n \times n$) or non-square ($n \times m$) single- or double-precision floating-point array to decompose.

W

On output, W is an n -element output vector containing the “singular values.”

U

On output, U is an n -column, m -row orthogonal array used in the decomposition of A .

V

On output, V is an n -column, n -row orthogonal array used in the decomposition of A .

Keywords

COLUMN

Set this keyword if the input array *A* is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

ITMAX

Set this keyword to specify the maximum number of iterations. The default value is 30.

Examples

To find the singular values of an array *A*:

```

; Define the array A:
A = [[1.0, 2.0, -1.0, 2.5], $
      [1.5, 3.3, -0.5, 2.0], $
      [3.1, 0.7, 2.2, 0.0], $
      [0.0, 0.3, -2.0, 5.3], $
      [2.1, 1.0, 4.3, 2.2], $
      [0.0, 5.5, 3.8, 0.2]]

; Compute the Singular Value Decomposition:
SVDC, A, W, U, V

; Print the singular values:
PRINT, W

```

IDL prints:

```

8.81973      2.65502      4.30598      6.84484

```

To verify the decomposition, use the relationship $A = U \## SV \## \text{TRANSPPOSE}(V)$, where *SV* is a diagonal array created from the output vector *W*:

```

sv = FLTARR(4, 4)
FOR K = 0, 3 DO sv[K,K] = W[K]
result = U ## sv ## TRANSPPOSE(V)
PRINT, result

```

IDL prints:

1.00000	2.00000	-1.00000	2.50000
1.50000	3.30000	-0.500001	2.00000
3.10000	0.700000	2.20000	0.00000
2.23517e-08	0.300000	-2.00000	5.30000
2.10000	0.999999	4.30000	2.20000
-3.91155e-07	5.50000	3.80000	0.200000

This is the input array, to within machine precision.

Version History

Introduced: 4.0

See Also

[CHOLDC](#), [LA_SVD](#), [LUDC](#), [SVSOL](#), “[Linear Systems](#)” in Chapter 22 of the *Using IDL* manual.

SVDFIT

The SVDFIT function performs a least squares fit with optional error estimates. Either a user-supplied function written in the IDL language or a built-in polynomial can be used to fit the data.

SVDFIT is based on the routine `svdfit` described in section 15.4 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = SVDFIT( X, Y [, M] [, A=vector] [, CHISQ=variable] [, COVAR=variable]
  [, /DOUBLE] [, FUNCTION_NAME=string] [, /LEGENDRE]
  [, MEASURE_ERRORS=vector] [, SIGMA=variable]
  [, SING_VALUES=variable] [, SINGULAR=variable] [, STATUS=variable]
  [, TOL=value] [, VARIANCE=variable] [, YFIT=variable] )
```

Return Value

Returns a vector of coefficients.

Arguments

X

An n-element vector of independent variables.

Y

A vector of dependent variables, the same length as X.

M

The number of coefficients in the fitting function. For polynomials, *M* is equal to the degree of the polynomial + 1. If the *M* argument is not specified, you must supply initial coefficient estimates using the A keyword. In this case, *M* is set equal to the number of elements of the array specified by the A keyword.

Keywords

A

This keyword is both an input and output keyword. Set this keyword equal to a variable containing a vector of initial estimates for the fitted function parameters. On exit, SVDFIT returns in this variable a vector of coefficients that are improvements of the initial estimates. If A is supplied, the *M* argument will be set equal to the number of elements in the vector specified by A.

CHISQ

Set this keyword equal to a named variable that will contain the value of the unreduced chi-square goodness-of-fit.

COVAR

Set this keyword equal to a named variable that will contain the covariance matrix of the fitted coefficients.

Note

The COVAR matrix depends only upon the independent variable X and (optionally) the MEASURE_ERRORS. The values do not depend upon Y. See section 15.4 of *Numerical Recipes in C* (Second Edition) for details.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FUNCTION_NAME

Set this keyword equal to a string containing the name of a user-supplied IDL basis function with *M* coefficients. If this keyword is omitted, and the LEGENDRE keyword is not set, IDL assumes that the IDL procedure SVDFUNCT, found in the file `svdfunct.pro`, located in the `lib` subdirectory of the IDL distribution, is to be used. SVDFUNCT uses the basis functions for the fitting polynomial:

$$y = \sum_{i=0}^M A(i)x^i$$

The function to be fit must be written as an IDL function and compiled prior to calling SVDFIT. The function must accept values of X (a scalar), and M (a scalar). It must return an M -element vector containing the basis functions.

See the “[Examples](#)” section below for an example function.

LEGENDRE

Set this keyword to use Legendre polynomials instead of the function specified by the FUNCTION_NAME keyword. If the LEGENDRE keyword is set, the IDL uses the function SVDLEG found in the file `svdleg.pro`, located in the `lib` subdirectory of the IDL distribution.

MEASURE_ERRORS

Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y .

Note

For Gaussian errors (e.g., instrumental uncertainties), MEASURE_ERRORS should be set to the standard deviations of each point in Y . For Poisson or statistical weighting, MEASURE_ERRORS should be set to $\text{SQRT}(Y)$.

SIGMA

Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters.

Note

If MEASURE_ERRORS is omitted, then you are assuming that the polynomial (or your user-supplied model) is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number of coefficients. See section 15.2 of *Numerical Recipes in C* (Second Edition) for details.

SING_VALUES

Set this keyword to a named variable in which to return the singular values from the SVD. Singular values which have been removed will be set to zero.

SINGULAR

Set this keyword equal to a named variable that will contain the number of singular values returned. This value should be 0. If not, the basis functions do not accurately characterize the data.

STATUS

Set this keyword to a named variable that will contain the status of the computation. Possible values are:

- STATUS = 0: The computation was successful.
- STATUS > 0: Singular values were found and were removed. STATUS contains the number of singular values.

Note

If STATUS is not specified, any error messages will be output to the screen.

TOL

Set this keyword to the tolerance used when removing singular values. The default is 10^{-5} for single precision, and 2×10^{-12} for double precision (these defaults are approximately 100 and 10000 times the machine precisions for single and double precision, respectively).

Setting TOL to a larger value may remove coefficients that do not contribute to the solution, which may reduce the errors on the remaining coefficients.

VARIANCE

Set this keyword equal to a named variable that will contain the variance (sigma squared) of each coefficient M .

WEIGHTS

The WEIGHTS keyword is obsolete and has been replaced by the [MEASURE_ERRORS](#) keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $1/\text{WEIGHTS}[i]$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error is represented as simply $\text{MEASURE_ERRORS}[i]$.

YFIT

Set this keyword equal to a named variable that will contain the vector of calculated Y values.

Examples

This example fits a function of the following form:

$$F(x) = A(0) + A(1) \sin\left(\frac{2x}{x}\right) + A(2) \cos(4x)^2$$

First, create the function in IDL, then create a procedure to perform the fit. Create the following file called `example_svdfit.pro`:

```

PRO example_svdfit

; Provide an array of coefficients:
C = [7.77, 8.88, -9.99]
X = FINDGEN(100)/15.0 + 0.1
Y = C[0] + C[1] * SIN(2*X)/X + C[2] * COS(4.*X)^2.

; Set uncertainties to 5%:
measure_errors = 0.05 * Y

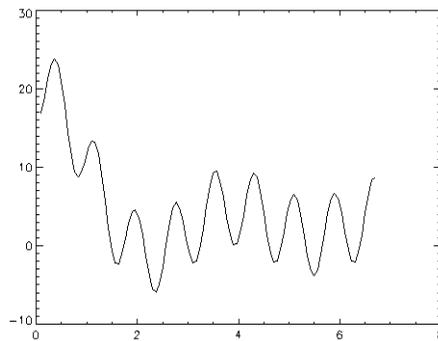
; Provide an initial guess:
A=[1,1,1]
result_a = SVDFIT(X, Y, A=A, MEASURE_ERRORS=measure_errors, $
    FUNCTION_NAME='myfuncnt', SIGMA=SIGMA, YFIT=YFIT)

; Plot the results:
PLOT, X, YFIT
FOR I = 0, N_ELEMENTS(A)-1 DO $
    PRINT, I, result_a[I], SIGMA[I], C[I], $
    FORMAT = $
    '(' result_a ( ",I1," ) = ",F7.4," +- ",F7.4," VS. ",F7.4)'
END

FUNCTION myfuncnt, X ,M
    RETURN, [ [1.0], [SIN(2*X)/X], [COS(4.*X)^2.] ]
END

```

Place the file `example_svdfit.pro` in a directory in the IDL search path, and enter `example_svdfit` at the command prompt to create the plot.



In addition to creating the above plot, IDL prints:

```
result_a ( 0 ) = 7.7700 +- 0.0390 VS. 7.7700
result_a ( 1 ) = 8.8800 +- 0.0468 VS. 8.8800
result_a ( 2 ) = -9.9900 +- 0.0506 VS. -9.9900
```

Version History

Introduced: Original

SING_VALUES, STATUS, and TOL keywords added: 5.6

See Also

[CURVEFIT](#), [GAUSSFIT](#), [LINFIT](#), [LMFIT](#), [POLY_FIT](#), [REGRESS](#), [SFIT](#)

SVSOL

The SVSOL function uses “back-substitution” to solve a set of simultaneous linear equations $Ax = b$, given the U , W , and V arrays returned by the SVDC procedure. None of the input arguments are modified, making it possible to call SVSOL multiple times with different right hand vectors, B .

SVSOL is based on the routine `svbksb` described in section 2.6 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = SVSOL( U, W, V, B [, /COLUMN] [, /DOUBLE] )
```

Return Value

Returns the solution to the linear system using decomposition and back substitution.

Arguments

U

An n -column, m -row orthogonal array used in the decomposition of A . Normally, U is returned from the SVDC procedure.

W

An n -element vector containing “singular values.” Normally, W is returned from the SVDC procedure. Small values (close to machine floating-point precision) should be set to zero prior to calling SVSOL.

V

An n -column, n -row orthogonal array used in the decomposition of A . Normally, V is returned from the SVDC procedure.

B

An m -element vector containing the right hand side of the linear system $Ax = b$.

Keywords

COLUMN

Set this keyword if the input arrays U and V are in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To solve the linear system $Ax = b$ using Singular-value decomposition and back substitution, begin with an array A which serves as the coefficient array:

```

; Define the array A:
A = [[1.0, 2.0, -1.0, 2.5], $
      [1.5, 3.3, -0.5, 2.0], $
      [3.1, 0.7, 2.2, 0.0], $
      [0.0, 0.3, -2.0, 5.3], $
      [2.1, 1.0, 4.3, 2.2], $
      [0.0, 5.5, 3.8, 0.2]]

; Define the right-hand side vector B:
B = [0.0, 1.0, 5.3, -2.0, 6.3, 3.8]

; Decompose A:
SVDC, A, W, U, V

; Compute the solution and print the result:
PRINT, SVSOL(U, W, V, B)

```

IDL prints:

```
1.00095    0.00881170    0.984176    -0.0100954
```

This is the correct solution.

Version History

Introduced: 4.0

See Also

[CRAMER](#), [GS_ITER](#), [LU_COMPLEX](#), [CHOLSOL](#), [LUSOL](#), [SVDC](#), [TRISOL](#)

SWAP_ENDIAN

The SWAP_ENDIAN function reverses the byte ordering of arbitrary scalars, arrays or structures. It can make “big endian” number “little endian” and vice-versa.

Note

The BYTEORDER procedure can be used to reverse the byte ordering of *scalars and arrays* (SWAP_ENDIAN also allows structures).

This routine is written in the IDL language. Its source code can be found in the file `swap_endian.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = SWAP_ENDIAN(Variable [, /SWAP_IF_BIG_ENDIAN]  
                    [, /SWAP_IF_LITTLE_ENDIAN])
```

Return Value

SWAP_ENDIAN returns values of the same type and structure as the input value, with the pertinent bytes reversed.

Arguments

Variable

The named variable—scalar, array, or structure—to be swapped.

Keywords

SWAP_IF_BIG_ENDIAN

If this keyword is set, the swap request will only be performed if the platform running IDL uses “big endian” byte ordering. On little endian machines, the SWAP_ENDIAN request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

If this keyword is set, the swap request will only be performed if the platform running IDL uses “little endian” byte ordering. On big endian machines, the SWAP_ENDIAN

request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

Examples

```
; Reverse the byte order of A:  
A = SWAP_ENDIAN(A)
```

Version History

Introduced: Pre 4.0

SWAP_IF_BIG_ENDIAN and SWAP_IF_LITTLE_ENDIAN keywords added: 5.6

See Also

[BYTEORDER](#), [SWAP_ENDIAN_INPLACE](#)

SWAP_ENDIAN_INPLACE

The `SWAP_ENDIAN_INPLACE` procedure reverses the byte ordering of arbitrary scalars, arrays or structures. It can make “big endian” number “little endian” and vice-versa.

Note

The `BYTEORDER` procedure can be used to reverse the byte ordering of *scalars and arrays* (`SWAP_ENDIAN_INPLACE` also allows structures).

`SWAP_ENDIAN_INPLACE` differs from the `SWAP_ENDIAN` function in that it alters the input data in place rather than making a copy as does `SWAP_ENDIAN`. `SWAP_ENDIAN_INPLACE` can therefore be more efficient, if a copy of the data is not needed. The pertinent bytes in the input variable are reversed.

This routine is written in the IDL language. Its source code can be found in the file `swap_endian_inplace.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
SWAP_ENDIAN_INPLACE, Variable [, /SWAP_IF_BIG_ENDIAN]  
                        [, /SWAP_IF_LITTLE_ENDIAN]
```

Arguments

Variable

The named variable—scalar, array, or structure—to be swapped.

Keywords

SWAP_IF_BIG_ENDIAN

If this keyword is set, the swap request will only be performed if the platform running IDL uses “big endian” byte ordering. On little endian machines, the `SWAP_ENDIAN_INPLACE` request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

If this keyword is set, the swap request will only be performed if the platform running IDL uses “little endian” byte ordering. On big endian machines, the `SWAP_ENDIAN_INPLACE` request quietly returns without doing anything. Note that this keyword does not refer to the byte ordering of the input data, but to the computer hardware.

Examples

Reverse the byte order of A:

```
SWAP_ENDIAN_INPLACE, A
```

Version History

Introduced: 5.6

See Also

[BYTEORDER](#), [SWAP_ENDIAN](#)

SWITCH

The SWITCH statement is used to select one statement for execution from multiple choices, depending upon the value of the expression following the word SWITCH.

Each statement that is part of a SWITCH statement is preceded by an expression that is compared to the value of the SWITCH expression. SWITCH executes by comparing the SWITCH expression with each selector expression in the order written. If a match is found, program execution jumps to that statement and execution continues from that point. Whereas CASE executes at most one statement within the CASE block, SWITCH executes the first matching statement and any following statements in the SWITCH block. Once a match is found in the SWITCH block, execution falls through to any remaining statements. For this reason, the BREAK statement is commonly used within SWITCH statements to force an immediate exit from the SWITCH block.

The ELSE clause of the SWITCH statement is optional. If included, it matches any selector expression, causing its code to be executed. For this reason, it is usually written as the last clause in the switch statement. The ELSE statement is executed only if none of the preceding statement expressions match. If an ELSE clause is not included and none of the values match the selector, program execution continues immediately below the SWITCH without executing any of the SWITCH statements.

SWITCH is similar to the CASE statement. For more information on using SWITCH and other IDL program control statements, as well as the differences between SWITCH and CASE, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

```
SWITCH expression OF
    expression: statement
    ...
    expression: statement
ELSE: statement
ENDSWITCH
```

Examples

This example illustrates how, unlike CASE, SWITCH executes the first matching statement and any following statements in the SWITCH block:

```
x=2

SWITCH x OF
  1: PRINT, 'one'
  2: PRINT, 'two'
  3: PRINT, 'three'
  4: PRINT, 'four'
ENDSWITCH
```

IDL Prints:

```
two
three
four
```

Version History

Introduced: 5.4

See Also

[CASE](#)

SYSTIME

The SYSTIME function returns the current time as either a date/time string, as the number of seconds elapsed since 1 January 1970, or as a Julian date/time value.

Syntax

String = SYSTIME([0 [, *ElapsedSeconds*]] [, /UTC])

or

Seconds = SYSTIME(1 | /SECONDS)

or

Julian = SYSTIME(/JULIAN [, /UTC])

Return Value

Returns the specified time.

Arguments

SecondsFlag

If *SecondsFlag* is present and nonzero, the number of seconds elapsed since 1 January 1970 UTC is returned as a double-precision, floating-point value.

Otherwise, if *SecondsFlag* is not present or zero, a scalar string containing the date/time is returned in standard 24-character system format as follows:

```
DOW MON DD HH:MM:SS YEAR
```

where DOW is the day of the week, MON is the month, DD is the day of the month, HH is the hour, MM is the minute, SS is the second, and YEAR is the year. By default, the date/time string is adjusted for the local time zone; use the UTC keyword to override this default.

Note

If the JULIAN or SECONDS keyword is set, the *SecondsFlag* argument is ignored.

ElapsedSeconds

If the *SecondsFlag* argument is zero, the *ElapsedSeconds* argument may be set to the number of seconds past 1 January 1970 UTC. In this case, SYSTIME returns the

corresponding date/time string (rather than the string for the current time). The returned date/time string is adjusted for the local time zone, unless the UTC keyword is set. If this argument is present, the JULIAN keyword is not allowed.

Keywords

JULIAN

Set this keyword to specify that the current time is to be returned as a double precision floating value containing the current Julian date/time. By default, the current time is adjusted for the local time zone; use the UTC keyword to override this default. This keyword is not allowed if the *ElapsedSeconds* argument is present.

Note

If the JULIAN keyword is set, a small offset is added to the returned Julian date to eliminate roundoff errors when calculating the day fraction from hours, minutes, and seconds. This offset is given by the larger of EPS and EPS*Julian, where Julian is the integer portion of the Julian date, and EPS is the EPS field from MACHAR. For typical Julian dates, this offset is approximately 6×10^{-10} (which corresponds to 5×10^{-5} seconds). This offset ensures that if the Julian date is converted back to hour, minute, and second, then the hour, minute, and second will have the same integer values as were originally input.

SECONDS

Set this keyword to specify that the current time is to be returned as the number of seconds elapsed since 1 January 1970 UTC. This option is equivalent to setting the *SecondsFlag* argument to a non-zero value.

UTC

Set this keyword to specify that the value returned by SYSTIME is to be returned in Universal Time Coordinated (UTC) rather than being adjusted for the current time zone. UTC time is defined as Greenwich Mean Time updated with leap seconds.

Examples

Print today's date as a string:

```
PRINT, SYSTIME()
```

Print today's date as a string in UTC (rather than local time zone):

```
PRINT, SYSTIME(/UTC)
```

Print today's date as a Julian date/time value in UTC:

```
PRINT, SYSTIME(/JULIAN, /UTC), FORMAT='(f12.2)'
```

Compute the seconds elapsed since 1 January 1970 UTC:

```
seconds = SYSTIME(1) ; or seconds = SYSTIME(/SECONDS)
```

Verify that the seconds from the previous example are correct:

```
PRINT, SYSTIME(0, seconds)
```

Print the day of the week:

```
PRINT, STRMID(SYSTIME(0), 0, 3)
```

Compute the time required to perform a 16,384 point FFT:

```
T = SYSTIME(1)
A = FFT(FINDGEN(16384), -1)
PRINT, SYSTIME(1) - T, 'Seconds'
```

Version History

Introduced: Original

See Also

[CALDAT](#), [CALENDAR](#), [JULDAY](#), [TIMEGEN](#)

T_CVF

The `T_CVF` function computes the cutoff value V in a Student's t distribution with Df degrees of freedom such that the probability that a random variable X is greater than V is equal to a user-supplied probability P .

Note

`T_CVF` computes the cutoff value using the one-tailed probability. The cutoff value for the two-tailed probability, which is the probability that the absolute value of X is greater than V , can be computed as `T_CVF(P/2, Df)`.

This routine is written in the IDL language. Its source code can be found in the file `t_cvf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = `T_CVF(P, Df)`

Return Value

Returns the cutoff value.

Arguments

P

A non-negative single- or double-precision floating-point scalar, in the interval $[0.0, 1.0]$, that specifies the probability of occurrence or success.

Df

A positive integer, single- or double-precision floating-point scalar that specifies the number of degrees of freedom of the Student's t distribution.

Keywords

None.

Examples

Use the following command to compute the cutoff value in a Student's t distribution with five degrees of freedom such that the probability that a random variable X is greater than the cutoff value is 0.025.

```
result = T_CVF(0.025, 5)
PRINT, result
```

IDL prints:

```
2.57058
```

Version History

Introduced: 4.0

See Also

[CHISQR_CVF](#), [F_CVF](#), [GAUSS_CVF](#), [T_PDF](#)

T_PDF

The T_PDF function computes the probability P that, in a Student's t distribution with Df degrees of freedom, a random variable X is less than or equal to a user-specified cutoff value V .

Note

T_PDF computes the one-tailed probability. The two-tailed probability, which is the probability that the absolute value of X is less than or equal to V , can be computed as $1 - 2 \times (1 - \text{T_PDF}(V, Df))$.

This routine is written in the IDL language. Its source code can be found in the file `t_pdf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = T_PDF(*V*, *Df*)

Return Value

If both arguments are scalar, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of V and Df , returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

If any of the arguments are double-precision, the result is double-precision, otherwise the result is single-precision.

Arguments

V

A scalar or array that specifies the cutoff value(s).

Df

A scalar or array that specifies the number of degrees of freedom of the Student's t distribution.

Keywords

None.

Examples

Use the following command to compute the probability that a random variable X , from the Student's t distribution with 15 degrees of freedom, is less than or equal to 0.691:

```
PRINT, T_PDF(0.691, 15)
```

IDL prints:

```
0.749940
```

Version History

Introduced: 4.0

See Also

[BINOMIAL](#), [CHISQR_PDF](#), [F_PDF](#), [GAUSS_PDF](#), [T_CVF](#)

T3D

The T3D procedure implements three-dimensional transforms.

This routine accumulates one or more sequences of translation, scaling, rotation, perspective, and oblique transformations and stores the result in !P.T, the 3D transformation system variable. All the IDL graphic routines use this (4,4) matrix for output. Note that !P.T3D is *not* set, so for the transformations to have effect you must set !P.T3D = 1 (or set the T3D keyword in subsequent calls to graphics routines).

This procedure is based on that of Foley & Van Dam, *Fundamentals of Interactive Computer Graphics*, Chapter 8, “Viewing in Three Dimensions”. The matrix notation is reversed from the normal IDL sense, i.e., here, the first subscript is the column, the second is the row, in order to conform with this reference.

A right-handed system is used. Positive rotations are counterclockwise when looking from a positive axis position towards the origin.

This routine is written in the IDL language. Its source code can be found in the file `t3d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
T3D [, Array | , /RESET] [, MATRIX=variable] [, OBLIQUE=vector]
    [, PERSPECTIVE=p{eye at (0,0,p)}] [, ROTATE=[x, y, z]] [, SCALE=[x, y, z]]
    [, TRANSLATE=[x, y, z]] [, /XYEXCH | , /XZEXCH | , /YZEXCH]
```

Arguments

Array

An optional 4 x 4 matrix used as the starting transformation. If *Array* is missing, the current !P.T transformation is used. *Array* is ignored if /RESET is set.

Keywords

The transformation specified by each keyword is performed in the order of their descriptions below (e.g., if both TRANSLATE and SCALE are specified, the translation is done first).

MATRIX

Set this keyword to a named variable that will contain the result. If this keyword is specified, !P.T is not modified.

OBLIQUE

A two-element vector of oblique projection parameters. Points are projected onto the XY plane at Z=0 as follows:

$$\begin{aligned}x' &= x + z(d * \text{COS}(a)) \\y' &= y + z(d * \text{SIN}(a))\end{aligned}$$

where OBLIQUE[0] = d and OBLIQUE[1] = a.

PERSPECTIVE

Perspective transformation. This parameter is a scalar (p) that indicates the Z distance of the center of the projection. Objects are projected into the XY plane at Z=0, and the “eye” is at point (0,0,p).

RESET

Set this keyword to reset the transformation to the default identity matrix.

ROTATE

A three-element vector of the rotations, in DEGREES, about the X, Y, and Z axes. Rotations are performed in the order of X, Y, and then Z.

SCALE

A three-element vector of scale factors for the X, Y, and Z axes.

TRANSLATE

A three-element vector of the translations in the X, Y, and Z directions.

XYEXCH

Set this keyword to exchange the X and Y axes.

XZEXCH

Set this keyword to exchange the X and Z axes.

YZEXCH

Set this keyword to exchange the Y and Z axes.

Examples

To reset the transformation, rotate 30 degs about the X axis and do perspective transformation with the center of the projection at Z = -1, X=0, and Y=0, enter:

```
T3D, /RESET, ROT = [ 30,0,0], PERS = 1.
```

Transformations may be cascaded, for example:

```
T3D, /RESET, TRANS = [-.5,-.5,0], ROT = [0,0,45]  
T3D, TRANS = [.5,.5,0]
```

The first command resets, translates the point (.5,.5,0) to the center of the viewport, then rotates 45 degrees counterclockwise about the Z axis. The second call to T3D moves the origin back to the center of the viewport.

Version History

Introduced: Original

See Also

[SCALE3](#), [SCALE3D](#), [SURFR](#)

TAG_NAMES

The TAG_NAMES function returns a string array containing the names of the tags in a structure expression. It can also be used to determine the expression's structure name (if the structure has a name).

Syntax

Result = TAG_NAMES(*Expression* [, /STRUCTURE_NAME])

Return Value

Returns structure tag names or the expression's structure name.

Arguments

Expression

The structure expression for which the tag names are returned. This argument must be of structure type. TAG_NAMES does not search for tags recursively, so if *Expression* is a structure containing nested structures, only the names of tags in the outermost structure are returned.

Keywords

STRUCTURE_NAME

Set this keyword to return a scalar string that contains the name of the structure instead of the names of the tags in the structure. If the structure is “anonymous”, a null string is returned.

Examples

Print the names of the tags in the system variable !P by entering:

```
PRINT, TAG_NAMES(!P)
```

IDL prints:

```
BACKGROUND CHARSIZE CHARTHICK CLIP COLOR FONT LINSTYLE MULTI  
NOCLIP NOERASE NSUM POSITION PSYM REGION SUBTITLE SYMSIZE T  
T3D THICK TITLE TICKLEN CHANNEL
```

Print the name of the structure in the system variable !P:

```
PRINT, TAG_NAMES(!P, /STRUCTURE_NAME)
```

IDL prints:

```
!PLT
```

Version History

Introduced: Original

See Also

[CREATE_STRUCT](#), [N_TAGS](#)

TAN

The TAN function computes the tangent of X .

Syntax

$Result = \text{TAN}(X)$

Return Value

Returns the tangent of the specified angle.

Arguments

X

The angle for which the tangent is desired, specified in radians. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. If X is an array, the result has the same structure, with each element containing the tangent of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

```
; Find the tangent of 0.5 radians and store the result in
; the variable T:
T = TAN(0.5)
```

Version History

Introduced: Original

See Also

[ATAN](#), [TANH](#)

TANH

The TANH function returns the hyperbolic tangent of X .

Syntax

Result = TANH(X)

Return Value

Returns the single- or double-precision hyperbolic tangent.

Arguments

X

The value for which the hyperbolic tangent is desired, specified in radians. If X is double-precision floating or complex, the result is of the same type. All other types are converted to single-precision floating-point and yield floating-point results. TANH is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

If X is an array, the result has the same structure, with each element containing the hyperbolic tangent of the corresponding element of X .

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

```
; Find the hyperbolic tangent of 1 radian and print the result:  
PRINT, TANH(1)
```

```
; Plot the hyperbolic tangent from -5 to +5 with an increment  
; of 0.1:  
PLOT, TANH(FINDGEN(101)/10. - 5)
```

Version History

Introduced: Original

See Also

[ATAN](#), [TAN](#)

TEK_COLOR

The TEK_COLOR procedure loads a 32-color colortable similar to the default Tektronix 4115 colortable. This colortable is useful because of its distinct colors.

By default, this palette consists of 32 colors. The first 9 colors are: Index 0=black, 1=white, 2=red, 3=green, 4=blue, 5=cyan, 6=magenta, 8=orange.

Syntax

```
TEK_COLOR [, Start_Index, Colors]
```

Arguments

Start_Index

An optional starting index for the palette. The default is 0. If this argument is included, the colors are loaded into the current colortable starting at the specified index.

Colors

The number of colors to load. The default is 32, which is also the maximum.

Keywords

None.

Version History

Introduced: Pre 4.0

See Also

[LOADCT](#), [XLOADCT](#)

TEMPORARY

The TEMPORARY function returns a temporary copy of a variable, and sets the original variable to “undefined”. This function can be used to conserve memory when performing operations on large arrays, as it avoids making a new copy of results that are only temporary. In general, the TEMPORARY routine can be used to advantage whenever a variable containing an array on the left hand side of an assignment statement is also referenced on the right hand side.

Syntax

Result = TEMPORARY(*Variable*)

Return Value

Returns a copy of a specified variable.

Arguments

Variable

The variable to be referenced and deleted.

Keywords

None.

Examples

Assume the variable `A` is a large array of integers. The statement:

```
A = A + 1
```

creates a new array for the result of the addition, places the sum into the new array, assigns it to `A`, and then frees the old allocation of `A`. Total storage required is twice the size of `A`. The statement:

```
A = TEMPORARY(A) + 1
```

requires no additional space.

Note

If the operation performed on *Variable* requires that *Variable* be converted to another data type, there is no benefit to using TEMPORARY. For instance, if the

array A in the above example contained byte data rather than integer data, the array would have to be converted to integer type before the addition could be performed. In such a case, memory could not be re-used.

Version History

Introduced: Pre 4.0

See Also

[DELVAR](#)

TETRA_CLIP

The TETRA_CLIP function clips a tetrahedral mesh to an arbitrary plane in space and returns a tetrahedral mesh of the remaining portion. An auxiliary array of data may also be passed and clipped. This array can have multiple values for each vertex (the trailing array dimension must match the number of vertices in the Vertsin array).

A tetrahedral connectivity array consists of groups of four vertex index values. Each set of four index values specifies four vertices which define a single tetrahedron.

Syntax

```
Result = TETRA_CLIP ( Plane, Vertsin, Connin, Vertsout, Connout  
    [, AUXDATA_IN=array, AUXDATA_OUT=variable]  
    [, CUT_VERTS=variable] )
```

Return Value

The return value is the number of tetrahedra returned.

Arguments

Plane

Input four-element array describing the equation of the plane to be clipped to. The elements are the coefficients (a,b,c,d) of the equation $ax+by+cz+d=0$.

Vertsin

Input array of tetrahedral vertices $[3, n]$.

Connin

Input tetrahedral mesh connectivity array.

Vertsout

Output array of tetrahedral vertices $[3, n]$.

Connout

Output tetrahedral mesh connectivity array.

Keywords

AUXDATA_IN

Input array of auxiliary data. If present, these values are interpolated and returned through AUXDATA_OUT. The trailing array dimension must match the number of vertices in the Vertsin array.

AUXDATA_OUT

Set this keyword to a named variable to contain an output array of interpolated auxiliary data.

CUT_VERTS

Set this keyword to a named variable to contain an output array of vertex indices (into Vertsout) of the vertices which are considered to be 'on' the clipped surface.

Version History

Introduced: 5.5

TETRA_SURFACE

The TETRA_SURFACE function extracts a polygonal mesh as the exterior surface of a tetrahedral mesh. The output of this function is a polygonal mesh connectivity array that can be used with the input Verts array to display the outer surface of the tetrahedral mesh.

Syntax

Result = TETRA_SURFACE (*Verts*, *Connin*)

Return Value

Returns a polygonal mesh connectivity array. When used with the input vertex array, this function yields the exposed tetrahedral mesh surface.

Arguments

Verts

Array of vertices [3, *n*].

Connin

Tetrahedral connectivity array.

Keywords

None.

Version History

Introduced: 5.5

TETRA_VOLUME

The TETRA_VOLUME function computes properties of a tetrahedral mesh array. The basic property is the volume. An auxiliary data array may be supplied which specifies weights at each vertex which are interpolated through the volume during integration. Higher order moments (with respect to the X, Y, and Z axis) may be computed as well (with or without weights).

Syntax

```
Result = TETRA_VOLUME ( Verts, Conn [, AUXDATA=array]  
[, MOMENT=variable] )
```

Return Value

Returns the cumulative (weighted) volume of the tetrahedrons in the mesh.

Arguments

Verts

Array of vertices [3, *n*].

Conn

Tetrahedral connectivity array.

Keywords

AUXDATA

Array of input auxiliary data (one value per vertex). If present, these values are used to weight a vertex. The volume area integral will linearly interpolate these values. The volume integral will linearly interpolate these values within each tetrahedra. The default weight is 1.0 which results in a basic volume.

MOMENT

Set this keyword to a named variable that will contain a three-element float vector which corresponds to the first order moments computed with respect to the X, Y and Z axis. The computation is:

$$\vec{m} = \sum_{ntetras} v_i \vec{c}_i$$

where v is the (weighted) volume of the tetrahedron and c is the centroid of the tetrahedron, thus

$$\vec{m}/volume$$

yields the (weighted) centroid of the tetrahedral mesh.

Version History

Introduced: 5.5

THIN

The THIN function returns the “skeleton” of a bi-level image. The skeleton of an object in an image is a set of lines that reflect the shape of the object. The set of skeletal pixels can be considered to be the medial axis of the object. For a much more extensive discussion of skeletons and thinning algorithms, see *Algorithms for Graphics and Image Processing*, Theo Pavlidis, Computer Science Press, 1982. The THIN function is adapted from Algorithm 9.1 (the classical thinning algorithm).

On input, the bi-level image is a rectangular array in which pixels that compose the object have a nonzero value. All other pixels are zero. The result is a byte type image in which skeletal pixels are set to 2 and all other pixels are zero.

Syntax

Result = THIN(*Image* [, /NEIGHBOR_COUNT] [, /PRUNE])

Return Value

Returns the thinned, two-dimensional byte array.

Arguments

Image

The two-dimensional image (array) to be thinned.

Keywords

NEIGHBOR_COUNT

Set this keyword to select an alternate form of output. In this form, output pixel values count the number of neighbors an individual skeletal pixel has (including itself). For example, a pixel that is part of a line will have the value 3 (two neighbors and itself). Terminal pixels will have the value 2, while isolated pixels have the value 1.

PRUNE

If the PRUNE keyword is set, pixels with single neighbors are removed iteratively until only pixels with 2 or more neighbors exist. This effectively removes (or “prunes”) skeleton branches, leaving only closed paths.

Examples

The following commands display the “thinned” edges of a Sobel filtered image:

```
; Open a file for reading:
OPENR, 1, FILEPATH('people.dat', SUBDIR = ['examples', 'data'])

; Create a byte array in which to store the image:
A = BYTARR(192, 192)

; Read first 192 by 192 image:
READU, 1, A

; Close the file:
CLOSE, 1

; Display the image:
TV, A, 0

; Apply the Sobel filter, threshold the image at value 75, and
; display the thinned edges:
TVSCL, THIN(SOBEL(A) GT 75), 1
```

Version History

Introduced: Pre 4.0

See Also

[ROBERTS](#), [SOBEL](#)

THREED

The THREED procedure plots a 2-D array as a pseudo 3-D plot. The orientation of the data is fixed. This routine is written in the IDL language. Its source code can be found in the file `threed.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
THREED, A [, Sp] [, TITLE=string] [, XTITLE=string] [, YTITLE=string]
```

Arguments

A

The two-dimensional array to plot.

Sp

The spacing between plot lines. If *Sp* is omitted, the spacing is set to: $(\text{MAX}(A) - \text{MIN}(A)) / \text{ROWS}$. If *Sp* is negative, hidden lines are not removed.

Keywords

TITLE

Set this keyword to the main plot title.

XTITLE

Set this keyword to the X axis title.

YTITLE

Set this keyword to the Y axis title.

Examples

```
; Create a 2-D dataset:
A = -SHIFT(DIST(30), 15, 15)
; Make a THREED plot:
THREED, A
; Compare to SURFACE:
SURFACE, A
```

Version History

Introduced: Original

See Also

[SURFACE](#)

TIME_TEST2

The TIME_TEST2 procedure is a general-purpose IDL benchmark program that performs approximately 20 common operations and prints the time required.

This routine is written in the IDL language. Its source code can be found in the file `time_test.pro` in the `lib` subdirectory of the IDL distribution. This file also contains the procedure GRAPHICS_TIMES, used to time graphical operations.

Syntax

```
TIME_TEST2 [, Filename]
```

Arguments

Filename

An optional string that contains the name of output file for the results of the time test.

Keywords

None.

Examples

```
; Run the computational tests:  
TIME_TEST2  
  
; Run the graphics tests. Note that TIME_TEST2 must be compiled  
; before GRAPHICS_TIMES will run:  
GRAPHICS_TIMES
```

Version History

Introduced: 4.0

See Also

[SYSTIME](#)

TIMEGEN

The TIMEGEN function returns an array, with specified dimensions, of double-precision floating-point values that represent times in terms of Julian dates.

The Julian date is the number of days elapsed since Jan. 1, 4713 B.C.E., plus the time expressed as a day fraction. Following the astronomical convention, the day is defined to start at 12 PM (noon). Julian date 0.0d is therefore Jan. 1, 4713 B.C.E. at 12:00:00.

The first value of the returned array corresponds to a Julian date start time, and each subsequent value corresponds to the next Julian date in the sequence. The sequence is determined by specifying the time unit (such as months or seconds) and the step size, or spacing, between the units. You can also construct more complicated arrays by including smaller time units within each major time interval.

A small offset is added to each Julian date to eliminate roundoff errors when calculating the day fraction from the hour, minute, second. This offset is given by the larger of EPS and EPS*Julian, where Julian is the integer portion of the Julian date and EPS is the double-precision floating-point precision parameter from [MACHAR](#). For typical Julian dates the offset is approximately 6×10^{-10} (which corresponds to 5×10^{-5} seconds). This offset ensures that when the Julian date is converted back to the hour, minute, and second, the hour, minute, and second will have the same integer values.

Tip

Because of the large magnitude of the Julian date (1 Jan 2000 is Julian day 2451545), the precision of most Julian dates is limited to 1 millisecond (0.001 seconds). If you are not interested in the date itself, you can improve the precision by subtracting a large offset or setting the START keyword to zero.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000, respectively.

Syntax

```
Result = TIMEGEN( [D1,...,Dg | , FINAL=value] [, DAYS=vector]
  [, HOURS=vector] [, MINUTES=vector] [, MONTHS=vector]
  [, SECONDS=vector] [, START=value] [, STEP_SIZE=value] [, UNITS=string]
  [, YEAR=value] )
```

Return Value

Returns the specified time values.

Arguments

D_i

The dimensions of the result. The dimension parameters may be any scalar expression. Up to eight dimensions may be specified. If the dimension arguments are not integer values, IDL will truncate them to integer values before creating the new array. The dimension arguments are required unless keyword `FINAL` is set, in which case they are ignored.

Keywords

DAYS

Set this keyword to a scalar or a vector giving the day values that should be included within each month. This keyword is ignored if the `UNITS` keyword is set to “Days”, “Hours”, “Minutes”, or “Seconds”.

Note

Day values that are beyond the end of the month will be set equal to the last day for that month. For example, setting `DAY=[31]` will automatically return the last day in each month.

FINAL

Set this keyword to a double-precision value representing the Julian date/time to use as the last value in the returned array. In this case, the dimension arguments are ignored and *Result* is a one-dimensional array, with the number of elements depending upon the step size. The `FINAL` time may be less than the `START` time, in which case `STEP_SIZE` should be negative.

Note

If the step size is not an integer then the last element may not be equal to the `FINAL` time. In this case, `TIMEGEN` will return enough elements such that the last element is less than or equal to `FINAL`.

HOURS

Set this keyword to a scalar or a vector giving the hour values that should be included within each day. This keyword is ignored if UNITS is set to “Hours”, “Minutes”, or “Seconds”.

MINUTES

Set this keyword to a scalar or a vector giving the minute values that should be included within each hour. This keyword is ignored if UNITS is set to “Minutes” or “Seconds”.

MONTHS

Set this keyword to a scalar or a vector giving the month values that should be included within each year. This keyword is ignored if UNITS is set to “Months”, “Days”, “Hours”, “Minutes”, or “Seconds”.

SECONDS

Set this keyword to a scalar or a vector giving the second values that should be included within each minute. This keyword is ignored if UNITS is set to “Seconds”.

START

Set this keyword to a double-precision value representing the Julian date/time to use as the first value in the returned array. The default is 0.0d [corresponding to January 1, 4713 B.C.E. at 12 pm (noon)].

Note

If subintervals are provided by MONTHS, DAYS, HOURS, MINUTES, or SECONDS, then the first element may not be equal to the START time. In this case the first element in the returned array will be greater than or equal to START.

Tip

Other array generation routines in IDL (such as FINDGEN) do not allow you to specify a starting value because the resulting array can be added to a scalar representing the start value. For TIMEGEN it is correct to add a scalar to the array if the units are days, hours, minutes, seconds, or sub-seconds. For example:

```
MyTimes = TIMEGEN(365, UNITS="Days") + SYSTIME(/JULIAN)
```

However, if the units are months or years, the start value is necessary because the number of days in a month or year can vary depending upon the year in which they

fall (for instance, consider leap years). For example:

```
MyTimes = TIMEGEN(12, UNITS="Months", START=JULDAY(1,1,2000))
```

STEP_SIZE

Set this keyword to a scalar value representing the step size between the major intervals of the returned array. The step size may be negative. The default step size is 1. When the UNITS keyword is set to “Years” or “Months”, the STEP_SIZE value is rounded to the nearest integer.

UNITS

Set this keyword to a scalar string indicating the time units to be used for the major intervals for the generated array. Valid values include:

- “Years” or “Y”
- “Months” or “M”
- “Days” or “D”
- “Hours” or “H”
- “Minutes” or “T”
- “Seconds” or “S”

The case (upper or lower) is ignored. If this keyword is not specified, then the default for UNITS is the time unit that is larger than the largest keyword present:

Largest Keyword Present	Default UNITS
SECONDS= <i>vector</i>	“Minutes”
MINUTES= <i>vector</i>	“Hours”
HOURS= <i>vector</i>	“Days”
DAYS= <i>vector</i>	“Months”
MONTHS= <i>vector</i>	“Years”
YEAR= <i>value</i>	“Years”

Table 3-111: Defaults for the UNITS keyword

If none of the above keywords are present, the default is UNITS="Days".

YEAR

Set this keyword to a scalar giving the starting year. If YEAR is specified then the starting year from START is ignored.

Examples

Generate an array of 366 time values that are one day apart starting with January 1, 2000:

```
MyDates = TIMEGEN(366, START=JULDAY(1,1,2000))
```

Generate an array of 20 time values that are 12 hours apart starting with the current time:

```
MyTimes = TIMEGEN(20, UNITS='Hours', STEP_SIZE=12, $
    START=SYSTIME(/JULIAN))
```

Generate an array of time values that are 1 hour apart from 1 January 2000 until the current time:

```
MyTimes = TIMEGEN(START=JULDAY(1,1,2000), $
    FINAL=SYSTIME(/JULIAN), UNITS='Hours')
```

Generate an array of time values composed of seconds, minutes, and hours that start from the current hour:

```
MyTimes = TIMEGEN(60, 60, 24, $
    START=FLOOR(SYSTIME(/JULIAN)*24)/24d, UNITS='S')
```

Generate an array of 24 time values with monthly intervals, but with subintervals at 5 PM on the first and fifteenth of each month:

```
MyTimes = TIMEGEN(24, START=FLOOR(SYSTIME(/JULIAN)), $
    DAYS=[1,15], HOURS=17)
```

Version History

Introduced: 5.4

See Also

“Format Codes” in Chapter 11 of the *Building IDL Applications* manual, [CALDAT](#), [JULDAY](#), [LABEL_DATE](#), [SYSTIME](#)

TM_TEST

The TM_TEST function computes the Student's T-statistic and the probability that two sample populations X and Y have significantly different means. X and Y may be of different lengths. The default assumption is that the data is drawn from populations with the same true variance. This type of test is often referred to as the t-means test.

The T-statistic for sample populations x and y with means \bar{x} and \bar{y} is defined as:

$$T = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{\sum_{i=0}^{N-1} (x_i - \bar{x})^2 + \sum_{i=0}^{M-1} (y_i - \bar{y})^2}{(N + M - 2)} \left(\frac{1}{N} + \frac{1}{M} \right)}}$$

where $x = (x_0, x_1, x_2, \dots, x_{N-1})$ and $y = (y_0, y_1, y_2, \dots, y_{M-1})$

This routine is written in the IDL language. Its source code can be found in the file `tm_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = TM_TEST(X , Y [, /PAIRED] [, /UNEQUAL])

Return Value

The result is a two-element vector containing the T-statistic and its significance. The significance is a value in the interval [0.0, 1.0]; a small value (0.05 or 0.01) indicates that X and Y have significantly different means.

Arguments

X

An n -element integer, single-, or double-precision floating-point vector.

Y

An m -element integer, single-, or double-precision floating-point vector. If the PAIRED keyword is set, X and Y must have the same number of elements.

Keywords

PAIRED

If this keyword is set, X and Y are assumed to be paired samples and must have the same number of elements.

UNEQUAL

If this keyword is set, X and Y are assumed to be from populations with unequal variances.

Examples

```
; Define two n-element sample populations.
X = [257, 208, 296, 324, 240, 246, 267, 311, 324, 323, 263, $
     305, 270, 260, 251, 275, 288, 242, 304, 267]
Y = [201, 56, 185, 221, 165, 161, 182, 239, 278, 243, 197, $
     271, 214, 216, 175, 192, 208, 150, 281, 196]

; Compute the Student's t-statistic and its significance assuming
; that X and Y belong to populations with the same true variance:
PRINT, TM_TEST(X, Y)
```

IDL prints:

```
5.52839  2.52455e-06
```

The result indicates that X and Y have significantly different means.

Version History

Introduced: 4.0

See Also

[FV_TEST](#), [KW_TEST](#), [RS_TEST](#), [S_TEST](#)

TOTAL

The TOTAL function returns the sum of the elements of *Array*. The sum of the array elements over a given dimension is returned if the *Dimension* argument is present.

Syntax

```
Result = TOTAL( Array [, Dimension] [, /CUMULATIVE] [, /DOUBLE]
               [, /INTEGER] [, /NAN] [, /PRESERVE_TYPE] )
```

Return Value

Returns the array sum for the specified dimensions.

Arguments

Array

The array to be summed. This array can be of any basic type except string. If *Array* is double-precision floating-point, complex, or double-precision complex, the result is of the same type. Otherwise, the result is single-precision floating-point.

Dimension

The dimension over which to sum, starting at one. If this argument is not present or zero, the scalar sum of all the array elements is returned. If this argument is present, the result is an array with one less dimension than *Array*. For example, if the dimensions of *Array* are N_1, N_2, N_3 , and *Dimension* is 2, the dimensions of the result are (N_1, N_3) , and element (i, j) of the result contains the sum:

$$\sum_{k=0}^{N_2-1} A_{i, k, j}$$

Keywords

CUMULATIVE

If this keyword is set, the result is an array of the same size as the input, with each element, i , containing the sum of the input array elements 0 to i . This keyword also works with the *Dimension* parameter, in which case the sum is performed over the given dimension.

Tip

If the input array is a temporary variable or an expression, and the result type matches the input type (for example by using the `PRESERVE_TYPE` keyword), then `TOTAL` will perform the cumulative sum in place and no additional memory will be used.

DOUBLE

Set this keyword to perform the summation in double-precision floating-point.

INTEGER

Set this keyword to perform the `TOTAL` using integer arithmetic, and to return an integer result. If *Array* is of type `ULONG64` then unsigned 64-bit integers are used for the computation and the `Result` is of type `ULONG64`, otherwise signed 64-bit integers are used and the `Result` is of type `LONG64`. If *Array* is complex and `INTEGER` is set, then only the real part of each value is used for the computation. The `DOUBLE` keyword is ignored if `INTEGER` is set.

Note

If *Array* has a large number of values or the values themselves are large, then the `TOTAL` with the `INTEGER` keyword may easily overflow the largest 64-bit integer and return an incorrect result. In this case you may want to avoid using the `INTEGER` keyword.

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Note

Since the value `NaN` is treated as missing data, if *Array* contains only `NaN` values the `TOTAL` routine will return 0.

PRESERVE_TYPE

Set this keyword to perform the `TOTAL` using the input type, and to return a result of the same type. The `DOUBLE` and `INTEGER` keywords are ignored if `PRESERVE_TYPE` is set.

Note

For byte or integer data the `TOTAL` with `PRESERVE_TYPE` may easily overflow the largest value for that type. The `PRESERVE_TYPE` keyword is only useful if you know that your values will not overflow, or you want to exploit the overflow, for example in computing a checksum. Otherwise, you may want to use the `INTEGER` keyword instead.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the `!CPU` system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords `TPOOL_MAX_ELTS`, `TPOOL_MIN_ELTS`, and `TPOOL_NOTHREAD` to override the defaults established by `!CPU` for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

The thread pool is used for non-cumulative sums, and is not used if the `CUMULATIVE` keyword is specified. In a cumulative sum, each result value depends on all of the previous results, so overlapped execution is not possible.

You should be aware that when summing a large number of values, the result from `TOTAL` can depend heavily upon the order in which the numbers are added. Since the thread pool will add values in a different order, you may obtain a different — but equally correct — result than that obtained using the standard non-threaded implementation. This effect occurs because `TOTAL` uses floating point arithmetic, and the mantissa of a floating point value has a fixed number of significant digits. The effect is especially obvious when using single precision arithmetic, but can also affect double precision computations. Such differences do not mean that the sums are incorrect. Rather, they mean that they are equal within the ability of the floating point representation used to represent them. For more information on floating-point numbers, see [“Accuracy & Floating-Point Operations”](#) in Chapter 22 of the *Using IDL* manual.

It is also worth noting that this effect, while illustrated by the use of the thread pool, is not caused by the use of threading. It is simply caused by the different order in which

the numbers are summed, as can be illustrated by the following non-threaded example:

```
vec = FINDGEN(100000)
PRINT, TOTAL(vec, /TPOOL_NO) - TOTAL(REVERSE(vec), /TPOOL_NO)
```

IDL prints:

```
-96768.0
```

As you can see, the small floating-point errors can accumulate across the sum of a large number of values.

Note

The computation above was done on a Sun Sparc workstation. Your result will depend on the architecture of your CPU; it may be slightly different, and in the notable case of Intel-compatible X86 CPUs may actually be zero due to the use of internal 80-bit floating point registers on that CPU which give it better than double precision accuracy for some computations. Nonetheless, you should be aware of the fact that the order of operations can influence the result.

Examples

Example 1

This example sums the elements of a one-dimensional array:

```
; Define a one-dimensional array:
A = [20, 10, 5, 5, 3]

; Sum the elements of the array:
SUMA = TOTAL([20, 10, 5, 5, 3])

; Print the results:
PRINT, 'A = ', A
PRINT, 'Sum of A = ', SUMA
```

IDL prints:

```
A =   20   10   5   5   3
Sum of A =  43.0000
```

Example 2

The results are different when a multi-dimensional array is used:

```
; Define a multi-dimensional array:
A = FINDGEN(5,5)
```

```

; Sum each of the rows in A:
SUMROWS = TOTAL(A, 1)

; Sum each of the columns in A:
SUMCOLS = TOTAL(A, 2)

; Print the results:
PRINT, 'A = ', A
PRINT, 'Sum of each row:', SUMROWS
PRINT, 'Sum of each column:', SUMCOLS

```

IDL prints:

```

A = 0.000000  1.00000  2.00000  3.00000  4.00000
     5.00000  6.00000  7.00000  8.00000  9.00000
     10.0000  11.0000  12.0000  13.0000  14.0000
     15.0000  16.0000  17.0000  18.0000  19.0000
     20.0000  21.0000  22.0000  23.0000  24.0000

```

```

Sum of each row:  10.0000  35.0000  60.0000  85.0000  110.000

```

```

Sum of each column:  50.0000  55.0000  60.0000  65.0000  70.0000

```

Version History

Introduced: Original

INTEGER and PRESERVE_TYPE keywords: 6.1

See Also

[FACTORIAL](#), [PRODUCT](#)

TRACE

The TRACE function computes the trace of an n by n array.

This routine is written in the IDL language. Its source code can be found in the file `trace.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = TRACE( A [, /DOUBLE] )
```

Return Value

Returns the sum of the values along the array diagonal.

Arguments

A

An n by n real or complex array.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```
; Define an array:
A = [[ 2.0,1.0, 1.0, 1.5], $
      [ 4.0, -6.0, 0.0, 0.0], $
      [-2.0, 7.0, 2.0, 2.5], $
      [ 1.0, 0.5, 0.0, 5.0]]

; Compute the trace of A:
result = TRACE(A)

;Print the result:
PRINT, 'TRACE(A) = ', result
```

IDL prints:

```
TRACE(A) = 3.00000
```

Version History

Introduced: 5.0

See Also

[TOTAL](#)

TrackBall Object

See [“TrackBall”](#) on page 4231.

TRANSPOSE

The TRANSPOSE function returns the transpose of *Array*. If an optional permutation vector is provided, the dimensions of *Array* are rearranged as well.

Syntax

Result = TRANSPOSE(*Array* [, *P*])

Return Value

Returns the reflection of the array along a diagonal.

Arguments

Array

The array to be transposed.

P

A vector specifying how the dimensions of *Array* will be permuted. The elements of *P* correspond to the dimensions of *Array*; the *i*th dimension of the output array is dimension *P*[*i*] of the input array. Each element of the vector *P* must be unique. Dimensions start at zero and can not be repeated.

If *P* is not present, the order of the dimensions of *Array* is reversed.

Keywords

None.

Examples

Example 1

Print a simple array and its transpose by entering:

```
; Create an array:  
A = INDGEN(3,3)  
TRANSA = TRANSPOSE(A)
```

```

; Print the array and its transpose:
PRINT, 'A:'
PRINT, A
PRINT, 'Transpose of A:'
PRINT, TRANSA

```

IDL prints:

```

A:
  0  1  2
  3  4  5
  6  7  8

Transpose of A:
  0  3  6
  1  4  7
  2  5  8

```

Example 2

This example demonstrates multi-dimensional transposition:

```

; Create the array:
A = INDGEN(2, 3, 4)

; Take the transpose, reversing the order of the indices:
B = TRANSPOSE(A)

; Re-order the dimensions of A, so that the second dimension
; becomes the first, the third becomes the second, and the first
; becomes the third:
C = TRANSPOSE(A, [1, 2, 0])

; View the sizes of the three arrays:
HELP, A, B, C

```

IDL prints:

```

A  INT  = Array[2, 3, 4]
B  INT  = Array[4, 3, 2]
C  INT  = Array[3, 4, 2]

```

Version History

Introduced: Original

See Also

[REFORM](#), [ROT](#), [ROTATE](#), [REVERSE](#)

TRI_SURF

The TRI_SURF function interpolates a regularly- or irregularly-gridded set of points with a smooth quintic surface.

TRI_SURF is similar to MIN_CURVE_SURF but the surface fitted is a smooth surface, not a minimum curvature surface. TRI_SURF has the advantage of being much more efficient for larger numbers of points.

Note

The TRI_SURF function is designed to interpolate low resolution data. Large arrays may cause TRI_SURF to issue the following error message:

Partial Derivative Approximation Failed to Converge”

In such cases, interpolation is most likely unnecessary.

This routine is written in the IDL language. Its source code can be found in the file `tri_surf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = TRI_SURF( Z [, X, Y] [, /EXTRAPOLATE] [, /LINEAR]
  [, MISSING=value] [, /REGULAR] [, XGRID=[xstart, xspacing] |
  [, XVALUES=array]] [, YGRID=[yxstart, yspacing] | [, YVALUES=array]]
  [, GS=[xspacing, yspacing]] [, BOUNDS=[xmin, ymin, xmax, ymax]]
  [, NX=value] [, NY=value] )
```

Return Value

The result is a two-dimensional floating-point array containing the interpolated surface, sampled at the grid points.

Arguments

X, Y, Z

arrays containing the X, Y, and Z coordinates of the data points on the surface. Points need not be regularly gridded. For regularly gridded input data, X and Y are not used: the grid spacing is specified via the XGRID and YGRID (or XVALUES and YVALUES) keywords, and Z must be a two dimensional array. For irregular grids, all three parameters must be present and have the same number of elements.

Keywords

EXTRAPOLATE

Set this keyword to cause TRI_SURF to extrapolate the surface to points outside the convex hull of input points. This keyword has no effect if the input points are regularly gridded.

LINEAR

Set this keyword to use linear interpolation, without gradient estimates, instead of quintic interpolation. Linear interpolation does not extrapolate, although it is faster and more numerically stable.

MISSING

Set this keyword equal to the value to which points outside the convex hull of input points should be set. The default is 0. This keyword has no effect if the input points are regularly gridded.

Input Grid Description:

REGULAR

If set, the Z parameter is a two-dimensional array of dimensions (n,m) , containing measurements over a regular grid. If any of XGRID, YGRID, XVALUES, or YVALUES are specified, REGULAR is implied. REGULAR is also implied if there is only one parameter, Z . If REGULAR is set, and no grid specifications are present, the grid is set to $(0, 1, 2, \dots)$.

XGRID

A two-element array, $[xstart, xspacing]$, defining the input grid in the x direction. Do not specify both XGRID and XVALUES.

XVALUES

An n -element array defining the x locations of $Z[i,j]$. Do not specify both XGRID and XVALUES.

YGRID

A two-element array, $[ystart, yspacing]$, defining the input grid in the y direction. Do not specify both YGRID and YVALUES.

YVALUES

An n -element array defining the y locations of $Z[i,j]$. Do not specify both YGRID and YVALUES.

Output Grid Description:

Note

The output grid must enclose the convex hull of the input points.

GS

The output grid spacing. If present, GS must be a two-element vector $[xs, ys]$, where xs is the horizontal spacing between grid points and ys is the vertical spacing. The default is based on the extents of x and y . If the grid starts at x value $xmin$ and ends at $xmax$, then the default horizontal spacing is $(xmax - xmin)/(NX-1)$. YS is computed in the same way. The default grid size, if neither NX or NY are specified, is 26 by 26.

BOUNDS

If present, BOUNDS must be a four-element array containing the grid limits in x and y of the output grid: $[xmin, ymin, xmax, ymax]$. If not specified, the grid limits are set to the extent of x and y .

NX

The output grid size in the x direction. NX need not be specified if the size can be inferred from GS and BOUNDS. The default value is 26.

NY

The output grid size in the y direction. NY need not be specified if the size can be inferred from GS and BOUNDS. The default value is 26.

Examples

Example 1

Regularly gridded case:

```
; Make some random data
Z = randomu(seed, 5, 6)

; Interpolate to a 26 x 26 grid:
CONTOUR, TRI_SURF(Z, /REGULAR)
```

Example 2

Irregularly gridded case:

```
; Make a random set of points that lie on a Gaussian:
N = 15
X = RANDOMU(seed, N)
Y = RANDOMU(seed, N)

; The Gaussian:
Z = EXP(-2 * ((X-.5)^2 + (Y-.5)^2))

; Use a 26 by 26 grid over the rectangle bounding x and y.
; Get the surface:
R = TRI_SURF(Z, X, Y)

; Alternatively, get a surface over the unit square, with spacing
; of 0.05:
R = TRI_SURF(z, x, y, GS=[0.05, 0.05], BOUNDS=[0,0,1,1])

; Alternatively, get a 10 by 10 surface over the rectangle bounding
; x and y:
R = TRI_SURF(z, x, y, NX=10, NY=10)
```

Version History

Introduced: Pre 4.0

See Also

[CONTOUR](#), [MIN_CURVE_SURF](#)

TRIANGULATE

The TRIANGULATE procedure constructs a Delaunay triangulation of a planar set of points. Delaunay triangulations are very useful for the interpolation, analysis, and visual display of irregularly-gridded data. In most applications, after the irregularly gridded data points have been triangulated, the function TRIGRID is invoked to interpolate surface values to a regular grid.

Since Delaunay triangulations have the property that the circumcircle of any triangle in the triangulation contains no other vertices in its interior, interpolated values are only computed from nearby points.

TRIANGULATE can, optionally, return the adjacency list that describes, for each node, the adjacent nodes in the Delaunay triangulation. With this list, the Voronoi polygon (the polygon described by the set of points which are closer to that node than to any other node) can be computed for each node. This polygon contains the area influenced by its associated node. Tiling of the region in this manner is also called Dirichlet, Wigner-Seitz, or Thiessen tessellation.

The grid returned by the TRIGRID function can be input to various routines such as SURFACE, TV, and CONTOUR. See the description of TRIGRID for an example.

TRIANGULATE and TRIDGRID can also be used to perform gridding and interpolation over the surface of a sphere. The interpolation is C_1 continuous, meaning that the result is continuous over both the function value and its first derivative. This feature is ideal for interpolating an irregularly-sampled dataset over part or all of the surface of the earth (or other (spherical) celestial bodies). Extrapolation outside the convex hull of sample points is also supported. To perform spherical gridding, you must include the FVALUE and SPHERE keywords described below. The spherical gridding technique used in IDL is based on the paper "Interpolation of Data on the Surface of a Sphere", R. Renka, *Oak Ridge National Laboratory Report ORNL/CSD-108*, 1982.

Syntax

```
TRIANGULATE, X, Y, Triangles [, B] [, CONNECTIVITY=variable]
  [, SPHERE=variable [, /DEGREES]] [, FVALUE=variable]
  [, REPEATS=variable]
```

Arguments

X

An array that contains the X coordinates of the points to be triangulated.

Y

An array that contains the Y coordinates of the points to be triangulated. Parameters X and Y must have the same number of elements.

Triangles

A named variable that, on exit, contains the list of triangles in the Delaunay triangulation of the points specified by the X and Y arguments. Triangles is a longword array dimensioned (3, *number of triangles*), where `Triangles[0, i]`, `Triangles[1, i]`, and `Triangles[2, i]` contain the indices of the vertices of the *i*-th triangle (i.e., `X[Tr[* , i]]` and `Y[Triangles[* , i]]` are the X and Y coordinates of the vertices of the *i*-th triangle).

B

An optional, named variable that, upon return, contains a list of the indices of the boundary points in counterclockwise order.

Keywords

CONNECTIVITY

Set this keyword to a named variable in which the adjacency list for each of the *N* nodes (xy point) is returned. The list has the following form:

Each element *i*, $0 \leq i < N$, contains the starting index of the connectivity list for node *i* within the list array. To obtain the adjacency list for node *i*, extract the list elements from `LIST[i]` to `LIST[i+1]-1`.

The adjacency list is ordered in the counter-clockwise direction. The first item on the list of boundary nodes is the subscript of the node itself. For interior nodes, the list contains the subscripts of the adjacent nodes in counter-clockwise order.

For example, the call:

```
TRIANGULATE, X, Y, CONNECTIVITY = LIST
```

returns the adjacency list in the variable LIST. The subscripts of the nodes adjacent to `X[i]` and `Y[i]` are contained in the array:

```
LIST[LIST[i] : LIST[i+1]-1]
```

DEGREES

Set this keyword to indicate that the X and Y arguments contain longitude and latitude coordinates specified in degrees. This keyword is only effective if the SPHERE keyword is specified. If DEGREES is not set, X and Y are assumed to be specified in radians when a spherical triangulation is performed.

FVALUE

Set this keyword to a named variable that contains sample values for each longitude/latitude point in a spherical triangulation. On output, the elements of FVALUE are rearranged to correspond to the new ordering of X and Y (as described in the SPHERE keyword, below). This reordered array can be passed to TRIGRID to complete the interpolation.

REPEATS

Set this keyword to a named variable to return a $(2, n)$ list of the indices of duplicated points. That is, for each i ,

```
X[REPEATS[0,i]] = X[REPEATS[1,i]]
```

and

```
Y[REPEATS[0,i]] = Y[REPEATS[1,i]]
```

Note

Use the [GRID_INPUT](#) procedure to handle repeated points (duplicate locations).

SPHERE

Set this keyword to a named variable in which the results from a spherical triangulation are returned. This result is a structure that can be passed to TRIGRID to perform spherical gridding. The structure contains the 3D Cartesian locations sample points and the adjacency list that describes the triangulation.

When spherical triangulation is performed, X and Y are interpreted as longitude and latitude, in either degrees or radians (see the DEGREE keyword, above). Also, the order of the elements within the X and Y parameters is rearranged (see the FVALUE keyword, above).

Examples

For some examples using the TRIANGULATE routine, see the [TRIGRID](#) function.

Version History

Introduced: Pre 4.0

See Also

[SPH_SCAT](#), [TRIGRID](#)

TRIGRID

For non-spherical data, given data points defined by the parameters X , Y , and Z and a triangulation of the planar set of points determined by X and Y , the TRIGRID function returns a regular grid of interpolated Z values. Linear or smooth quintic polynomial interpolation can be selected. Extrapolation for gridpoints outside of the triangulation area is also an option.

For spherical data, given the data values F and the spherical triangulation, the TRIGRID function returns a regular grid of interpolated F values using quadratic polynomial interpolation

An input triangulation can be constructed using the procedure TRIANGULATE. Together, the TRIANGULATE procedure and the TRIGRID function constitute IDL's solution to the problem of irregularly-gridded data, including spherical gridding.

Syntax

Result = TRIGRID(X , Y , Z , *Triangles* [, *GS*, *Limits*])

For spherical gridding: *Result* = TRIGRID(F , *GS*, *Limits*, SPHERE= S)

Keywords: [, /DEGREES] [, EXTRAPOLATE=*array*] [, INPUT=*variable*] [, MAX_VALUE=*value*] [, MIN_VALUE=*value*] [, MISSING=*value*] [, NX=*value*] [, NY=*value*] [, /QUINTIC] [, SPHERE=*variable*] [, XGRID=*variable*] [, XOUT=*vector*] [, YGRID=*variable*] [, YOUT=*vector*]

Return Value

The resulting grid is a two-dimensional array with user-specified bounds and spacing. The data type of the resulting grid is either floating-point or double-precision floating-point depending upon the data type of Z (or F for spherical).

Arguments

X , Y , Z

Input arrays of X , Y , and Z coordinates of data points. Integer, long, double-precision and floating-point values are allowed. In addition, Z can be a complex array. All three arrays must have the same number of elements.

F

When performing a spherical gridding, this argument should be the named variable that contains the rearranged sample values that were returned by TRIANGULATE's FVALUE keyword.

Triangles

A longword array of the form output by TRIANGULATE. That is, *Triangles* has the dimensions (3, *number of triangles*) and, for each *i*, `Triangles[0, i]`, `Triangles[1, i]`, and `Triangles[2, i]` are the indices of the vertices of the *i*-th triangle.

GS

If present, *GS* should be a two-element vector [*XS*, *YS*], where *XS* is the horizontal spacing between grid points and *YS* is the vertical spacing. The default is based on the extents of *X* and *Y*. If the grid starts at *X* value x_0 and ends at x_1 , then the horizontal spacing is

$$(x_1 - x_0)/50$$

The default for *YS* is computed in the same way. Since the default grid spacing divides each axis into 50 intervals and produces 51 samples, TRIGRID returns a grid with dimensions (51, 51).

If the *NX* or *NY* keywords are set to specify the output grid dimensions, either or both of the values of *GS* may be set to 0. In this case, the grid spacing is computed as the respective range divided by the dimension minus one:

$$(x_1 - x_0)/(NX-1) \text{ and } (y_1 - y_0)/(NY-1)$$

For spherical gridding, *GS* is assumed to be specified in radians, unless the DEGREES keyword is set.

Limits

If present, *Limits* should be a four-element vector [x_0 , y_0 , x_1 , y_1] that specifies the data range to be gridded (x_0 and y_0 are the lower X and Y data limits, and x_1 and y_1 are the upper limits). The default for *Limits* is:

$$[\text{MIN}(X), \text{MIN}(Y), \text{MAX}(X), \text{MAX}(Y)]$$

If the *NX* or *NY* keywords are not specified, the size of the grid produced is specified by the value of *Limits*. If the *NX* or *NY* keywords are set to specify the output grid dimensions, a grid of the specified size will be used regardless of the value of *Limits*.

Keywords

DEGREES

For a spherical gridding, set this keyword to indicate that the grid spacing (the *GS* argument) is specified in degrees rather than radians.

EXTRAPOLATE

Set this keyword equal to an array of boundary node indices (as returned by the optional parameter *B* of the [TRIANGULATE](#) procedure) to extrapolate to grid points outside the triangulation. The extrapolation is not smooth, but should give acceptable results in most cases.

Setting this keyword sets the quintic interpolation mode, as if the *QUINTIC* keyword has been specified.

This keyword is ignored for spherical gridding.

INPUT

Set this keyword to a named variable (which must be an array of the appropriate size to hold the output from *TRIGRID*) in which the results of the gridding are returned. This keyword is provided to make it easy and memory-efficient to perform multiple calls to *TRIGRID*. The interpolates within each triangle overwrite the array and the array is not initialized.

MAX_VALUE

Set this keyword to a value that represents the maximum *Z* value to be gridded. Data larger than this value are treated as missing data and are not gridded.

This keyword is ignored for spherical gridding.

MIN_VALUE

Set this keyword to a value that represents the minimum *Z* value to be gridded. Data smaller than this value are treated as missing data and are not gridded.

This keyword is ignored for spherical gridding.

MISSING

The *Z* value to be used for grid points that lie outside the triangles in *Triangles*. The default is 0. This keyword also applies to data points outside the range specified by *MIN_VALUE* and *MAX_VALUE*.

Note

Letting MISSING default to 0 does not always produce the same result as explicitly setting it to 0. For example, if you specify INPUT and not EXTRAPOLATE, letting MISSING default to 0 will result in the INPUT values being used for data outside the Triangles; explicitly setting MISSING to 0 will result in 0 being used for the data outside the Triangles.

NX

The output grid size in the x direction. The default value is 51.

NY

The output grid size in the y direction. The default value is 51.

QUINTIC

If QUINTIC is set, smooth interpolation is performed using Akima's quintic polynomials from "A Method of Bivariate Interpolation and Smooth Surface Fitting for Irregularly Distributed Data Points" in *ACM Transactions on Mathematical Software*, 4, 148-159. For non-spherical data, the default method is linear interpolation. For spherical data, the default method is quadratic interpolation, and the QUINTIC keyword is ignored.

Derivatives are estimated by Renka's global method in "A Triangle-Based C1 Interpolation Method" in *Rocky Mountain Journal of Mathematics*, vol. 14, no. 1, 1984.

QUINTIC is not available for complex data values. Setting the EXTRAPOLATE keyword implies the use of quintic interpolation; it is not necessary to specify both.

SPHERE

For a spherical gridding, set this keyword to the named variable that contains the results of the spherical triangulation returned by TRIANGULATE's SPHERE keyword.

XGRID

Set this keyword to a named variable that will contain a vector of X values for the output grid.

XOUT

Set this keyword to a vector specifying the output grid X values. If this keyword is supplied, the *GS* and *Limits* arguments are ignored. Use this keyword to specify irregularly spaced rectangular output grids. If XOUT is specified, YOUT must also be specified. If keyword NX is also supplied then only the first NX points of XOUT will be used.

YGRID

Set this keyword to a named variable that will contain a vector of Y values for the output grid.

The following table shows the interrelationships between the keywords EXATRAPOLATE, INPUT, MAX_VALUE, MIN_VALUE, MISSING, and QUINTIC.

INPUT	EXTRAPOLATE	MISSING	Not in Triangles	Beyond MIN_VALUE, MAX_VALUE
no	no	no	uses 0	uses 0
no	no	yes	uses MISSING	uses MISSING
no	yes	no	EXTRAPOLATEs	uses 0
no	yes	yes	EXTRAPOLATEs	uses MISSING
yes	no	no	uses INPUT	uses INPUT
yes	no	yes	uses MISSING	uses MISSING
yes	yes	no	EXTRAPOLATEs	uses INPUT
yes	yes	yes	EXTRAPOLATEs	uses MISSING

Table 3-112: Keyword Interrelationships for the TRIGRID function

YOUT

Set this keyword to a vector specifying the output grid Y values. If this keyword is supplied, the *GS* and *Limits* arguments are ignored. Use this keyword to specify irregularly spaced rectangular output grids. If keyword NY is also supplied then only the first NY points of YOUT will be used.

Examples

Example 1

This example creates and displays a 50 point random normal distribution. The random points are then triangulated, with the triangulation displayed. Next, the interpolated surface is computed and displayed using linear and quintic interpolation. Finally, the smooth extrapolated surface is generated and shown.

```

PRO TrigridExample

; Make 50 normal x, y points:
x = RANDOMN(seed, 50)
y = RANDOMN(seed, 50)

; Make the Gaussian:
z = EXP(-(x^2 + y^2))

; Show points:
PLOT, x, y, psym=1
in=' '
READ,"Press enter",in

; Obtain triangulation:
TRIANGULATE, x, y, tr, b

; Show the triangles:
FOR i=0, N_ELEMENTS(tr)/3-1 DO BEGIN & $
    ; Subscripts of vertices [0,1,2,0]:
    t = [tr[*],i], tr[0,i]] & $
    ; Connect triangles:
    PLOTS, x[t], y[t] & $
ENDFOR

; Show linear surface:
SURFACE, TRIGRID(x, y, z, tr)
in=' '
READ,"Press enter",in

; Show smooth quintic surface:
SURFACE, TRIGRID(x, y, z, tr, /QUINTIC)
in=' '
READ,"Press enter",in

; Show smooth extrapolated surface:
SURFACE, TRIGRID(x, y, z, tr, EXTRA = b)
in=' '
READ,"Press enter",in

```

```

; Output grid size is 12 x 24:
SURFACE, TRIGRID(X, Y, Z, Tr, NX=12, NY=24)
in=' '
READ,"Press enter",in

; Output grid size is 20 x 11. The X grid is
; [0, .1, .2, ..., 19 * .1 = 1.9]. The Y grid goes from 0 to 1:
SURFACE, TRIGRID(X, Y, Z, Tr, [.1, .1], NX=20)
in=' '
READ,"Press enter",in

; Output size is 20 x 40. The range of the grid in X and Y is
; specified by the Limits parameter. Grid spacing in X is
;  $[5-0]/(20-1) = 0.263$ . Grid spacing in Y is  $(4-0)/(40-1) = 0.128$ :
SURFACE, TRIGRID(X, Y, Z, Tr, [0,0], [0,0,5,4],NX=20, NY=40)
in=' '
READ,"Press enter",in

WDELETE

END

```

Example 2

This example shows how to perform spherical gridding:

```

PRO SphericalGrid
; Create some random longitude points:
lon = RANDOMU(seed, 50) * 360. - 180.

; Create some random latitude points:
lat = RANDOMU(seed, 50) * 180. - 90.

; Make a fake function value to be passed to FVALUE. The system
; variable !DTOR contains the conversion value for degrees to
; radians.
f = SIN(lon * !DTOR)^2 * COS(lat * !DTOR)

; Perform a spherical triangulation:
TRIANGULATE, lon, lat, tr, $
    SPHERE=s, FVALUE=f, /DEGREES

; Perform a spherical triangulation using the values returned from
; TRIANGULATE. The result, r, is a 180 by 91 element array:
r=TRIGRID(f, SPHERE=s, [2.,2.],$
    [-180.,-90.,178.,90.], /DEGREES)

; Display the surface

```

```
SURFACE, r
END
```

Example 3

This example demonstrates the use of the INPUT keyword:

```
PRO TrigridInputKeyword

; Make 50 normal x, y points:
x = RANDOMN(seed, 50)
y = RANDOMN(seed, 50)

; Make the Gaussian:
z = EXP(-(x^2 + y^2))

; Show points:
PLOT, x, y, psym=1

; Obtain triangulation:
TRIANGULATE, x, y, tr, b

; Show the triangles.
FOR i=0, N_ELEMENTS(tr)/3-1 DO BEGIN $
    ; Subscripts of vertices [0,1,2,0]:
    t = [tr[*],i], tr[0,i]] & $
    ; Connect triangles:
    PLOTS, x[t], y[t]
ENDFOR

; The default size for the return value of trigrid. xtemp should be
; the same type as Z. xtemp provides temporary space for trigrid:
xtemp=FLTARR(51,51)
xtemp = TRIGRID(x, y, z, INPUT = xtemp, tr)

; Show linear surface:
SURFACE, xtemp, TITLE='Linear surface', CHARSIZE=2
in=' '
READ,"Press enter",in
xtemp = TRIGRID(x, y, z, tr, INPUT = xtemp, /QUINTIC)

; Show smooth quintic surface:
SURFACE, xtemp, TITLE='Smooth Quintic surface', CHARSIZE=2
in=' '
READ,"Press enter",in
xtemp = TRIGRID(x, y, z, tr, INPUT = xtemp, EXTRA = b)

; Show smooth extrapolated surface:
SURFACE, xtemp, TITLE='Smooth Extrapolated surface', CHARSIZE=2
```

```

in= ' '
READ, "Press enter", in
END

```

Example 4

The XOUT and YOUT keywords allow you to obtain an irregular interval from the TRIGRID routine. This example creates an irregularly-gridded dataset of a Gaussian surface. A grid is formed from these points with the TRIANGULATE and TRIGRID routines. The inputs to the XOUT and YOUT keywords are determined at random to produce an irregular interval. These inputs are sorted before setting them to XOUT and YOUT because these keywords require monotonically ascending or descending values. The lines of the resulting surface are spaced at the irregular intervals provided by the settings of the XOUT and YOUT keywords.

```

PRO GriddingIrregularIntervals

; Make 100 normal x, y points:
x = RANDOMN(seed, 100)
y = RANDOMN(seed, 100)
PRINT, MIN(x), MAX(x)
PRINT, MIN(y), MAX(y)

; Make a Gaussian surface:
z = EXP(-(x^2 + y^2))

; Obtain triangulation:
TRIANGULATE, x, y, triangles, boundary

; Create random x values. These values will be used to
; form the x locations of the resulting grid.
gridX = RANDOMN(seed, 30)
; Sort x values. Sorted values are required for the XOUT
; keyword.
sortX = UNIQ(gridX, SORT(gridX))
gridX = gridX[sortX]
; Output sorted x values to be used with the XOUT
; keyword.
PRINT, 'gridX:'
PRINT, gridX

; Create random y values. These values will be used to
; form the y locations of the resulting grid.
gridY = RANDOMN(seed, 30)
; Sort y values. Sorted values are required for the YOUT
; keyword.
sortY = UNIQ(gridY, SORT(gridY))
gridY = gridY[sortY]

```

```
; Output sorted y values to be used with the YOUT
; keyword.
PRINT, 'gridY:'
PRINT, gridY

; Derive grid of initial values. The location of the
; resulting grid points are the inputs to the XOUT and
; YOUT keywords.
grid = TRIGRID(x, y, z, triangles, XOUT = gridX, $
              YOUT = gridY, EXTRAPOLATE = boundary)

; Display resulting grid. The grid lines are not
; at regular intervals because of the randomness of the
; inputs to the XOUT and YOUT keywords.
SURFACE, grid, gridX, gridY, /XSTYLE, /YSTYLE

END
```

Version History

Introduced: Pre 4.0

See Also

[SPH_SCAT](#), [TRIANGULATE](#)

TRIQL

The TRIQL procedure uses the QL algorithm with implicit shifts to determine the eigenvalues and eigenvectors of a real, symmetric, tridiagonal array. The routine TRIRED can be used to reduce a real, symmetric array to the tridiagonal form suitable for input to this procedure.

TRIQL is based on the routine `tqli` described in section 11.3 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the LA_TRIQL procedure.

Syntax

TRIQL, *D*, *E*, *A* [, /DOUBLE]

Arguments

D

On input, this argument should be an n -element vector containing the diagonal elements of the array being analyzed. On output, D contains the eigenvalues.

E

An n -element vector containing the off-diagonal elements of the array. E_0 is arbitrary. On output, this parameter is destroyed.

A

A named variable that returns the n eigenvectors. If the eigenvectors of a tridiagonal array are desired, A should be input as an identity array. If the eigenvectors of an array that has been reduced by TRIRED are desired, A is input as the array Q output by TRIRED.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

To compute eigenvalues and eigenvectors of a real, symmetric, tridiagonal array, begin with an array A representing a symmetric array:

```

; Create the array A:
A = [[ 3.0,  1.0, -4.0], $
      [ 1.0,  3.0, -4.0], $
      [-4.0, -4.0,  8.0]]

; Compute the tridiagonal form of A:
TRIRED, A, D, E

; Compute the eigenvalues (returned in vector D) and the
; eigenvectors (returned in the rows of the array A):
TRIQL, D, E, A

; Print eigenvalues:
PRINT, 'Eigenvalues:'
PRINT, D

; Print eigenvectors:
PRINT, 'Eigenvectors:'
PRINT, A

```

IDL prints:

```

Eigenvalues:
  2.00000  4.76837e-7  12.0000

Eigenvectors:
  0.707107  -0.707107  0.00000
 -0.577350  -0.577350  -0.577350
 -0.408248  -0.408248  0.816497

```

The exact eigenvalues are:

```
[2.0, 0.0, 12.0]
```

The exact eigenvectors are:

```

[ 1.0/sqrt(2.0), -1.0/sqrt(2.0), 0.0/sqrt(2.0)],
[-1.0/sqrt(3.0), -1.0/sqrt(3.0), -1.0/sqrt(3.0)],
[-1.0/sqrt(6.0), -1.0/sqrt(6.0), 2.0/sqrt(6.0)]

```

Version History

Introduced: 4.0

See Also

[EIGENVEC](#), [ELMHES](#), [HQR](#), [LA_TRIQL](#), [TRIRED](#)

TRIRED

The TRIRED procedure uses Householder's method to reduce a real, symmetric array to tridiagonal form.

TRIRED is based on the routine `tred2` described in section 11.2 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the `LA_TRIRED` procedure.

Syntax

TRIRED, A, D, E [, /DOUBLE]

Arguments

A

An n by n real, symmetric array that is replaced, on exit, by the orthogonal array Q effecting the transformation. The routine `TRIQL` can use this result to find the eigenvectors of the array A .

D

An n -element output vector containing the diagonal elements of the tridiagonal array.

E

An n -element output vector containing the off-diagonal elements.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

See the description of `TRIQL` for an example using this function.

Version History

Introduced: 4.0

See Also

[EIGENVEC](#), [ELMHES](#), [HQR](#), [LA_TRIRED](#), [TRIQL](#)

TRISOL

The TRISOL function solves tridiagonal systems of linear equations that have the form: $A^T U = R$

Note

Because IDL subscripts are in column-row order, the equation above is written $A^T U = R$ rather than $AU = R$. The result U is a vector of length n whose type is identical to A .

TRISOL is based on the routine `tridag` described in section 2.4 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Note

If you are working with complex inputs, instead use the `LA_TRISOL` procedure.

Syntax

Result = TRISOL(*A*, *B*, *C*, *R* [, /DOUBLE])

Return Value

Returns a vector containing the solutions.

Arguments

A

A vector of length n containing the $n-1$ sub-diagonal elements of A^T . The first element of A , A_0 , is ignored.

B

An n -element vector containing the main diagonal elements of A^T .

C

An n -element vector containing the $n-1$ super-diagonal elements of A^T . The last element of C , C_{n-1} , is ignored.

R

An n -element vector containing the right hand side of the linear system
 $A^T U = R$.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Example

To solve a tridiagonal linear system, begin with an array representing a real tridiagonal linear system. (Note that only three vectors need be specified; there is no need to enter the entire array shown.)

$$\begin{bmatrix} -4.0 & 1.0 & 0.0 & 0.0 \\ 2.0 & -4.0 & 1.0 & 0.0 \\ 0.0 & 2.0 & -4.0 & 1.0 \\ 0.0 & 0.0 & 2.0 & -4.0 \end{bmatrix}$$

```

; Define a vector A containing the sub-diagonal elements with a
; leading 0.0 element:
A = [0.0, 2.0, 2.0, 2.0]

; Define B containing the main diagonal elements:
B = [-4.0, -4.0, -4.0, -4.0]

; Define C containing the super-diagonal elements with a trailing
; 0.0 element:
C = [1.0, 1.0, 1.0, 0.0]

; Define the right-hand side vector:
R = [6.0, -8.0, -5.0, 8.0]

; Compute the solution and print:
result = TRISOL(A, B, C, R)
PRINT, result

```

IDL prints:

```
-1.00000  2.00000  2.00000  -1.00000
```

The exact solution vector is [-1.0, 2.0, 2.0, -1.0].

Version History

Introduced: 4.0

See Also

[CRAMER](#), [GS_ITER](#), [LA_TRISOL](#), [LU_COMPLEX](#), [CHOLSOL](#), [LUSOL](#),
[SVSOL](#), [TRISOL](#)

TRUNCATE_LUN

The TRUNCATE_LUN procedure truncates the contents of a file (which must be open for write access) at the current position of the file pointer. After this operation, all data before the current file pointer remains intact, and all data following the file pointer are gone. The position of the current file pointer is not altered.

Syntax

```
TRUNCATE_LUN, Unit1, ..., Unitn
```

Arguments

Unit_n

Scalar or array variables containing the logical file unit numbers of the open files to be truncated.

Keywords

None.

Examples

Example 1

Truncate the entire contents of an existing file:

```
OPENU, unit, 'baddata.dat', /GET_LUN
TRUNCATE_LUN, unit
FREE_LUN, unit
```

Example 2

Given an existing file of 10,000 bytes, throw away the final 5,000 bytes, and then write an additional 2,000 byte array in their place. The resulting file will be 7,000 bytes in length.

```
OPENU, unit, 'mydata.dat', /GET_LUN
POINT_LUN, unit, 5000
TRUNCATE_LUN, unit
WRITEU, unit, BYTARR(2000)
FREE_LUN, unit
```

Version History

Introduced: 5.6

See Also

[GET_LUN](#), [OPEN](#), [POINT_LUN](#)

TS_COEF

The `TS_COEF` function computes the coefficients $\phi_1, \phi_2, \dots, \phi_P$ used in a P -th order autoregressive time-series forecasting model. This routine is written in the IDL language. Its source code can be found in the file `ts_coef.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = `TS_COEF`(*X*, *P* [, `/DOUBLE`] [, `MSE=variable`])

Return Value

Returns a P -element vector whose type is identical to X .

Arguments

X

An n -element single- or double-precision floating-point vector containing time-series samples.

P

An integer or long integer scalar that specifies the number of coefficients to be computed.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

MSE

Set this keyword to a named variable that will contain the mean square error of the P -th order autoregressive model.

Examples

```
; Define an n-element vector of time-series samples:  
X = [6.63, 6.59, 6.46, 6.49, 6.45, 6.41, 6.38, 6.26, 6.09, 5.99, $  
      5.92, 5.93, 5.83, 5.82, 5.95, 5.91, 5.81, 5.64, 5.51, 5.31, $  
      5.36, 5.17, 5.07, 4.97, 5.00, 5.01, 4.85, 4.79, 4.73, 4.76]  
; Compute the coefficients of a 5th order autoregressive model:  
PRINT, TS_COEF(X, 5)
```

IDL prints:

```
1.30168      -0.111783      -0.224527      0.267629      -0.233363
```

Version History

Introduced: 4.0

See Also

[TS_FCAST](#)

TS_DIFF

The `TS_DIFF` function recursively computes the forward differences of an n -element time-series k times. This routine is written in the IDL language. Its source code can be found in the file `ts_diff.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = `TS_DIFF`(*X*, *K* [, `/DOUBLE`])

Return Value

The result is an n -element differenced time-series with its last k elements as zeros.

Arguments

X

An n -element integer, single- or double-precision floating-point vector containing time-series samples.

K

A positive integer or long integer scalar that specifies the number of times X is to be differenced. K must be in the interval $[1, N_ELEMENTS(X) - 1]$.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```
; Define an n-element vector of time-series samples:
X = [389, 345, 303, 362, 412, 356, 325, 375, $
     410, 350, 310, 388, 399, 362, 325, 382, $
     399, 382, 318, 385, 437, 357, 310, 391]
; Compute the second forward differences of X:
PRINT, TS_DIFF(X, 2)
```

IDL prints:

```
      2  101   -9 -106   25   81  -15  -95   20
     118  -67  -48    0   94  -40  -34  -47  131
     -15 -132   33  128    0    0
```

Version History

Introduced: 4.0

See Also

[SMOOTH, TS_FCAST](#)

TS_FCAST

The TS_FCAST function computes future or past values of a stationary time-series using a P -th order autoregressive model.

A P -th order autoregressive model relates a forecasted value x_t of the time series $X = [x_0, x_1, x_2, \dots, x_{t-1}]$, as a linear combination of P past values.

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \dots + \phi_P x_{t-P} + w_t$$

The coefficients $\phi_1, \phi_2, \dots, \phi_P$ are calculated such that they minimize the uncorrelated random error terms, w_t .

This routine is written in the IDL language. Its source code can be found in the file `ts_fcast.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = TS_FCAST(*X*, *P*, *Nvalues* [, /BACKCAST] [, /DOUBLE])

Return Value

The result is an *Nvalues*-element vector whose type is identical to *X*.

Arguments

X

An n -element single- or double-precision floating-point vector containing time-series samples.

P

An integer or long integer scalar that specifies the number of actual time-series values to be used in the forecast. In general, a larger number of values results in a more accurate forecast.

Nvalues

An integer or long integer scalar that specifies the number of future or past values to be computed.

Keywords

BACKCAST

Set this keyword to produce past values (backward forecasts or “backcasts”)

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

```

; Define an n-element vector of time-series samples:
X = [6.63, 6.59, 6.46, 6.49, 6.45, 6.41, 6.38, 6.26, 6.09, 5.99, $
      5.92, 5.93, 5.83, 5.82, 5.95, 5.91, 5.81, 5.64, 5.51, 5.31, $
      5.36, 5.17, 5.07, 4.97, 5.00, 5.01, 4.85, 4.79, 4.73, 4.76]

; Compute and print five future values of the time-series using ten
; time-series values:
PRINT, TS_FCAST(X, 10, 5)

; Compute five past values of the time-series using ten time-series
; values:
PRINT, TS_FCAST(X, 10, 5, /BACKCAST)

```

IDL prints:

4.65870	4.58380	4.50030	4.48828	4.46971
6.94862	6.91103	6.86297	6.77826	6.70282

Version History

Introduced: 4.0

See Also

[A_CORRELATE](#), [COMFIT](#), [CURVEFIT](#), [SMOOTH](#), [TS_COEF](#), [TS_DIFF](#)

TS_SMOOTH

The `TS_SMOOTH` function computes central, backward, or forward moving averages of an n -element time-series. Autoregressive forecasting and backcasting are used to extrapolate the time-series and compute a moving average for each point.

Note

Central moving averages require $Nvalues/2$ forecasts and $Nvalues/2$ backcasts. Backward moving averages require $Nvalues-1$ backcasts. Forward moving averages require $Nvalues-1$ forecasts.

This routine is written in the IDL language. Its source code can be found in the file `ts_smooth.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = TS_SMOOTH( X, Nvalues [, /BACKWARD] [, /DOUBLE] [, /FORWARD]
                  [, ORDER=value] )
```

Return Value

The result is an n -element vector of the same data type as the input vector.

Arguments

X

An n -element single- or double-precision floating-point vector containing time-series samples. Note that n must be greater than or equal to 11.

Nvalues

A scalar of type integer or long integer that specifies the number of time-series values used to compute each moving-average. If central-moving averages are computed (the default), this parameter must be an odd integer greater than or equal to three.

Keywords

BACKWARD

Set this keyword to compute backward-moving averages. If BACKWARD is set, the *Nvalues* argument must be an integer greater than one.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FORWARD

Set this keyword to compute forward-moving averages. If FORWARD is set, the *Nvalues* argument must be an integer greater than one.

ORDER

An integer or long-integer scalar that specifies the order of the autoregressive model used to compute the forecasts and backcasts of the time-series. By default, a time-series with a length between 11 and 219 elements will use an autoregressive model with an order of 10. A time-series with a length greater than 219 will use an autoregressive model with an order equal to 5% of its length. The ORDER keyword is used to override this default.

Examples

```
; Define an n-element vector of time-series samples:
X = [6.63, 6.59, 6.46, 6.49, 6.45, 6.41, 6.38, 6.26, 6.09, 5.99,$
     5.92, 5.93, 5.83, 5.82, 5.95, 5.91, 5.81, 5.64, 5.51, 5.31,$
     5.36, 5.17, 5.07, 4.97, 5.00, 5.01, 4.85, 4.79, 4.73, 4.76]

; Compute the 11-point central-moving-averages of the time-series:
PRINT, TS_SMOOTH(X, 11)
```

IDL prints:

```
6.65761 6.60592 6.54673 6.47646 6.40480 6.33364
6.27000 6.20091 6.14273 6.09364 6.04455 5.99000
5.92273 5.85455 5.78364 5.72636 5.65818 5.58000
5.50182 5.42727 5.34182 5.24545 5.15273 5.07000
5.00182 4.94261 4.87205 4.81116 4.75828 4.71280
```

Version History

Introduced: 5.0

See Also

[SMOOTH](#), [TS_DIFF](#), [TS_FCAST](#)

TV

The TV procedure displays images on the image display without scaling the intensity. To display an image with scaling, use the TVSCL procedure.

Note

To display a TrueColor image (an image with 16, 24, or 32 bits per pixel) you must specify the TRUE keyword.

While the TV procedure does not *scale* the intensity of an image, it does convert the input image data to byte type. Values outside the range [0,255] are “wrapped” during the conversion. In addition, for displays with less than 256 colors, elements of the input image with values between !D.TABLE_SIZE and 255 will be displayed using the color index !D.TABLE_SIZE-1.

Syntax

TV, *Image* [, *Position*]

or

TV, *Image* [, *X*, *Y* [, *Channel*]]

Keywords: [, /CENTIMETERS | , /INCHES] [, /ORDER] [, TRUE={1 | 2 | 3}]
[, /WORDS] [, XSIZE=*value*] [, YSIZE=*value*]

Graphics Keywords: [, CHANNEL=*value*] [, /DATA | , /DEVICE | , /NORMAL]
[, /T3D | Z=*value*]

Arguments

Image

A vector or two-dimensional array to be displayed as an image. If this argument is not already of byte type, it is converted prior to use.

X, Y

If *X* and *Y* are present, they specify the lower-left coordinate of the displayed image, relative to the lower-left corner of the screen.

Position

An integer specifying the position for *Image* within the graphics window. Image positions run from the top left of the screen to the bottom right. If a position number is used instead of *X* and *Y*, the position of the image is calculated from the dimensions of the image as follows (integer arithmetic is used).

$Xsize, Ysize =$ Size of display or window

$Xdim, Ydim =$ Dimensions of image to be displayed

$$N_x = \frac{Xsize}{Xdim} = \text{Images across screen}$$

$X = Xdim \text{Position}_{\text{modulo}N_x} =$ Starting X

$$Y = Ysize - Ydim \left[1 + \frac{\text{Position}}{N_x} \right] = \text{Starting Y}$$

For example, when displaying 128 by 128 images on a 512 by 512 display, the position numbers run from 0 to 15 as follows:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Note

When using a device with scalable pixels (e.g., PostScript), the `XSIZE` and `YSIZE` keywords should also be used.

Channel

The memory channel to be written to. The *Channel* argument is identical to the `CHANNEL` graphics keyword.

Keywords

CENTIMETERS

Set this keyword to indicate that the X , Y , $Xsize$, $Ysize$, and Z arguments are given in centimeters from the origin. This system is useful when dealing with devices, such as PostScript printers, that do not provide a direct relationship between image pixels and the size of the resulting image.

INCHES

Set this keyword to indicate that all position and size values are given in inches from the origin. This system is useful when dealing with devices, such as PostScript printers, that do not provide a direct relationship between image pixels and the size of the resulting image.

ORDER

If specified, ORDER overrides the current setting of the !ORDER system variable for the current image only. If set, the image is drawn from the top down instead of the normal bottom up.

TRUE

Set this keyword to a nonzero value to indicate that a TrueColor (16-, 24-, or 32-bit) image is to be displayed. The value assigned to TRUE specifies the index of the dimension over which color is interleaved. The image parameter must have three dimensions, one of which must be equal to three. For example, set TRUE to 1 to display an image that is pixel interleaved and has dimensions of $(3, m, n)$. Specify 2 for row-interleaved images, of size $(m, 3, n)$, and 3 for band-interleaved images of the form $(m, n, 3)$.

See “[TrueColor Images](#)” on page 4304 for an example using this keyword to write 24-bit images to the PostScript device.

WORDS

Set this keyword to indicate that words (short integers) instead of 8-bit bytes are to be transferred to the device. This keyword is valid only when using devices that can transfer 16-bit pixels. The normal transfer uses 8-bit pixels. If this keyword is set, the *Image* parameter is converted to short integer type, if necessary, and then written to the display.

XSIZE

The width of the resulting image. On devices with scalable pixel size (such as PostScript), if XSIZE is specified the image will be scaled to fit the specified width. If neither XSIZE nor YSIZE is specified, the image will be scaled to fill the plotting area, while preserving the image's aspect ratio. This keyword is ignored by pixel-based devices that are unable to change the size of their pixels.

YSIZE

The height of the resulting image. On devices with scalable pixel size (such as PostScript), if YSIZE is specified the image will be scaled to fit the specified height. If neither XSIZE nor YSIZE is specified, the image will be scaled to fill the plotting area, while preserving the image's aspect ratio. This keyword is ignored by pixel-based devices that are unable to change the size of their pixels.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above. [CHANNEL](#), [DATA](#), [DEVICE](#), [NORMAL](#), [T3D](#), [Z](#).

Examples

```

; Create and display a simple image:
D = BYTSCL(DIST(256)) & TV, D

; Erase the screen:
ERASE

; Use the position parameter to display a number of images in the
; same window.
; Display the image in the upper left corner.
TV, D, 0

; Display another copy of the image in the next position:
TV, D, 1

```

Version History

Introduced: Original

See Also

[ERASE](#), [IMAGE](#), [SLIDE_IMAGE](#), [TVRD](#), [TVSCL](#), [WIDGET_DRAW](#), [WINDOW](#)

TVCRS

The TVCRS procedure manipulates the display device cursor. The initial state of the cursor is device dependent. Call TVCRS with one argument to enable or disable the cursor. Call TVCRS with two parameters to enable the cursor and place it on pixel location (*X*, *Y*).

Syntax

TVCRS [, *ON_OFF*]

or

TVCRS [, *X*, *Y*]

Keywords: [, [/CENTIMETERS](#) | , [/INCHES](#)] [, [/HIDE_CURSOR](#)]

Graphics Keywords: [, [/DATA](#) | , [/DEVICE](#) | , [/NORMAL](#)] [, [/T3D](#) | *Z=value*]

Arguments

ON_OFF

This argument specifies whether the cursor should be on or off. If this argument is present and nonzero, the cursor is enabled. If *ON_OFF* is zero or no parameters are specified, the cursor is turned off.

X

The column to which the cursor is set.

Y

The row to which the cursor is set.

Keywords

CENTIMETERS

Set this keyword to cause *X* and *Y* to be interpreted as centimeters, based on the current device resolution.

INCHES

Set this keyword to cause X and Y to be interpreted as inches, based on the current device resolution.

HIDE_CURSOR

By default, disabling the cursor works differently for window systems than for other devices. For window systems, the cursor is restored to the standard cursor used for non-IDL windows (and remains visible), while for other devices it is completely blanked out. If the HIDE keyword is set, disabling the cursor causes it to always be blanked out.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above. [DATA](#), [DEVICE](#), [NORMAL](#), [T3D](#), [Z](#).

Examples

To enable the graphics cursor and position it at device coordinate (100, 100), enter:

```
TVCRS, 100, 100
```

To position the cursor at data coordinate (0.5, 3.2), enter:

```
TVCRS, 0.5, 3.2, /DATA
```

Version History

Introduced: Original

See Also

[CURSOR](#), [RDPIX](#)

TVLCT

The TVLCT procedure loads the display color translation tables from the specified variables. Although IDL uses the RGB color system internally, color tables can be specified to TVLCT using any of the following color systems: RGB (Red, Green, Blue), HLS (Hue, Lightness, Saturation), and HSV (Hue, Saturation, Value). Alpha values may also be used when using the second form of the command. The type and meaning of each argument is dependent upon the color system selected, as described below. Color arguments can be either scalar or vector expressions. If no color-system keywords are present, the RGB color system is used.

Syntax

```
TVLCT, V1, V2, V3 [, Start] [, /GET] [, /HLS | , /HSV]
```

or

```
TVLCT, V [, Start] [, /GET] [, /HLS | , /HSV]
```

Arguments

TVLCT will accept either three n -element vectors (V_1 , V_2 , and V_3) or a single n -by-3 array (V) as an argument. The vectors (or rows of the array) have different meanings depending on the color system chosen. If an array V is specified, $V[*],0$ is the same as V_1 , $V[*],1$ is the same as V_2 , $V[*],2$ is the same as V_3 . In the description below, we assume that three vectors, V_1 , V_2 , and V_3 are specified.

The V_1 , V_2 , and V_3 arguments have different meanings depending upon which color system they represent.

If the **GET** keyword is specified, V_1 , V_2 , and V_3 should be set to named variables.

R, G, B Color System

The parameters V_1 , V_2 , and V_3 contain the red, green, and blue values, respectively. Values are interpreted as integers in the range 0 (lowest intensity) to 255 (highest intensity). The parameters can be scalars or vectors of up to 256 elements. By default, the three arguments are assumed to be R, G, and B values.

H, L, S Color System

Parameters V_1 , V_2 , and V_3 contain the Hue, Lightness, and Saturation values respectively. All parameters are floating-point. Hue is expressed in degrees and is

reduced modulo 360. V_2 (lightness) and V_3 (saturation) and can range from 0 to 1.0. Set the HLS keyword to have the arguments interpreted this way.

H, S, V Color System

Parameters V_1 , V_2 , and V_3 contain values for Hue, Saturation, and Value (similar to intensity). All parameters are floating-point. Hue is in degrees. The Saturation and Value can range from 0 to 1.0. Set the HSV keyword to have the arguments interpreted this way.

Start

An integer value that specifies the starting point in the color translation table into which the color intensities are loaded. If this argument is not specified, a value of zero is used, causing the tables to be loaded starting at the first element of the translation tables.

Keywords

GET

Set this keyword to return the RGB values from the internal color tables into the V_1 , V_2 , and V_3 parameters. For example, the statements:

```
TVLCT, H, S, V, /HSV
TVLCT, R, G, B, /GET
```

load a color table based in the HSV system, and then read the equivalent RGB values into the variables R, G, and B.

If GET is specified, V_1 , V_2 , and V_3 must be set to named variables. A single V argument can be used with this keyword only if V is an array.

HLS

Set this keyword to indicate that the parameters specify color using the HLS color system.

HSV

Set this keyword to indicate that the parameters specify color using the HSV color system.

Examples

```
; Initialize display.  
DEVICE, DECOMPOSED = 0  
  
; Create a set of R, G, and B colormap vectors:  
R = BYTSCL(SIN(FINDGEN(256)))  
G = BYTSCL(COS(FINDGEN(256)))  
B = BINDGEN(256)  
  
; Load these vectors into the color table:  
TVLCT, R, G, B  
  
; Display an image to see the effect of the new color table:  
TVSCL, DIST(400)
```

Version History

Introduced: Original

See Also

[LOADCT](#), [XLOADCT](#), [XPALETTE](#)

TVRD

The TVRD function returns the contents of the specified rectangular portion of the current graphics window or device. (X_0, Y_0) is the coordinate of the lower left corner of the area to be read and N_x, N_y is the size of the rectangle in columns and rows. The result is a byte array of dimensions N_x by N_y . All parameters are optional. If no arguments are supplied, the entire display device area is read.

Important Note about TVRD and Backing Store

On some systems, when backing store is provided by the window system (the RETAIN keyword to DEVICE or WINDOW is set to 1), reading data from a window using TVRD may cause unexpected results. For example, data may be improperly read from the window even when the image displayed on screen is correct. Having IDL provide the backing store (set the RETAIN keyword to 2) ensures that the window contents will be read properly. More detailed notes about TVRD and the X Window system can be found below in [“Unexpected Results Using TVRD with X Windows”](#) on page 2176.

Note

When using TVRD on 16-bit TrueColor displays only 5 or 6 bits of pixel data are stored for each color channel. Using TVRD on these displays returns channel data where only the high 5 or 6 bits are significant; the other bits are indeterminate.

Syntax

$$Result = TVRD([X_0 [, Y_0 [, N_x [, N_y [, Channel]]]]) [, CHANNEL=value] [, /ORDER] [, TRUE={1 | 2 | 3}] [, /WORDS])$$

Return Value

Returns a byte array of the specified dimensions.

Arguments

X_0

The starting column of data to read. The default is 0.

Y_0

The starting row of data to read. The default is 0.

 N_x

The number of columns to read. The default is the width of the display device or window less X_0 .

 N_y

The number of rows to read. The default is the height of the display device or window less Y_0 .

Channel

The memory channel to be read. If not specified, this argument is assumed to be zero. This parameter is ignored on display systems that have only one memory channel.

Keywords**CHANNEL**

The memory channel to be read. The CHANNEL keyword is identical to the optional *Channel* argument.

Note

If the display is a 24-bit display, and both the CHANNEL and TRUE parameters are absent, the maximum RGB value in each pixel is returned.

ORDER

Set this keyword to override the current setting of the !ORDER system variable for the current image only. If set, it causes the image to be read from the top down instead of the normal bottom up.

TRUE

If this keyword is present, it indicates that a TrueColor image is to be read, if the display is capable. The value assigned to TRUE specifies the index of the dimension over which color is interleaved. The result is an $(3, n_x, n_y)$ pixel-interleaved array if TRUE is 1; or an $(n_x, 3, n_y)$ line-interleaved array if TRUE is 2; or an $(n_x, n_y, 3)$ image-interleaved array if TRUE is 3.

WORDS

Set this keyword to indicate that words are to be transferred from the device. This keyword is valid only when using devices that can transfer 16-bit pixels. The normal transfer uses 8-bit pixels. If this keyword is set, the function result is an integer array.

Unexpected Results Using TVRD with X Windows

When using TVRD with the X Windows graphics device, there are two unexpected behaviors that can be confusing to users:

- When reading from a window that is obscured by another window (i.e., the target window has another window “on top” or “in front” of it), TVRD may return the contents of the window in front as part of the image contained in the target window.
- When reading from an iconified window, the X server may return a stream of “BadMatch” protocol events.

IDL uses the Xlib function `XGetSubImage()` to implement TVRD. The following quote is from the documentation for `XGetSubImage()` found in *The X Window System* by Robert W. Scheifler and James Gettys, Second Edition, page 174. It explains the reasons for the behaviors described above:

“If the drawable is a window, the window must be viewable, and it must be the case that if there were no... overlapping windows, the specified rectangle of the window would be fully visible on the screen, ... or a BadMatch error results. If the window has backing-store, then the backing-store contents are returned for regions of the window that are obscured... If the window does not have backing-store, the returned contents of such obscured regions are undefined.”

Hence, the first behavior is caused by attempting to use TVRD on an obscured window that does not have backing store provided by the X server. The result in this case is undefined, meaning that the different servers can produce entirely different results. Many servers simply return the image of the obscuring window.

The second behavior is caused by attempting to read from a non-viewable (i.e., unmapped) window. Although IDL could refuse to allow TVRD to work with unmapped windows, some X servers return valid and useful results. Therefore, TVRD is allowed to attempt to read from unmapped windows.

Both of these behavior problems can be solved by using one of the following methods:

- Always make sure that your target window is mapped and is not obscured before using TVRD on it. The following IDL command can be used:

```
WSET, Window_Index
```

- Make IDL provide backing store (rather than the window system) by setting the RETAIN keyword to DEVICE or WINDOW equal to 2.

For a full description of backing store, see “[Backing Store](#)” on page 4286. Note that under X Windows, backing store is a request that may or may not be honored by the X server. Many servers will honor backing store for 8-bit visuals but ignore them for 24-bit visuals because they require three times as much memory.

Examples

```
; Read the entire contents of the current display device into the  
; variable T:  
T = TVRD()
```

Version History

Introduced: Original

See Also

[RDPIX](#), [TV](#), [WINDOW](#)

TVSCL

The TVSCL procedure scales the intensity values of *Image* into the range of the image display and outputs the data to the image display at the specified location. The array is scaled so the minimum data value becomes 0 and the maximum value becomes the maximum number of available colors (held in the system variable `!D.TABLE_SIZE`) as follows:

$$\text{Output} = (!D.TABLE_SIZE - 1) \frac{\text{Data} - \text{Data}_{\min}}{\text{Data}_{\max} - \text{Data}_{\min}}$$

where the maximum and minimum are found by scanning the array. The parameters and keywords of the TVSCL procedure are identical to those accepted by the TV procedure. For additional information about each parameter, consult the description of TV.

Syntax

TVSCL, *Image* [, *Position*]

or

TVSCL, *Image* [, *X*, *Y* [, *Channel*]]

Keywords: [, /CENTIMETERS | , /INCHES] [, /NAN] [, /ORDER] [, TOP=*value*] [, TRUE={1 | 2 | 3}] [, /WORDS] [, XSIZE=*value*] [, YSIZE=*value*]

Graphics Keywords: [, CHANNEL=*value*] [, /DATA | , /DEVICE | , /NORMAL] [, /T3D | Z=*value*]

Arguments

Image

A two-dimensional array to be displayed as an image. If this argument is not already of byte type, it is converted prior to use.

X, Y

If *X* and *Y* are present, they specify the lower left coordinate of the displayed image.

Position

Image position. See the discussion of the TV procedure for a full description.

Channel

The memory channel to be written. This argument is assumed to be zero if not specified. This parameter is ignored on display systems that have only one memory channel.

Keywords

TVSCL accepts all of the keywords accepted by the TV routine. See “TV” on page 2165. In addition, there are two unique keywords:

NAN

Set this keyword to cause TVSCL to treat elements of *Image* that are not numbers (that is, elements that have the special IEEE floating-point values *Infinity* or *NaN*) as missing data, and display them using color index 0 (zero). Note that color index 0 is also used to display elements that have the minimum value in the *Image* array. (See “Special Floating-Point Values” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

TOP

The maximum value of the scaled result. If TOP is not specified, !D.TABLE_SIZE-1 is used. Note that the minimum value of the scaled result is always 0.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above. CHANNEL, DATA, DEVICE, NORMAL, T3D, Z.

Thread Pool Keywords

This routine is written to make use of IDL’s *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

Display a floating-point array as an image using the TV command:

```
TV, DIST(200)
```

Note that the image is not easily visible because the values in the array have not been scaled into the full range of display values. Now display the image with the TVSCL command by entering:

```
TVSCL, DIST(200)
```

Notice how much brighter the image appears.

Version History

Introduced: Original

See Also

[ERASE](#), [SLIDE_IMAGE](#), [TV](#), [WIDGET_DRAW](#), [WINDOW](#)

UINDGEN

The UINDGEN function creates an unsigned integer array. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = UINDGEN(D_1 [, \dots, D_8])$$

Return Value

Returns an unsigned integer array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create UI, a 10-element by 10-element 16-bit array where each element is set to the value of its one-dimensional subscript, enter:

```
UI = UINDGEN(10, 10)
```

Version History

Introduced: 5.2

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [L64INDGEN](#),
[LINDGEN](#), [SINDGEN](#), [UL64INDGEN](#), [ULINDGEN](#)

UINT

The `UINT` function returns a result equal to *Expression* converted to unsigned integer type.

Syntax

$$\text{Result} = \text{UINT}(\text{Expression}[, \text{Offset} [, D_1 [, \dots, D_8]]])$$

Return Value

Returns the specified dimensions of the given expression as an unsigned integer type.

Arguments

Expression

The expression to be converted to unsigned integer.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as unsigned integer data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid integer and no conversion is possible. The default action in such cases is to print a warning message and return 0. The `ON_IOERROR` procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

If A contains the floating-point value 32000.0, it can be converted to an unsigned integer and stored in the variable B by entering:

```
B = UINT(A)
```

Version History

Introduced: 5.2

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#), [LONG64](#), [STRING](#), [ULONG](#), [ULONG64](#)

UINTARR

The `UINTARR` function creates an unsigned integer vector or array.

Syntax

$$\text{Result} = \text{UINTARR}(D_1 [, \dots, D_g] [, \text{NOZERO}])$$

Return Value

Returns an unsigned integer vector or array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, `UINTARR` sets every element of the result to zero. If `NOZERO` is set, this zeroing is not performed and `UINTARR` executes faster.

Examples

To create `L`, a 100-element, unsigned integer vector with each element set to 0, enter:

```
L = UINTARR(100)
```

Version History

Introduced: 5.2

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#),
[LON64ARR](#), [LONARR](#), [MAKE_ARRAY](#), [STRARR](#), [ULON64ARR](#), [ULONARR](#)

UL64INDGEN

The UL64INDGEN function returns an unsigned 64-bit integer array with the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$\text{Result} = \text{UL64INDGEN}(D_1 [, \dots, D_8])$$

Return Value

Returns the specified unsigned integer array.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create L, a 10-element by 10-element 64-bit array where each element is set to the value of its one-dimensional subscript, enter:

```
L = UL64INDGEN(10, 10)
```

Version History

Introduced: 5.2

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [L64INDGEN](#),
[LINDGEN](#), [SINDGEN](#), [UINDGEN](#), [ULINDGEN](#)

ULINDGEN

The ULINDGEN function returns an unsigned longword array with the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

Syntax

$$Result = ULINDGEN(D_1 [, \dots, D_8])$$

Return Value

Returns the specified unsigned longword array.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified. If the dimension arguments are not integer values, IDL will convert them to integer values before creating the new array.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

To create L, a 10-element by 10-element 32-bit array where each element is set to the value of its one-dimensional subscript, enter:

```
L = ULINDGEN(10, 10)
```

Version History

Introduced: 5.2

See Also

[BINDGEN](#), [CINDGEN](#), [DCINDGEN](#), [DINDGEN](#), [FINDGEN](#), [L64INDGEN](#), [LINDGEN](#), [SINDGEN](#), [UINDGEN](#), [UL64INDGEN](#)

ULON64ARR

The ULON64ARR function returns an unsigned 64-bit integer vector or array.

Syntax

$$Result = ULON64ARR(D_1 [, \dots, D_8] [, NOZERO])$$

Return Value

Returns a vector or array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, ULON64ARR sets every element of the result to zero. If NOZERO is set, this zeroing is not performed and ULON64ARR executes faster.

Examples

To create L, a 100-element, unsigned 64-bit vector with each element set to 0, enter:

```
L = ULON64ARR(100)
```

Version History

Introduced: 5.2

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#),
[LON64ARR](#), [LONARR](#), [MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULONARR](#)

ULONARR

The ULONARR function returns an unsigned longword integer vector or array.

Syntax

$$Result = ULONARR(D_1 [, \dots, D_8] [, /NOZERO])$$

Return Value

Returns an unsigned longword integer array of the specified dimensions.

Arguments

D_i

Either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

Keywords

NOZERO

Normally, ULONARR sets every element of the result to zero. If NOZERO is set, this zeroing is not performed and ULONARR executes more quickly.

Examples

To create L, a 100-element, unsigned longword vector with each element set to 0, enter:

```
L = ULONARR(100)
```

Version History

Introduced: 5.2

See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [DCOMPLEXARR](#), [FLTARR](#), [INTARR](#),
[LON64ARR](#), [LONARR](#), [MAKE_ARRAY](#), [STRARR](#), [UINTARR](#), [ULON64ARR](#),

ULONG

The ULONG function returns a result equal to *Expression* converted to the unsigned longword integer type.

Syntax

$$\text{Result} = \text{ULONG}(\text{Expression}[, \text{Offset} [, D_1 [, \dots, D_8]]])$$

Return Value

Returns the specified dimensions of the given expression as an unsigned longword integer type.

Arguments

Expression

The expression to be converted to unsigned longword integer.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as unsigned longword integer data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid integer and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

If A contains the floating-point value 32000.0, it can be converted to an unsigned longword integer and stored in the variable B by entering:

```
B = ULONG(A)
```

Version History

Introduced: 5.2

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#), [LONG64](#), [STRING](#), [UINT](#), [ULONG64](#)

ULONG64

The ULONG64 function returns a result equal to *Expression* converted to the unsigned 64-bit integer type.

Syntax

$$\text{Result} = \text{ULONG64}(\text{Expression}[, \text{Offset} [, D_1 [, \dots, D_8]]])$$

Return Value

Returns the specified dimensions of the given expression as an unsigned 64-bit integer type.

Arguments

Expression

The expression to be converted to unsigned 64-bit integer.

Offset

Offset from beginning of the *Expression* data area. Specifying this argument allows fields of data extracted from *Expression* to be treated as unsigned 64-bit integer data. See the description in [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for details.

D_i

When extracting fields of data, the D_i arguments specify the dimensions of the result. If no dimension arguments are given, the result is taken to be scalar.

The D_i arguments can be either an array or a series of scalar expressions specifying the dimensions of the result. If a single argument is specified, it can be either a scalar expression or an array of up to eight elements. If multiple arguments are specified, they must all be scalar expressions. Up to eight dimensions can be specified.

When converting from a string argument, it is possible that the string does not contain a valid integer and no conversion is possible. The default action in such cases is to print a warning message and return 0. The ON_IOERROR procedure can be used to establish a statement to be jumped to in case of such errors.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, “Thread Pool Keywords”](#) for details.

Examples

If A contains the floating-point value 32000.0, it can be converted to an unsigned 64-bit integer and stored in the variable B by entering:

```
B = ULONG64(A)
```

Version History

Introduced: 5.2

See Also

[BYTE](#), [COMPLEX](#), [DCOMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#), [LONG64](#), [STRING](#), [UINT](#), [ULONG](#)

UNIQ

The UNIQ function returns the subscripts of the unique elements in an array. Note that repeated elements must be adjacent in order to be found. This routine is intended to be used with the SORT function: see the examples below. This function was inspired by the UNIX `uniq(1)` command.

This routine is written in the IDL language. Its source code can be found in the file `uniq.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = UNIQ(*Array* [, *Index*])

Return Value

UNIQ returns an array of indices into the original array. Note that the index of the last element in each set of non-unique elements is returned. If the optional *Index* argument is supplied, then the last index in the order provided by the *Index* array is returned.

The following expression returns a copy of the sorted array with duplicate adjacent elements removed:

```
Array[UNIQ(Array)]
```

UNIQ returns 0 (zero) if the argument supplied is a scalar rather than an array.

Arguments

Array

The array to be scanned. For UNIQ to work properly, the array must be sorted into monotonic order unless the optional parameter *Index* is supplied.

Index

This optional parameter is an array of indices into *Array* that order the elements into monotonic order. That is, the expression:

```
Array[Index]
```

yields an array in which the elements of *Array* are rearranged into monotonic order.

The *Index* array can be generated from *Array* using the SORT function:

```
Index = SORT(Array)
```

If *Array* is not already in monotonic order, use the command:

```
UNIQ(Array, SORT(Array))
```

Examples

Find the unique elements of an unsorted array:

```
; Create an array:
array = [1, 2, 1, 2, 3, 4, 5, 6, 6, 5]

; Variable B holds an array containing the sorted, unique values in
; array:
b = array[UNIQ(array, SORT(array))]
PRINT, b
```

IDL prints

```
1  2  3  4  5  6
```

Version History

Introduced: Pre 4.0

See Also

[SORT, WHERE](#)

UNSHARP_MASK

The UNSHARP_MASK function performs an unsharp-mask sharpening filter on a two-dimensional array or a TrueColor image. For TrueColor images the unsharp mask is applied to each channel.

The unsharp mask algorithm works by enhancing the contrast between neighboring pixels in an image, and is widely used for astronomical images and for digital photographs.

The algorithm involves the following steps:

1. Smooth the original image with a Gaussian filter, with the width controlled by the RADIUS keyword.
2. Subtract the smoothed image from the original to create a high-pass filtered image.
3. Apply any clipping needed on the high-pass image, as controlled by the THRESHOLD keyword.
4. Add a certain percentage of the high-pass filtered image to the original image, with the percentage controlled by the AMOUNT keyword.

In pseudocode this algorithm can be written as:

```
HighPass = Image - Convolve ( Image, Gaussian )
Result = Image + A * HighPass * ( |HighPass| ≥ T )
```

where A is the amount, T is the threshold, and \geq indicates a Boolean operation, 1 if true, or 0 otherwise.

Note

To avoid overflow for byte or integer data, the computations are performed using a larger integer type, the result is clipped to the minimum and maximum values for the original type, and the result is then converted back to the original type.

This routine is written in the IDL language. Its source code can be found in the file `unsharp_mask.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = UNSHARP_MASK(Image [, AMOUNT=value] [, RADIUS=value]
  [, THRESHOLD=value] [, TRUE={1|2|3}])
```

Return Value

The *Result* is an array with the same dimensions and type as *Image* that contains the filtered image.

Arguments

Image

The two-dimensional array or multichannel image to be filtered. If *Image* is a multichannel image, then the TRUE keyword may be used to indicate which dimension represents the channels.

Keywords

AMOUNT

Set this keyword to a floating-point value giving the amount (or strength) of filtering to be applied. The default is AMOUNT=1.0, which implies that 100% of the filter difference will be applied to the *Image*. AMOUNT may be negative, which will have the effect of blurring the image instead of sharpening.

RADIUS

Set this keyword to a floating-point value giving the radius in pixels of the Gaussian smoothing filter. The default is RADIUS=3. The Gaussian filter is designed to fall to $1/e$ (≈ 0.367) at a distance of RADIUS/sqrt(2). The total width of the Gaussian filter is given by $\text{ceil}(2 * \text{RADIUS})/2 * 2 + 1$ (if RADIUS is an integer then this is just $2 * \text{RADIUS} + 1$).

Note

The value of RADIUS must be smaller than one-half of the largest array dimension.

Tip

Use small RADIUS values (such as 3) for small images or images with fine details.
Use larger RADIUS values for large images with larger details.

THRESHOLD

Set this keyword to a non-negative integer (or a floating-point if *Image* is floating point) giving the clipping threshold. For each element, if the absolute value of the difference between the original image and the low-pass filtered array is greater than

or equal to THRESHOLD then the filter is applied to that point. The default is THRESHOLD=0, which implies that every point will be filtered.

Tip

Lower values of THRESHOLD will provide greater sharpening but may cause more speckling, while higher values of THRESHOLD will exclude regions of low contrast and cause less speckling.

TRUE

If *Image* is a three-dimensional array (a multichannel image), set this keyword to 1, 2, or 3 to indicate which dimension represents the channels. The default is 1, for pixel interleaving, $(3, m, n)$. A value of 2 indicates line interleaving $(m, 3, n)$, and 3 indicates band interleaving, $(m, n, 3)$.

Example

```
file = FILEPATH('marsglobe.jpg', SUBDIR=['examples','data'])
READ_JPEG, file, image
; Display the original image.
iImage, image, VIEW_GRID=[2,1], DIMENSIONS = [800, 500]
; Display the unsharp-mask-filtered image.
result = UNSHARP_MASK(image, RADIUS=4)
iImage, result, /VIEW_NEXT
```

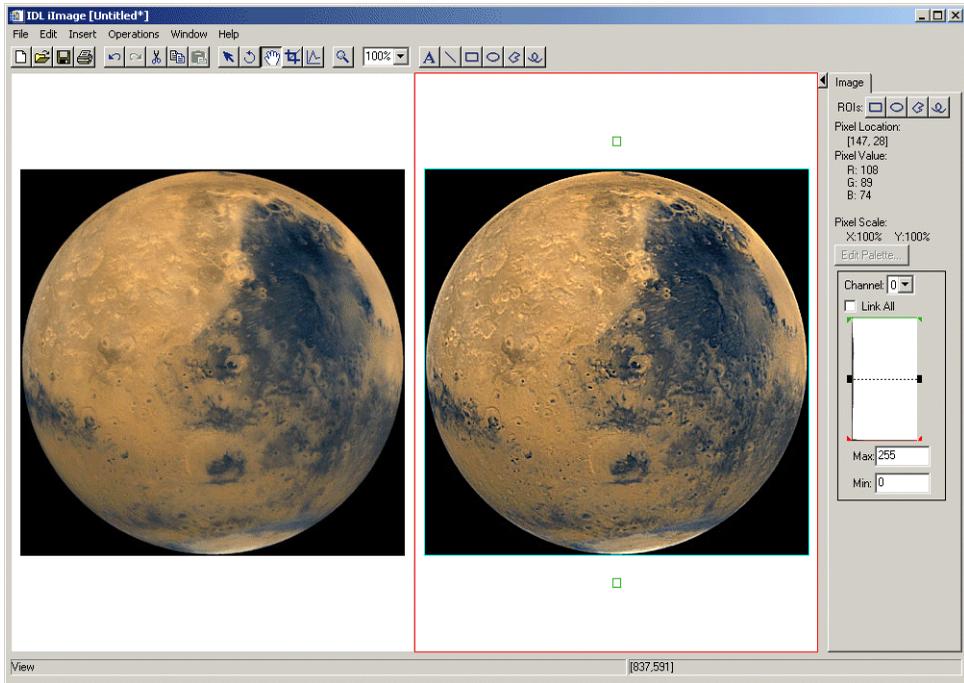


Figure 3-65: Mars Viewed in `image` with `UNSHARP_MASK`

Version History

Introduced: 6.1

See Also

[CONVOL](#), [LEEFILT](#), [ROBERTS](#), [SOBEL](#)

USERSYM

The USERSYM procedure is used to define the plotting symbol that marks points when the plotting symbol is set to plus or minus 8. Symbols can be drawn with vectors or can be filled. Symbols can be of any size and can have up to 49 vertices. See “[Defining Your Own Plotting Symbols](#)” in Chapter 17 of the *Using IDL* manual .

Syntax

```
USERSYM, X [, Y] [, COLOR=value] [, /FILL] [, THICK=value]
```

Arguments

X, Y

The X and/or Y parameters define the vertices of the symbol as offsets from the data point in units of approximately the size of a character. In the case of a vector drawn symbol, the symbol is formed by connecting the vertices in order. If only one argument is specified, it must be a (2, N) array of vertices, with element [0, i] containing the X coordinate of the vertex, and element [1, i] containing the Y. If both arguments are provided, X contains only the X coordinates.

Keywords

COLOR

The color used to draw the symbols, or used to fill the polygon. The default color is the same as the line color.

FILL

Set this keyword to fill the polygon defined by the vertices. If FILL is not set, lines are drawn connecting the vertices.

THICK

The thickness of the lines used in drawing the symbol. The default thickness is 1.0.

Examples

Make a large, diamond-shaped plotting symbol. Define the vectors of X values by entering:

```
X = [-6, 0, 6, 0, -6]
```

Define the vectors of Y values by entering:

```
Y = [0, 6, 0, -6, 0]
```

Now call `USERSYM` to create the new plotting symbol 8. Enter:

```
USERSYM, X, Y
```

Generate a simple plot to test the plotting symbol by entering:

```
PLOT, FINDGEN(20), PSYM = 8
```

Version History

Introduced: Original

See Also

[PLOT](#)

VALUE_LOCATE

The VALUE_LOCATE function finds the intervals within a given monotonic vector that brackets a given set of one or more search values. This function is useful for interpolation and table-lookup, and is an adaptation of the locate() routine in Numerical Recipes. VALUE_LOCATE uses the bisection method to locate the interval.

Syntax

$Result = \text{VALUE_LOCATE} (\text{Vector}, \text{Value} [, /L64])$

Return Value

Each return value, $Result [i]$, is an index, j , into $Vector$, corresponding to the interval into which the given $Value [i]$ falls. The returned values are in the range $-1 \leq j \leq N-1$, where N is the number of elements in the input vector.

If $Vector$ is monotonically increasing, the result j is:

if $j = -1$ $Value [i] < Vector [0]$
 if $0 \leq j < N-1$ $Vector [j] \leq Value [i] < Vector [j+1]$
 if $j = N-1$ $Vector [N-1] \leq Value [i]$

If $Vector$ is monotonically decreasing

if $j = -1$ $Vector [0] \leq Value [i]$
 if $0 \leq j < N-1$ $Vector [j+1] \leq Value [i] < Vector [j]$
 if $j = N-1$ $Value [i] < Vector [N-1]$

Arguments

Vector

A vector of monotonically increasing or decreasing values. $Vector$ may be of type string, or any numeric type except complex, and may not contain the value NaN (not-a-number).

Value

The value for which the location of the intervals is to be computed. *Value* may be either a scalar or an array. The return value will contain the same number of elements as this parameter.

Keywords

L64

By default, the result of `VALUE_LOCATE` is 32-bit integer when possible, and 64-bit integer if the number of elements being processed requires it. Set `L64` to force 64-bit integers to be returned in all cases.

Note

Only 64-bit versions of IDL are capable of creating variables requiring a 64-bit result. Check the value of `!VERSION.MEMORY_BITS` to see if your IDL is 64-bit or not.

Examples

```
; Define a vector of values.
vec = [2,5,8,10]

; Compute location of other values within that vector.
loc = VALUE_LOCATE(vec, [0,3,5,6,12])
PRINT, loc
```

IDL prints:

```
-1  0  1  1  3
```

Version History

Introduced: 5.3

VARIANCE

The VARIANCE function computes the statistical variance of an n -element vector.

Syntax

```
Result = VARIANCE( X [, /DOUBLE] [, /NAN] )
```

Return Value

Returns the floating-point or double-precision statistical variance of the input vector.

Arguments

X

An n -element, floating-point or double-precision vector.

Keywords

DOUBLE

If this keyword is set, VARIANCE performs its computations in double precision arithmetic and returns a double precision result. If this keyword is not set, the computations and result depend upon the type of the input data (integer and float data return float results, while double data returns double results).

NAN

Set this keyword to cause the routine to check for occurrences of the IEEE floating-point values *NaN* or *Infinity* in the input data. Elements with the value *NaN* or *Infinity* are treated as missing data. (See “[Special Floating-Point Values](#)” in Chapter 19 of the *Building IDL Applications* manual for more information on IEEE floating-point values.)

Examples

```
; Define the n-element vector of sample data:  
x = [1, 1, 1, 2, 5]  
; Compute the variance:  
result = VARIANCE(x)  
PRINT, result
```

IDL prints:

3.00000

Version History

Introduced: 5.1

See Also

[KURTOSIS](#), [MEAN](#), [MEANABSDEV](#), [MOMENT](#), [STDDEV](#), [SKEWNESS](#)

VECTOR_FIELD

The VECTOR_FIELD procedure is used to place colored, oriented vectors of specified length at each vertex in an input vertex array. The output can be sent directly to an IDLgrPolyline object. The generated display is generally referred to as a hedgehog display and is used to convey various aspects of a vector field.

Syntax

```
VECTOR_FIELD, Field, Outverts, Outconn [, ANISOTROPY=array]  
[, SCALE=value] [, VERTICES=array]
```

Arguments

Field

Input vector field array. This can be a [3, x , y , z] array or a [2, x , y] array. The leading dimension is the vector quantity to be displayed.

Outverts

Output vertex array ([3, N] or [2, N] array of floats). Useful if the routine is to be used with Direct Graphics or the user wants to manipulate the data directly.

Outconn

Output polyline connectivity array to be applied to the output vertices.

Keywords

ANISOTROPY

Set this keyword to a two- or three-element array describing the distance between grid points in each dimension. The default value is [1.0, 1.0, 1.0] for three-dimensional data and [1.0, 1.0] for two-dimensional data.

SCALE

Set this keyword to a scalar scaling factor. All vector lengths are multiplied by this value. The default is 1.0.

VERTICES

Set this keyword to a $[3, n]$ or $[2, n]$ array of points. If this keyword is set, the vector field is interpolated at these points. The resulting interpolated vectors are displayed as line segments at these locations. If the keyword is not set, each spatial sample point in the input Field grid is used as the base point for a line segment.

Version History

Introduced: 5.3

VEL

The VEL procedure draws a velocity (flow) field with arrows following the field proportional in length to the field strength. Arrows are composed of a number of small segments that follow the streamlines.

This routine is written in the IDL language. Its source code can be found in the file `vel.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
VEL, U, V [, NVECS=value] [, XMAX= value{ysize/ysize}]  
    [, LENGTH=value{longest/steps}] [, NSTEPS=value] [, TITLE=string]
```

Arguments

U

The X component at each point of the vector field. *U* must be a 2-D array.

V

The Y component at each point of the vector field. *V* must have the same dimensions as *U*.

Keywords

LENGTH

The length of each arrow line segment expressed as a fraction of the longest vector divided by the number of steps. The default is 0.1.

NSTEPS

The number of shoots or line segments for each arrow. The default is 10.

NVECS

The number of vectors (arrows) to draw. If this keyword is omitted, 200 vectors are drawn.

TITLE

A string containing the title for the plot.

XMAX

X axis size as a fraction of Y axis size. The default is 1.0. This argument is ignored when !P.MULTI is set.

Examples

```
; Create a vector of X values:  
X = DIST(20)  
  
; Create a vector of Y values:  
Y = SIN(X)*100  
  
; Plot the vector field:  
VEL, X, Y
```

Version History

Introduced: Pre 4.0

See Also

[FLOW3](#), [PLOT_FIELD](#), [VELOVECT](#)

VELOVECT

The VELOVECT procedure produces a two-dimensional velocity field plot. A directed arrow is drawn at each point showing the direction and magnitude of the field.

This routine is written in the IDL language. Its source code can be found in the file `velovect.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
VELOVECT, U, V [, X, Y] [, COLOR=index] [, MISSING=value] [, /DOTS]]  
      [, LENGTH=value] [, /OVERPLOT] [Also accepts all PLOT keywords]
```

Arguments

U

The X component of the two-dimensional field. *U* must be a two-dimensional array.

V

The Y component of the two dimensional field. *V* must have the same dimensions as *U*.

X

Optional abscissae values. *X* must be a vector with a length equal to the first dimension of *U* and *V*.

Y

Optional ordinate values. *Y* must be a vector with a length equal to the second dimension of *U* and *V*.

Keywords

Note

Keywords not described here are passed directly to the PLOT procedure and may be used to set options such as TITLE, POSITION, NOERASE, etc.

COLOR

Set this keyword equal to the color index used for the plot.

DOTS

Set this keyword to 1 to place a dot at each missing point. Set this keyword to 0 or omit it to draw nothing for missing points. Has effect only if MISSING is specified.

LENGTH

Set this keyword equal to the length factor. The default of 1.0 makes the longest (U, V) vector the length of a cell.

MISSING

Set this keyword equal to the missing data value. Vectors with a length greater than MISSING are ignored.

OVERPLOT

Set this keyword to make VELOVECT “overplot”. That is, the current graphics screen is not erased, no axes are drawn, and the previously established scaling remains in effect.

PLOT Keywords

In addition to the keywords described above, all other keywords accepted by the PLOT procedure are accepted by VELOVECT. See [PLOT](#).

Examples

```

; Create some random data:
U = RANDOMN(S, 20, 20)
V = RANDOMN(S, 20, 20)

; Plot the vector field:
VELOVECT, U, V

; Plot the field, using dots to represent vectors with values
; greater than 18:
VELOVECT, U, V, MISSING=18, /DOTS

; Plot with a title. Note that the XTITLE keyword is passed
; directly to the PLOT procedure:
VELOVECT, U, V, MISSING=18, /DOTS, XTITLE='Random Vectors'

```

Version History

Introduced: Original

See Also

[FLOW3](#), [PLOT](#), [PLOT_FIELD](#), [VEL](#)

VERT_T3D

The VERT_T3D function transforms a 3-D array by a 4x4 transformation matrix. The 3-D points are typically an array of polygon vertices that were generated by SHADE_VOLUME or MESH_OBJ.

This routine is written in the IDL language. Its source code can be found in the file `vert_t3d.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = VERT_T3D( Vertex_List [, DOUBLE=value] [, MATRIX=4x4_array]  
[, /NO_COPY] [, /NO_DIVIDE [, SAVE_DIVIDE=variable]] )
```

Return Value

Returns the single- or double-precision coordinates of the transformed array.

Arguments

Vertex_List

A 3 x *n* array of 3-D coordinates to transform.

Keywords

DOUBLE

Set this keyword to a nonzero value to indicate that the returned coordinates should be double-precision. If this keyword is not set, the default is to return single-precision coordinates (unless double-precision arguments are input, in which case the DOUBLE keyword is implied to be non-zero).

MATRIX

The 4x4 transformation matrix to use. The default is to use the system viewing matrix (!P.T).

NO_COPY

Normally, a copy of *Vertex_list* is transformed and the original *Vertex_list* is preserved. If NO_COPY is set, however, then the original *Vertex_List* will be undefined after the call to VERT_T3D. Using the NO_COPY requires less memory.

NO_DIVIDE

Normally, when a $[x, y, z, 1]$ vector is transformed by a 4x4 matrix, the final homogeneous coordinates are obtained by dividing the x , y , and z components of the result vector by the fourth element in the result vector. Setting the NO_DIVIDE keyword will prevent VERT_T3D from performing this division. In some cases (usually when a perspective transformation is involved) the fourth element in the result vector can be very close to (or equal to) zero.

SAVE_DIVIDE

Set this keyword to a named variable that will hold receive the fourth element of the transformed vector(s). If *Vertex_list* is a vector then SAVE_DIVIDE is a scalar. If *Vertex_list* is an array then SAVE_DIVIDE is an array of n elements. This keyword only has effect when the NO_DIVIDE keyword is set.

Examples

Transform four points representing a square in the x-y plane by first translating +2.0 in the positive X direction, and then rotating 60.0 degrees about the Y axis.

```
points = [[0.0, 0.0, 0.0], [1.0, 0.0, 0.0], $
          [1.0, 1.0, 0.0], [0.0, 1.0, 0.0]]
T3D, /RESET
T3D, TRANSLATE=[2.0, 0.0, 0.0]
T3D, ROTATE=[0.0, 60.0, 0.0]
points = VERT_T3D(points)
```

Version History

Introduced: Pre 4.0

See Also

[T3D](#)

VOIGT

The VOIGT function returns the value of the classical Voigt function, $H(a, u)$, defined in terms of the Voigt damping parameter a and the frequency offset u :

$$H(a, u) = \frac{a}{\pi} \int_{-\infty}^{\infty} \frac{e^{-y^2} dy}{a^2 + (u - y)^2}$$

The dimensionless frequency offset u and the damping parameter a are determined by:

$$u = \frac{v - v_0}{\Delta v_D}$$

$$a = \frac{\Gamma}{4\pi\Delta v_D}$$

where v_0 is the line center frequency and the Doppler width Δv_D (assuming no turbulence), is defined as:

$$\Delta v_D = \frac{v_0}{c} b = \frac{v_0}{c} \sqrt{2kT/m}$$

and, Γ is the transition rate:

$$\Gamma = \gamma + 2v_{col}$$

where γ is the spontaneous decay rate, and v_{col} is the atomic collision rate. (See *Radiative Processes in Astrophysics* by G. B. Rybicki and A. P. Lightman (1979) p. 291 for more information.)

The Voigt function can be used to compute the intensity of an atomic absorption line profile (also known as a VOIGT profile). The line profile $\phi(a, u)$ is defined as:

$$\phi(a, u) \equiv \frac{H(a, u)}{\Delta v_D \sqrt{\pi}}$$

The algorithm is from Armstrong, *JQSRT* 7, 85. (1967). The definition of the classical Voigt function $H(a,u)$ can be found in Equation 7.4.13 of Abramowitz, M. and Stegun, I.A., 1964, *Handbook of Mathematical Functions* (Washington:National Bureau of Standards).

Syntax

Result = VOIGT(*A*, *U*)

Return Value

If both arguments are scalars, the function returns a scalar. If both arguments are arrays, the function matches up the corresponding elements of *A* and *U*, returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the smallest input array. If *A* is double-precision, the result is double-precision, otherwise the result is single-precision.

Arguments

A

A scalar or array specifying the values for the Voigt damping parameter.

U

A scalar or array specifying the values for the dimensionless frequency offset in Doppler widths.

Keywords

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Version History

Introduced: Pre 4.0

See Also

[LEEFILT](#), [ROBERTS](#), [SOBEL](#)

VORONOI

The VORONOI procedure computes the Voronoi polygon of a point within an irregular grid of points, given the Delaunay triangulation. The Voronoi polygon of a point contains the region closer to that point than to any other point.

For interior points, the polygon is constructed by connecting the midpoints of the lines connecting the point with its Delaunay neighbors. Polygons are traversed in a counterclockwise direction.

For exterior points, the set is described by the midpoints of the connecting lines, plus the circumcenters of the two triangles that connect the point to the two adjacent exterior points.

This routine is written in the IDL language. Its source code can be found in the file `voronoi.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

VORONOI, X, Y, I0, C, Xp, Yp, Rect

Arguments

X

An array containing the X locations of the points.

Y

An array containing the Y locations of the points.

I0

An array containing the indices of the points.

C

A connectivity list from the Delaunay triangulation. This list is produced with the CONNECTIVITY keyword of the TRIANGULATE procedure.

Xp, Yp

Named variables that will contain the X and Y vertices of Voronoi polygon.

Rect

The bounding rectangle: [Xmin, Ymin, Xmax, Ymax]. Because the Voronoi polygon (VP) for points on the convex hull extends to infinity, a clipping rectangle must be supplied to close the polygon. This rectangle has no effect on the VP of interior points. If this rectangle does not enclose all the Voronoi vertices, the results will be incorrect. If this parameter, which must be a named variable, is undefined or set to a scalar value, it will be calculated.

Examples

To draw the Voronoi polygons of each point of an irregular grid:

```
N = 20

; Create a random grid of N points:
X = RANDOMU(seed, N)
Y = RANDOMU(seed, N)

; Triangulate it:
TRIANGULATE, X, Y, tr, CONN=C

FOR I=0, N-1 DO BEGIN & $
  ; Get the ith polygon:
  VORONOI, X, Y, I, C, Xp, Yp & $
  ; Draw it:
  POLYFILL, Xp, Yp, COLOR = (I MOD 10) + 2 & $
ENDFOR
```

Version History

Introduced: Pre 4.0

See Also

[TRIANGULATE](#)

VOXEL_PROJ

The VOXEL_PROJ function generates visualizations of volumetric data by computing 2-D projections of a colored, semi-transparent volume. Parallel rays from any given direction are cast through the volume, onto the viewing plane. User-selected colors and opacities can be assigned to arbitrary data ranges, simulating the appearance of the materials contained within the volume.

The VOXEL_PROJ function can be combined with the Z-buffer to render volume data over objects. Cutting planes can also be specified to view selected portions of the volume. Other options include: selectable resolution to allow quick “preview” renderings, and average and maximum projections.

VOXEL_PROJ renders volumes using an algorithm similar to the one described by Drebin, Carpenter, and Hanrahan, in “Volume Rendering”, *Computer Graphics*, Volume 22, Number 4, August 1988, pp. 125-134, but without the surface extraction and enhancement step.

Voxel rendering can be quite time consuming. The time required to render a volume is proportional to the viewing areas size, in pixels, times the thickness of the volume cube in the viewing direction, divided by the product of the user-specified X, Y, and Z steps.

Note

It is necessary to use SCALE3 to initialize a display before calling VOXEL_PROJ as shown in the following [Examples](#) section.

Syntax

```
Result = VOXEL_PROJ( V [, RGBO] [, BACKGROUND=array]
  [, CUTTING_PLANE=array] [, /INTERPOLATE]
  [, /MAXIMUM_INTENSITY] [, STEP={Sx, Sy, Sz}] [, XSIZE=pixels]
  [, YSIZE=pixels] [, ZBUFFER=int_array] [, ZPIXELS=byte_array] )
```

Return Value

Returns the specified voxel projection results.

Arguments

V

A three-dimensional array containing the volume to be rendered. This array is converted to byte type if necessary.

RGBO

This optional parameter is used to specify the look-up tables that indicate the color and opacity of each voxel value. This argument can be one of the following types:

- A (256, 4) byte array for TrueColor rendering. This array represents 256 sets of red, green, blue, and opacity (RGBO) components for each voxel value, scaled into the range of bytes (0 to 255). The R, G, and B components should already be scaled by the opacity. For example, if a voxel value of 100 contains a material that is red, and 35% opaque, the RGBO values should be, respectively: [89, 0, 0, 89] because $255 * 0.35 = 89$. If more than one material is present, the RGBO arrays contain the sum of the individual RGBO arrays. The content and shape of the RGBO curves is highly dependent upon the volume data and experimentation is often required to obtain the best display.
- A (256, 2) byte array for volumes with only one material or monochrome rendering. This array represents 256 sets of pixel values and their corresponding opacities for each voxel value.
- If this argument is omitted, the average projection method, or maximum intensity method (if the `MAXIMUM_INTENSITY` keyword is set) is used.

Keywords

BACKGROUND

A one- or three-element array containing the background color indices. The default is (0,0,0), yielding a black background with most color tables.

CUTTING_PLANE

A floating-point array specifying the coefficients of additional cutting planes. The array has dimensions of (4, N), where N is the number of additional cutting planes from 1 to 6. Cutting planes are constraints in the form of:

$$C[0] * X + C[1] * Y + C[2] * Z + D > 0$$

The X, Y, and Z coordinates are specified in voxel coordinates. For example, to specify a cutting plane that excludes all voxels with an X value greater than 10:

CUTTING_PLANE = [-1.0, 0, 0, 10.], for the constraint: $-X + 10 > 0$.

INTERPOLATE

Set this keyword to use tri-linear interpolation to determine the data value for each step on a ray. Otherwise, the nearest-neighbor method is used. Setting this keyword improves the quality of images produced, especially when the volume has low resolution in relation to the size of the viewing plane, at the cost of more computing time.

MAXIMUM_INTENSITY

Set this keyword to make the value of each pixel in the viewing plane the maximum data value along the corresponding ray. The *RGBO* argument is ignored if present.

STEP

Set this keyword to a three-element vector, [S_x , S_y , S_z], that controls the resolution of the resulting projection. The first two elements contain the step size in the X and Y view plane, in pixels. The third element is the sampling step size in the Z direction, given in voxels. S_x and S_y must be integers equal to or greater than one, while S_z can contain a fractional part. If S_x or S_y are greater than one, the values of intermediate pixels in the output image are linearly interpolated. Higher step sizes require less time because fewer rays are cast, at the expense of lower resolution in the output image.

XSIZE

The width, in pixels, of the output image. If this keyword is omitted, the output image is as wide as the currently-selected output device.

YSIZE

The height, in pixels, of the output image. If this keyword is omitted, the output image is as tall as the currently selected output device.

ZBUFFER

An integer array, with the same width and height as the output image, that contains the depth portion of the Z-buffer. Include this parameter to combine the previously-read contents of a Z-buffer with a voxel rendering. See the third example, below, for details.

ZPIXELS

A byte array, with the same width and height as the output image, that contains the image portion of the Z-buffer. Include this parameter to combine the contents of a Z-buffer with a voxel rendering. See the third example, below, for details.

Examples

Example 1

In the following example, assume that variable `V` contains a volume of data, with dimensions V_x by V_y by V_z . The volume contains two materials, muscle tissue represented by a voxel range of 50 to 70, that we want to render with red color, and an opacity of 20; and bone tissue represented by a voxel range of 220-255, that we want to render with white color, and an opacity of 50:

```
; Create the opacity vector:
rgbo = BYTARR(256,4)

; Red and opacity for muscle:
rgbo[50:70, [0,3]] = 20

; White and opacity for bone:
rgbo[220:255, *] = 50
```

Example 2

Although it is common to use trapezoidal or Gaussian functions when forming the RGBO arrays, this example uses rectangular functions for simplicity.

```
; Set up the axis scaling and default rotation:
SCALE3, XRANGE=[0, Vx-1], YRANGE=[0, Vy-1], ZRANGE=[0, Vz-1]

; Compute projected image:
C = VOXEL_PROJ(V, rgbo)

; Convert from 24-bit to 8-bit image and display:
TV, COLOR_QUAN(C, 3, R, G, B)

; Load quantized color tables:
TVLCT, R, G, B
```

This example required approximately 27 seconds on a typical workstation to compute the view in a 640- by 512-pixel viewing window. Adding the keyword `STEP=[2,2,1]` in the call to `VOXEL_PROJ` decreased the computing time to about 8 seconds, at the expense of slightly poorer resolution.

When viewing a volume with only one constituent, the RGBO array should contain only an intensity/opacity value pair. To illustrate, if in the above example, only muscle was of interest we create the RGBO argument as follows:

```

; Create an empty 256 x 2 array:
rgbo = BYTARR(256,2)

; Intensity and opacity for muscle:
rgbo[50:70, *] = 20
SCALE3, XRANGE=[0, Vx-1], YRANGE=[0, Vy-1], ZRANGE=[0, Vz-1]

; Compute and display the projected image:
TV, VOXEL_PROJ(V, rgbo)

; Create color table array for red:
C = (FINDGEN(256)/255.) # [255., 0., 0]

; Load colors:
TVLCT, C[*], C[*], C[*], C[*], 2]

```

Example 3

This example demonstrates combining a volume with the contents of the Z-buffer:

```

; Set plotting to Z-buffer:
SET_PLOT, 'Z'

; Turn on Z buffering:
DEVICE, /Z_BUFFER

; Set scaling:
SCALE3, XRANGE=[0, Vx-1], YRANGE=[0, Vy-1], ZRANGE=[0, Vz-1]

; Draw a polygon at z equal to half the depth:
POLYFILL, [0, Vx-1, Vx-1, 0], [0, 0, Vy-1, Vy-1], Vz/2., /T3D

; Read pixel values from the Z-buffer:
zpix = TVRD()

; Read depth values from the Z-buffer:
zbuff = TVRD(/WORDS,/CHAN)

; Back to display window:
SET_PLOT, 'X'

; Compute the voxel projection and use the ZPIXELS and ZBUFFER
; keywords to combine the volume with the previously-read contents
; of the Z-buffer:
C = VOXEL_PROJ(V, rgbo, ZPIX=zpix, ZBUFF=zbuff)

```

```
; Convert from 24-bit to 8-bit image.  
IMAGE = COLOR_QUAN(C, 3, R, G, B)  
  
; Load the quantized color tables:  
TVLCT, R, G, B  
  
; Display  
TV, IMAGE
```

Version History

Introduced: Pre 4.0

See Also

[POLYSHADE](#), [PROJECT_VOL](#), [RECON3](#), [SHADE_VOLUME](#)

WAIT

The WAIT procedure suspends execution of an IDL program for a specified period. Note that because of other activity on the system, the duration of program suspension may be longer than requested.

Syntax

WAIT, *Seconds*

Arguments

Seconds

The duration of the wait, specified in seconds. This parameter can be a floating-point value to specify a fractional number of seconds.

Keywords

None.

Examples

To make an IDL program suspend execution for about five and one half seconds, use the command:

```
WAIT, 5.5
```

Version History

Introduced: Original

See Also

[EXIT](#), [STOP](#)

WARP_TRI

The WARP_TRI function returns an image array with a specified geometric correction applied. Images are warped using control (tie) points such that locations (X_i, Y_i) are shifted to (X_o, Y_o) .

The irregular grid defined by (X_o, Y_o) is triangulated using TRIANGULATE. Then the surfaces defined by (X_o, Y_o, X_i) and (X_o, Y_o, Y_i) are interpolated using TRIGRID to get the locations in the input image of each pixel in the output image. Finally, INTERPOLATE is called to obtain the result. Linear interpolation is used by default. Smooth quintic interpolation is used if the QUINTIC keyword is set.

This routine is written in the IDL language. Its source code can be found in the file `warp_tri.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = WARP_TRI( Xo, Yo, Xi, Yi, Image [, OUTPUT_SIZE=vector] [, /QUINTIC]  
[, /EXTRAPOLATE] [, /TPS] )
```

Return Value

Returns the warped image array.

Arguments

Xo, Yo

Vectors containing the locations of the control (tie) points in the output image.

Xi, Yi

Vectors containing the location of the control (tie) points in the input image. X_i and Y_i must be the same length as X_o and Y_o .

Image

The image to be warped. May be any type of data.

Keywords

OUTPUT_SIZE

Set this keyword equal to a 2-element vector containing the size of the output image. If omitted, the output image is the same size as *Image*.

QUINTIC

Set this keyword to use smooth quintic interpolation. Quintic interpolation is slower but the derivatives are continuous across triangles, giving a more pleasing result than the default linear interpolation.

EXTRAPOLATE

Set this keyword to extrapolate outside the convex hull of the tie points. Setting this keyword implies the use of QUINTIC interpolation.

TPS

Set this keyword to use Thin Plate Spline interpolation, which interpolates a set of irregularly sampled data value over a regular two dimensional grid. Thin plate splines are ideal for modeling functions with complex local distortions, such as warping functions, which are too complex to be fit with polynomials.

Version History

Introduced: Pre 4.0

See Also

[INTERPOLATE](#), [TRIANGULATE](#), [TRIGRID](#)

WATERSHED

The WATERSHED function applies the morphological watershed operator to a grayscale image. This operator segments images into watershed regions and their boundaries. Considering the gray scale image as a surface, each local minimum can be thought of as the point to which water falling on the surrounding region drains. The boundaries of the watersheds lie on the tops of the ridges. This operator labels each watershed region with a unique index, and sets the boundaries to zero.

Typically, morphological gradients, or images containing extracted edges are used for input to the watershed operator. Noise and small unimportant fluctuations in the original image can produce spurious minima in the gradients, which leads to oversegmentation. Smoothing, or manually marking the seed points are two approaches to overcoming this problem. For further reading, see Dougherty, “An Introduction to Morphological Image Processing”, SPIE Optical Engineering Press, 1992.

Syntax

```
Result = WATERSHED ( Image [, CONNECTIVITY={4 | 8} ] [, /LONG]
    [, NREGIONS=variable] )
```

Return Value

Returns an integer array of the same dimensions as the input image. An element of the *Result* array contains the integer identifier of the watershed region in which the corresponding pixel falls, or a zero if the corresponding pixel falls along the separation between regions.

Note

If the LONG keyword is set, *Result* is an array of long integers; otherwise it is an array of short integers. If the LONG keyword is *not* set, *Result* will be clipped to a maximum of 65530 regions. (Regions above 32767 will have negative values; if you have more than 32767 regions you may want to convert your *Result* to unsigned integers using UINT). Use the NREGIONS keyword to determine the actual number of regions.

Arguments

Image

The two-dimensional image to be segmented. *Image* is converted to byte type if necessary.

Keywords

CONNECTIVITY

Set this keyword to either 4 (to select 4-neighbor connectivity) or 8 (to select 8-neighbor connectivity). Connectivity indicates which pixels in the neighborhood of a given pixel are sampled during the segmentation process. 4-neighbor connectivity samples only the pixels that are immediately adjacent horizontally and vertically. 8-neighbor connectivity samples the diagonally adjacent neighbors in addition to the immediate horizontal and vertical neighbors. The default is 4-neighbor connectivity.

LONG

Set this keyword to return an array of long integers, instead of short integers. This keyword is useful if your image contains more than 32766 regions.

NREGIONS

Set this keyword to a named variable that will contain the total number of regions within *Image*. In most cases, this will be equal to the maximum value of *Result*.

Tip

If the LONG keyword is *not* set, and NREGIONS is greater than 32766, then the *Result* overflowed the maximum. If this occurs, you may wish to re-run WATERSHED with the LONG keyword set.

Examples

The following code crudely segments the grains in the data file in the IDL Demo data directory containing a magnified image of grains of pollen. Note that the IDL Demos must be installed in order to read the image used in this example.

It inverts the image, because the watershed operator finds holes, brightening the grains of pollen. Next, the morphological closing operator is applied with a disc of radius 9, contained within a 19 by 19 kernel, to eliminate holes in the image smaller

than the disc. The watershed operator is then applied to segment this image. The borders of the watershed images, which have pixel values of zero, are then merged with the original image and displayed as white.

```

;Radius of disc...
r = 9

;Create a disc of radius r
disc = SHIFT(DIST(2*r+1), r, r) LE r

;Read the image
READ_JPEG, FILEPATH('pollens.jpg', $
    SUBDIR=['examples', 'demo', 'demodata']), a

;Invert the image
b = MAX(a) - a

TVSCL, b, 0

;Remove holes of radii less than r
c = MORPH_CLOSE(b, disc, /GRAY)

TVSCL, c, 1

;Create watershed image
d = WATERSHED(c)

;Display it, showing the watershed regions
TVSCL, d, 2

;Merge original image with boundaries of watershed regions
e = a > (MAX(a) * (d EQ 0b))

TVSCL, e, 3

```

Version History

Introduced: 5.3

LONG, NREGIONS keywords added: 6.1

Wavelet Toolkit

For information, see the [Chapter 1, “Introduction to the IDL Wavelet Toolkit”](#) in the *IDL Wavelet Toolkit* manual.

WDELETE

The WDELETE procedure deletes IDL windows.

Syntax

```
WDELETE [, Window_Index [, ...]]
```

Arguments

Window_Index

A list of one or more window indices to delete. If this argument is not specified, the current window (as specified by the system variable !D.WINDOW) is deleted. If the window being deleted is not the active window, the value of !D.WINDOW remains unchanged. If the window being deleted is the active window, !D.WINDOW is set to the highest numbered window index or to -1 if no windows remain open.

If this window index is the widget ID of a draw widget, that widget is deleted.

Keywords

None.

Examples

Create IDL graphics window number 5 by entering:

```
WINDOW, 5
```

Delete window 5 by entering:

```
WDELETE, 5
```

Version History

Introduced: Original

See Also

[WINDOW](#), [WSET](#), [WSHOW](#)

WF_DRAW

The WF_DRAW procedure draws weather fronts of various types using parametric spline interpolation to smooth the lines. WF_DRAW uses the POLYFILL routine to make the annotations on the front lines.

This routine is written in the IDL language. Its source code can be found in the file `wf_draw.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WF_DRAW, X, Y [[, /COLD |, FRONT_TYPE=1] | [, /WARM |, FRONT_TYPE=2]
| [, /OCCLUDED |, FRONT_TYPE=3] | [, /STATIONARY |, FRONT_TYPE=4]
| [, /CONVERGENCE |, FRONT_TYPE=5]] [, COLOR=value] [, /DATA |,
/DEVICE |, /NORMAL] [, INTERVAL=value] [, PSYM=value]
[, SYM_HT=value] [, SYM_LEN=value] [, THICK=value]
```

Arguments

X, Y

Vectors of abscissae and ordinates defining the front to be drawn.

Keywords

COLD

Set this keyword to draw a cold front. The default is a plain line with no annotations. A cold front can also be specified by setting the keyword `FRONT_TYPE = 1`.

COLOR

Use this keyword to specify the color to use. The default = `!P.COLOR`.

CONVERGENCE

Set this keyword to draw a convergence line. A convergence line can also be specified by setting the keyword `FRONT_TYPE = 5`.

DATA

Set this keyword if X and Y are specified in data coordinates.

DEVICE

Set this keyword if X and Y are specified in device coordinates.

FRONT_TYPE

Set this keyword equal to the numeric index of type of front to draw. Front type indices are as follows: COLD=1, WARM=2, OCCLUDED=3, STATIONARY=4, CONVERGENCE = 5. Not required if plain line is desired or if an explicit front type keyword is specified.

INTERVAL

Use this keyword to specify the spline interpolation interval, in normalized units. The default = 0.01. Larger values give coarser approximations to curves, smaller values make more interpolated points.

NORMAL

Set this keyword if X and Y are specified in normalized coordinates. This is the default.

OCCLUDED

Set this keyword to draw an occluded front. An occluded front can also be specified by setting the keyword FRONT_TYPE = 3.

PSYM

Set this keyword a standard PSYM value to draw a marker on each actual (X, Y) data point. See “PSYM” on page 4340 for a list of the symbol types.

STATIONARY

Set this keyword to draw a stationary front. A stationary front can also be specified by setting the keyword FRONT_TYPE = 4.

SYM_HT

Use this keyword to specify the height of front symbols, in normalized units. The default = 0.02.

SYM_LEN

Use this keyword to specify the length and spacing factor for front symbols, in normalized units. The default = 0.15.

THICK

Use this keyword to specify the line thickness. The default = 1.0.

WARM

Set this keyword to draw a warm front. A warm front can also be specified by setting the keyword `FRONT_TYPE = 2`.

Examples

This example draws various fronts on a map of the United States. Note that this example code is in the file `wf_draw.pro`, and can be run by entering `test_wf_draw` at the IDL command line.

```

PRO test_wf_draw

MAP_SET, LIMIT = [25, -125, 50, -70], /GRID, /USA
WF_DRAW, [ -120, -110, -100], [30, 50, 45], /COLD, /DATA, THICK=2
WF_DRAW, [ -80, -80, -75], [ 50, 40, 35], /WARM, /DATA, THICK=2
WF_DRAW, [ -80, -80, -75]-10., [ 50, 40, 35], /OCCLUDED, /DATA,$
    THICK=2
WF_DRAW, [ -120, -105], [ 40,35], /STATION, /DATA, THICK=2
WF_DRAW, [ -100, -90, -90], [ 30,35,40], /CONVERG, /DATA, THICK=2

names=['None','Cold','Warm','Occluded','Stationary','Convergent']
x = [.015, .30]
y = 0.04
dy = 0.05
ty = N_ELEMENTS(names) * dy + y
POLYFILL, x[[0,1,1,0]], [0, 0, ty, ty], /NORM, COLOR=!P.BACKGROUND
FOR i=0, N_ELEMENTS(names)-1 DO BEGIN
    WF_DRAW, x, y, /NORM, FRONT_TYPE=i, THICK=2
    XYOUTS, x[1]+0.015, y[0], names[i], /NORM, CHARS=1.5
    y = y + dy
ENDFOR

END

```

Version History

Introduced: Pre 4.0

See Also

[ANNOTATE](#), [XYOUTS](#)

WHERE

The WHERE function returns a vector that contains the one-dimensional subscripts of the nonzero elements of *Array_Expression*. The length of the resulting vector is equal to the number of nonzero elements in *Array_Expression*. Frequently the result of WHERE is used as a vector subscript to select elements of an array using given criteria.

Note: When WHERE Returns -1

If all the elements of *Array_Expression* are zero, WHERE returns a scalar integer with a value of -1. Attempting to use this result as an index into another array results in a “subscripts out of bounds” error. In situations where this is possible, code similar to the following can be used to avoid errors:

```
; Use Count to get the number of nonzero elements:
index = WHERE(array, count)

; Only subscript the array if it's safe:
IF count NE 0 THEN result = array[index]
```

Syntax

```
Result = WHERE( Array_Expression [, Count] [, COMPLEMENT=variable]
               [, /L64] [, NCOMPLEMENT=variable] )
```

Return Value

Returns a longword vector containing the subscripts of non-zero array elements matching the specified conditions.

Arguments

Array_Expression

The array to be searched. Both the real and imaginary parts of a complex number must be zero for the number to be considered zero.

Count

A named variable that will receive the number of nonzero elements found in *Array_Expression*. This value is returned as a longword integer.

Note

The system variable !ERR is set to the number of nonzero elements. This effect is for compatibility with previous versions of IDL and should *not* be used in new code. Use the COUNT argument to return this value instead.

Keywords

COMPLEMENT

Set this keyword to a named variable that receives the subscripts of the zero elements of *Array_Expression*. These are the subscripts that are not returned in *Result*. Together, *Result* and COMPLEMENT specify every subscript in *Array_Expression*. If there are no zero elements in *Array_Expression*, COMPLEMENT returns a scalar integer with the value -1.

L64

By default, the result of WHERE is 32-bit integer when possible, and 64-bit integer if the number of elements being processed requires it. Set L64 to force 64-bit integers to be returned in all cases.

Note

Only 64-bit versions of IDL are capable of creating variables requiring a 64-bit result. Check the value of !VERSION.MEMORY_BITS to see if your IDL is 64-bit or not.

NCOMPLEMENT

Set this keyword to a named variable that receives the number of zero elements found in *Array_Expression*. This value is the number of subscripts that will be returned via the COMPLEMENT keyword if it is specified.

Thread Pool Keywords

This routine is written to make use of IDL's *thread pool*, which can increase execution speed on systems with multiple CPUs. The values stored in the !CPU system variable control whether IDL uses the thread pool for a given computation. In addition, you can use the thread pool keywords TPOOL_MAX_ELTS, TPOOL_MIN_ELTS, and TPOOL_NOTHREAD to override the defaults established by !CPU for a single invocation of this routine. See [Appendix C, "Thread Pool Keywords"](#) for details.

Examples

Example 1

```

; Create a 10-element integer array where each element is
; set to the value of its subscript:
array = INDGEN(10)
PRINT, 'array = ', array

; Find the subscripts of all the elements in the array that have
; a value greater than 5:
B = WHERE(array GT 5, count, COMPLEMENT=B_C, NCOMPLEMENT=count_c)

; Print how many and which elements met the search criteria:
PRINT, 'Number of elements > 5: ', count
PRINT, 'Subscripts of elements > 5: ', B
PRINT, 'Number of elements <= 5: ', count_c
PRINT, 'Subscripts of elements <= 5: ', B_C

```

IDL prints:

```

array = 0 1 2 3 4 5 6 7 8 9
Number of elements > 5: 4
Subscripts of elements > 5: 6 7 8 9
Number of elements <= 5: 6
Subscripts of elements <= 5: 0 1 2 3 4 5

```

Example 2

The WHERE function behaves differently with different kinds of array expressions. For instance, if a relational operator is used to compare an array, A, with a scalar, B, then every element of A is searched for B. However, if a relational operator is used to compare two arrays, C and D, then a comparison is made between each corresponding element (i.e. C_i & D_i , C_{i+1} & D_{i+1} , etc) of the two arrays. If the two arrays have different lengths then a comparison is only made up to the number of elements for the shorter array. The following example illustrates this behavior:

```

; Compare array, a, and scalar, b:
a = [1,2,3,4,5,5,4,3,2,1]
b = 5
PRINT, 'a = ', a
PRINT, 'b = ', b

result=WHERE(a EQ b)
PRINT,'Subscripts of a that equal b: ', result

; Now compare two arrays of different lengths:
c = [1,2,3,4,5,5,4,3,2,1]

```

```
d = [0,2,4]
PRINT, 'c = ', c
PRINT, 'd = ', d

result=WHERE(c EQ d)
PRINT, 'Subscripts of c that equal d: ', result
```

IDL prints:

```
a = 1 2 3 4 5 5 4 3 2 1
b = 5
Subscripts of a that equal b: 4 5

c = 1 2 3 4 5 5 4 3 2 1
d = 0 2 4
Subscripts of c that equal d: 1
```

Note that WHERE found only one element in the array d that equals an element in array c. This is because only the first three elements of c were searched, since d has only three elements.

Version History

Introduced: Original

See Also

[ARRAY_INDICES](#), [UNIQ](#)

WHILE...DO

The WHILE...DO statement performs its subject statement(s) as long as the expression evaluates to true. The subject is never executed if the condition is initially false.

Note

For information on using WHILE...DO and other IDL program control statements, see [Chapter 13, “Program Control”](#) in the *Building IDL Applications* manual.

Syntax

WHILE *expression* DO *statement*

or

WHILE *expression* DO BEGIN

statements

ENDWHILE

Examples

```
i = 0
WHILE (i EQ 1) DO PRINT, i
```

Because the expression (which is false in this case) is evaluated before the subject statement is executed, this code yields no output.

Version History

Introduced: Original

WIDGET_ACTIVEX

The `WIDGET_ACTIVEX` function is used to incorporate an ActiveX control into an IDL widget hierarchy. IDL provides the same object method and property manipulation facilities for ActiveX controls as it does for COM objects incorporated using the `IDLcomIDispatch` object interface, but adds the ability to process events generated by the ActiveX control using IDL's widget event handling mechanisms.

Note

IDL can only incorporate ActiveX controls on Windows 2000/XP (and later) platforms.

When you use the `WIDGET_ACTIVEX` routine, IDL automatically creates an `IDLcomActiveX` object that encapsulates the ActiveX control. You can then call the ActiveX control's methods using IDL object syntax, as discussed in [Chapter 5, "Using ActiveX Controls in IDL"](#) in the *External Development Guide* manual. You should be familiar with the material in that chapter before attempting to incorporate ActiveX controls in your IDL programs.

Note

If the COM object you want to use in your IDL application is *not* an ActiveX control, use the `IDLcomIDispatch` object class.

Warning

When displaying an ActiveX form or dialog box, it is the responsibility of the COM object to pump messages. Modal dialogs pump messages themselves, but modeless dialogs do not. IDL's COM subsystem does not provide the ability to pump messages explicitly, giving IDL no way to pump messages while a modeless dialog is displayed. As a result, calling a modeless dialog from IDL will result in an error.

Syntax

```
Result = WIDGET_ACTIVEX( Parent, COM_ID, [, /ALIGN_BOTTOM | ,
  /ALIGN_CENTER | , /ALIGN_LEFT | , /ALIGN_RIGHT | , /ALIGN_TOP]
  [, EVENT_FUNC=string] [, EVENT_PRO=string]
  [, FUNC_GET_VALUE=string] [ID_TYPE=value] [, KILL_NOTIFY=string]
  [, /NO_COPY] [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
  [, SCR_XSIZE=width] [, SCR_YSIZE=height] [, /SENSITIVE]
  [, UNAME=string] [, UNITS={0 | 1 | 2}] [, UVALUE=value]
  [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

The returned value is the widget ID of the ActiveX widget. Note that the *widget value* of the ActiveX widget is an object reference to the IDLcomActiveX object that encapsulates the specified ActiveX control.

Arguments

Parent

The widget ID of the parent widget of the new ActiveX control.

COM_ID

A string value specifying the class or program ID of the COM object to create. Note that if you specify a COM program ID via this argument, you must also set the ID_TYPE keyword.

Note

The class or program ID must follow the standard Microsoft naming convention. See [Chapter 5, “Using ActiveX Controls in IDL”](#) in the *External Development Guide* manual for more on class and program IDs, and their use with WIDGET_ACTIVEX.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a ROW base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a ROW or COLUMN base. In ROW bases, the new widget will be vertically centered. In COLUMN bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a ROW base.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

ID_TYPE

The type of COM object ID specified by the *COM_ID* parameter (class or program). If set to 0 (zero), the value is a class ID (the default). If set to 1 (one), the value is a program ID.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such callback procedure. It can be removed by setting the routine to the null string ('').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the WIDGET_CONTROL procedure to get or set the user value. All other requests that require a widget ID are

disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_BASE` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such callback procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `XSIZE` keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the

convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

The user value for a widget can be accessed and modified at any time by using the GET_UVALUE and SET_UVALUE keywords to the WIDGET_CONTROL procedure.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a "hint" to the toolkit and may be ignored in some situations.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a "hint" to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

The [GET_VALUE](#) keyword to the [WIDGET_CONTROL](#) procedure is used to retrieve an object reference to the IDLcomActiveX object that underlies the ActiveX widget. The object reference returned in this manner can be used to call the ActiveX control's native methods.

Examples

For examples using [WIDGET_ACTIVEX](#), see [Chapter 5, “Using ActiveX Controls in IDL”](#) in the *External Development Guide* manual.

Version History

Introduced: 5.5

WIDGET_BASE

The WIDGET_BASE function is used to create base widgets. Base widgets serve as containers for other widgets.

Note

In most cases, you will want let IDL determine the placement of widgets within the base widget. Do this by specifying either the COLUMN keyword or the ROW keyword. See [“Positioning Child Widgets Within a Base”](#) in the following section for details.

Exclusive And Non-Exclusive Bases

If the EXCLUSIVE or NONEXCLUSIVE keywords are specified, the base only allows button widget children.

Positioning Child Widgets Within a Base

The standard base widget does not impose any placement constraints on its child widgets. Children of a “bulletin board” base (a base that was created without setting the COLUMN or ROW keywords) have an offset of (0,0) unless an offset is explicitly specified via the XOFFSET or YOFFSET keywords. This means that if you do not specify any of COLUMN, ROW, XOFFSET, or YOFFSET keywords, child widgets will be placed one on top of the other in the upper left corner of the base.

However, laying out widgets using the XSIZE, YSIZE, XOFFSET, and YOFFSET keywords can be both tedious and error-prone. Also, if you want your widget application to display properly on different platforms, you should use the COLUMN and ROW keywords to influence child widget layouts instead of explicitly formatting your interfaces.

When the ROW or COLUMN keywords are specified, the base decides how to lay out its children, and any XOFFSET and YOFFSET keywords specified for such children are ignored.

Positioning Top-Level Bases

When locating a new top level window, some window managers ignore the program’s positioning requests and either choose a position or allow the user to choose. In such cases, the XOFFSET and YOFFSET keywords to WIDGET_BASE will not have an effect. The window manager may provide a way to disable this positioning style. The Motif window manager (mwm) can be told to honor positioning requests by placing the following lines in your `.xdefaults` file:

```
Mwm*clientAutoPlace: False
Mwm*interactivePlacement: False
```

Iconizing, Layering, and Destroying Groups of Top-Level Bases

Group membership (defined via the `GROUP_LEADER` keyword) controls the way top-level base widgets are iconized, layered, and destroyed.

Note

A group can contain sub-groups. Group behavior affects all members of a group and its sub-groups. For example, suppose we create three top-level base widgets with the following group hierarchy:

```
base1 = WIDGET_BASE()
base2 = WIDGET_BASE(GROUP_LEADER=base1)
base3 = WIDGET_BASE(GROUP_LEADER=base2)
```

Effectively, two groups are created. One group has `base2` as its leader and `base3` as its member. The other group has `base1` as its leader and both `base2` and `base3` as members. If `base1` is iconized, both `base2` and `base3` are iconized as well. If `base2` is iconized, `base3` is iconized but `base1` is not.

Widgets behave slightly differently when displayed on different platforms, and depending on whether they are floating or modal bases. The following rules apply to groups of widgets within a group leader/member hierarchy. Widgets that do not belong to the same group hierarchy cannot influence each other.

Iconization and Mapping

Bases and groups of bases can be *iconized* (or *minimized*) by clicking the system minimize control. Bases and groups of bases can also be *mapped* (made visible) and *unmapped* (made invisible).

Mapping or unmapping a group leader has no effect on the mapped state of the group members. Iconifying or mapping a group member has no effect on the group leader. Modal bases cannot be unmapped.

Bases and groups of bases can be iconized (or minimized) by clicking the system minimize control. When a group leader is iconized, all members of the group are iconized as well. Similarly, when a group leader is restored, all members of the group are restored.

Layering

Layering is the process by which groups of widgets seem to share the same plane on the display screen. Within a layer on the screen, widgets have a *Z-order*, or front-to-back order, that defines which widgets appear to be on top of other widgets.

Motif — All elements on the screen—widgets, the IDLDE, other Motif applications—share a single layer and have an arbitrary Z-order. There is no special layering of IDL widgets.

Windows — All non-floating and non-modal widgets within a group hierarchy share the same layer—that is, when one group member has the input focus, all members of the group hierarchy are displayed in a layer that appears in front of all other groups or applications. Within the layer, the widgets can have an arbitrary Z-order.

Widgets that are floating or modal always float above their group leaders.

Destruction

When a group leader widget is destroyed, either programmatically or by clicking on the system “close” button, all members of the group and all sub-groups are destroyed as well.

If a modal base is on the display, it must be dismissed before any widget can be destroyed.

Syntax

```
Result = WIDGET_BASE( [Parent] [, /ALIGN_BOTTOM | /ALIGN_CENTER | ,
  /ALIGN_LEFT | /ALIGN_RIGHT | /ALIGN_TOP
  [, /BASE_ALIGN_BOTTOM | /BASE_ALIGN_CENTER | ,
  /BASE_ALIGN_LEFT | /BASE_ALIGN_RIGHT | /BASE_ALIGN_TOP]
  [, /COLUMN | /ROW] [, /CONTEXT_EVENTS] [, /CONTEXT_MENU]
  [, EVENT_FUNC=string] [, EVENT_PRO=string] [, /EXCLUSIVE | ,
  /NONEXCLUSIVE] [, /FLOATING] [, FRAME=width]
  [, FUNC_GET_VALUE=string] [, /GRID_LAYOUT]
  [, GROUP_LEADER=widget_id{must specify for modal dialogs}]
  [, /KBRD_FOCUS_EVENTS] [, KILL_NOTIFY=string] [, /MAP{not for modal
  bases}] [, /MBAR | /MODAL] [, /NO_COPY] [, NOTIFY_REALIZE=string]
  [, PRO_SET_VALUE=string] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
  [, /SCROLL{not for modal bases}] [, /SENSITIVE] [, SPACE=value{ignored if
  exclusive or nonexclusive}] [, TAB_MODE=value] [, TITLE=string]
  [, TLB_FRAME_ATTR=value{top-level bases only}]
  [, /TLB_ICONIFY_EVENTS{top-level bases only}]
  [, /TLB_KILL_REQUEST_EVENTS{top-level bases only}]
  [, /TLB_MOVE_EVENTS{top-level bases only}] [, /TLB_SIZE_EVENTS{top-
  level bases only}] [, /TOOLBAR] [, /TRACKING_EVENTS] [, UNAME=string]
  [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, XOFFSET=value]
  [, XPAD=value{ignored if exclusive or nonexclusive}] [, XSIZE=value]
```

[, `X_SCROLL_SIZE=value`] [, `YOFFSET=value`] [, `YPAD=value`{ignored if exclusive or nonexclusive}] [, `YSIZE=value`] [, `Y_SCROLL_SIZE=value`])

X Windows Keywords: [, `DISPLAY_NAME=string`]
[, `RESOURCE_NAME=string`] [, `RNAME_MBAR=string`]

Return Value

The returned value of this function is the widget ID of the newly-created base.

Arguments

Parent

The widget ID of the parent widget. To create a *top-level* base, omit the *Parent* argument.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a ROW base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a ROW or COLUMN base. In ROW bases, the new widget will be vertically centered. In COLUMN bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a ROW base.

To apply actions triggered by menu items to widgets other than the base that includes the menubar, use the [KBRD_FOCUS_EVENTS](#) keyword to keep track of which widget has (or last had) the keyboard focus.

BASE_ALIGN_BOTTOM

Set this keyword to make all children of the new base align themselves with the bottom of the base by default. To take effect, you must also set the ROW keyword for the new base. The default can be overridden for individual child widgets by setting a different ALIGN_XXX keyword when the child widget is created.

BASE_ALIGN_CENTER

Set this keyword to make all children of the new base align themselves with the center of the base by default. To take effect, you must also set the COLUMN or ROW keyword for the new base. The default can be overridden for individual child widgets by setting a different ALIGN_XXX keyword when the child widget is created. In ROW bases, child widgets will be vertically centered. In COLUMN bases, child widgets will be horizontally centered.

BASE_ALIGN_LEFT

Set this keyword to make all children of the new base align themselves with the left side of the base by default. To take effect, you must also set the COLUMN keyword for the new base. The default can be overridden for individual child widgets by setting a different ALIGN_XXX keyword when the child widget is created.

BASE_ALIGN_RIGHT

Set this keyword to make all children of the new base align themselves with the right side of the base by default. To take effect, you must also set the COLUMN keyword for the new base. The default can be overridden for individual child widgets by setting a different ALIGN_XXX keyword when the child widget is created.

BASE_ALIGN_TOP

Set this keyword to make all children of the new base align themselves with the top of the base by default. To take effect, you must also set the ROW keyword for the new base. The default can be overridden for individual child widgets by setting a different ALIGN_XXX keyword when the child widget is created.

COLUMN

If this keyword is included, the base lays out its children in columns. The value of this keyword specifies the number of columns to be used. The number of child widgets in each column is calculated by dividing the number of child widgets created by the number of columns specified. When one column is filled, a new one is started.

Specifying both the COLUMN and ROW keywords causes an error.

Column Width

The width of each column is determined by the width of the widest widget in that column. If the GRID_LAYOUT keyword is set, all columns are as wide as the widest widget in the base.

Horizontal Size of Widgets

If any of the BASE_ALIGN_* keywords to WIDGET_BASE is set, each widget has its “natural” width, determined either by the value of the widget or by the XSIZE keyword. Similarly, if any of the child widgets specifies one of the ALIGN_* keywords, that widget will have its “natural” width. If none of the BASE_ALIGN_* or (ALIGN_*) keywords are set, all widgets in the base are as wide as their column.

Vertical Placement

Child widgets are placed vertically one below the other, with no extra space. If the GRID_LAYOUT keyword is set, each row is as high as its tallest member.

CONTEXT_EVENTS

Set this keyword to cause context menu events (or simply context events) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus); pass the context event ID to the [WIDGET_DISPLAYCONTEXTMENU](#) procedure within your widget program’s event handler to display the context menu.

For more on detecting and handling context menu events, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

CONTEXT_MENU

Set this keyword to create a base widget that can be used as the base for a context-sensitive menu (also known as a pop-up or shortcut menu). A context menu base must have as its parent one of the following types of widgets: WIDGET_BASE, WIDGET_DRAW, WIDGET_TEXT, WIDGET_LIST,

WIDGET_PROPERTY_SHEET, WIDGET_TABLE, WIDGET_TEXT or WIDGET_TREE.

For more on creating context menus, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

DISPLAY_NAME

Set this keyword equal to a string that specifies the name of the X Windows display on which the base should be displayed. This keyword has no effect on Microsoft Windows platforms.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

Note

If you are using XMANAGER to manage events for your widget application, you should not specify an event processing routine for the top-level base of the application using this keyword. Instead, specify the event processing routine using the [EVENT_HANDLER](#) keyword to XMANAGER.

EVENT_PRO

A string containing the name of a procedure to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

Note

If you are using XMANAGER to manage events for your widget application, you should not specify an event processing routine for the top-level base of the application using this keyword. Instead, specify the event processing routine using the [EVENT_HANDLER](#) keyword to XMANAGER.

EXCLUSIVE

Set this keyword to specify that the base can have only button-widget children and that only one button can be set at a time. These buttons, unlike normal button widgets, have two states—set and unset.

When one exclusive button is pressed, any other exclusive buttons (in the same base) that are currently set are automatically released. Hence, only one button can ever be set at one time.

This keyword can be used to create exclusive button menus. See the [CW_BGROU](#)P and [CW_PDMENU](#) functions for high-level menu-creation utilities.

Note

If this keyword is set, the XOFFSET and YOFFSET keywords are ignored for any widgets in this base. Exclusive bases are always laid out in columns or rows. If neither the COLUMN nor ROW keyword is specified for an exclusive base, the base defaults to COLUMN layout.

FLOATING

Set this keyword—along with the GROUP_LEADER keyword—to create a “floating” top-level base widget. If the windowing system provides Z-order control, floating base widgets appear above the base specified as their group leader. If the windowing system does not provide Z-order control, the FLOATING keyword has no effect.

The iconizing, layering, and destruction behavior of floating bases and their group leaders is discussed in [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 2255.

FRAME

The value of this keyword specifies the width of a frame (in pixels) to be drawn around the borders of the widget. Note that this keyword is only a hint to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GRID_LAYOUT

Set this keyword to force the base to have a grid layout, in which all rows have the same height, and all columns have the same width. The row heights and column widths are taken from the largest child widget.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. Widget application hierarchies are defined by group membership relationships between top-level widget bases. When a group leader is killed, for any reason, all widgets in the group are also destroyed. Iconizing and layering behavior is discussed in [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 2255.

Note

If you specify a floating base (created with the [FLOATING](#) keyword) as a group leader, all member bases must also have either the [FLOATING](#) or [MODAL](#) keywords set. If you specify a modal base (created with the [MODAL](#) keyword) as a group leader, all member bases must have the [MODAL](#) keyword set as well.

A given widget can be in more than one group. The [WIDGET_CONTROL](#) procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

KBRD_FOCUS_EVENTS

Set this keyword to make the base return keyboard focus events whenever the keyboard focus of the base changes. See the [“Events Returned by Base Widgets”](#) section below for more information.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the [WIDGET_CONTROL](#) procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the [WIDGET_EVENT](#) function is called.

Note

A procedure specified via the [CLEANUP](#) keyword to [XMANAGER](#) will override a procedure specified for your application’s top-level base with [WIDGET_BASE](#), [KILL_NOTIFY](#).

MAP

Once a widget hierarchy has been realized, it can be mapped (visible) or unmapped (invisible). This keyword specifies the initial map state for the given base and its descendants. Specifying a non-zero value indicates that the base should be mapped when realized (the default). A zero value indicates that the base should be unmapped initially.

After the base is realized, its map state can be altered using the MAP keyword to the WIDGET_CONTROL procedure.

Note

Modal bases cannot be mapped and unmapped.

Warning

Under Microsoft Windows, when a hidden base is realized, then mapped, a Windows resize message is sent by the windowing system. This “extra” resize event is generated before any manipulation of the base widget by the user.

MBAR

Set this keyword to a named variable to cause a menubar to be placed at the top of the base (the base must be a top-level base). The menubar is itself a special kind of base widget that can only have buttons as children. Upon return, the named variable contains the widget ID of the new menubar base. This widget ID can then be used to fill the menubar with pulldown menus. For example, the following widget creation commands first create a base with a menubar, then populate the menubar with a simple pulldown menu (CW_PDMENU could also have been used to construct the pulldown menu):

```
base = WIDGET_BASE(TITLE = 'Example', MBAR=bar)
file_menu = WIDGET_BUTTON(bar, VALUE='File', /MENU)
file_btn1=WIDGET_BUTTON(file_menu, VALUE='Item 1', $
    UVALUE='FILE1')
file_btn2=WIDGET_BUTTON(file_menu, VALUE='Item 2', $
    UVALUE='FILE2')
```

Note that to set X Window System resources for menubars created with this keyword, you must use the RNAME_MBAR keyword rather than the RESOURCE_NAME keyword.

If you use CW_PDMENU to create a menu for the menubar, be sure to set the MBAR keyword to that function as well.

Note also that the size returned by the [GEOMETRY](#) keyword to `WIDGET_INFO` does not include the size of the menubar.

Warning

You cannot specify both the `MBAR` and `MODAL` keywords for the same widget. Doing so will cause an error.

Warning

You cannot specify both an `APP_MBAR` and an `MBAR` for the same top-level base widget. Doing so will cause an error.

To apply actions triggered by menu items to widgets other than the base that includes the menubar, use the [KBRD_FOCUS_EVENTS](#) keyword to keep track of which widget has (or last had) the keyboard focus.

MODAL

Set this keyword to create a modal dialog. Modal dialogs can have default and cancel buttons associated with them. Default buttons are highlighted by the window system and respond to a press on the “Return” or “Enter” keys as if they had been clicked on. Cancel buttons respond to a press on the “Escape” key as if they had been clicked on. See the [DEFAULT_BUTTON](#) and [CANCEL_BUTTON](#) keywords to `WIDGET_CONTROL` for details.

Note

Modal dialogs must have a group leader. Specify the group leader for a modal top-level base via the [GROUP_LEADER](#) keyword.

Modal dialogs cannot be scrollable, nor can they support menubars. Setting the `SCROLL`, `MBAR`, or `APP_MBAR` keywords in conjunction with the `MODAL` keyword will cause an error. Modal dialogs cannot be mapped or unmapped. Setting the `MAP` keyword on a modal base will cause an error. Modal dialogs cannot be resized, regardless of the setting of the `TLB_SIZE_EVENTS` keyword.

Note

On Windows platforms, the group leader of a modal base must be realized before the modal base itself can be realized. If the group leader has not been realized, it will be realized automatically.

The iconizing, layering, and destruction behavior of modal bases and their group leaders is discussed in “[Iconizing, Layering, and Destroying Groups of Top-Level Bases](#)” on page 2255.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_BASE` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NONEXCLUSIVE

Set this keyword to specify that the base can only have button widget children. These buttons, unlike normal button widgets, have two states—set and unset. Non-exclusive bases allow any number of the toggle buttons to be set at one time.

Note

If this keyword is set, the `XOFFSET` and `YOFFSET` keywords are ignored for any widgets in this base. Non-exclusive bases are always laid out in columns or rows. If neither the `COLUMN` nor `ROW` keyword is specified for a non-exclusive base, the base defaults to `COLUMN` layout.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this

technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. Once defined, this name can be used in the user's `.xdefaults` file to customize widget resources not directly supported via the IDL widget routines. This keyword is accepted by all widget creation routines. This keyword only works with the "X" device and is ignored on Microsoft Windows platforms.

`RESOURCE_NAME` allows unrestricted access to the underlying Motif widgets within the following limitations:

- Users must have the appropriate resources defined in their `.xdefaults` or application default resource file, or IDL will not see the definitions and they will not take effect.
- Motif resources are documented in the *OSF/Motif Programmer's Reference Manual*. To use them with `RESOURCE_NAME`, the IDL programmer must determine the type of widget being used by IDL, and then look up the resources that apply to them. Hence, `RESOURCE_NAME` requires some programmer-level familiarity with Motif.
- Only resources that are not set within IDL can be modified using this mechanism. Although it is not an error to set resources also set by IDL, the IDL settings will silently override user settings. RSI does not document the resources used by IDL since the actual resources used may differ from release to release as the IDL widgets evolve. Therefore, you should set only those resources that are obviously not being set by IDL. Among the resources that are not being set by IDL are those that control colors, menu mnemonics, and accelerator keys.

Example

The sample code below produces a pulldown menu named "Menu" with 2 entries named "Item 1" and "Item 2".

Using the `RESOURCE_NAME` keyword in conjunction with X resource definitions, we can alter "Item 1" in several ways not possible through the standard IDL widgets interface. We'll give Item 1 a red background color. We'll also assign "I" as the keyboard mnemonic. Note that Motif automatically underlines the "I" in the title to indicate this. We'll also select Meta-F4 as the keyboard accelerator for selecting "Item 1". If Meta-F4 is pressed while the pointer is anywhere over this application,

the effect will be as if the menu was pulled down and “Item 1” was selected with the mouse.

```

; Simple event handler:
PRO test_event, ev
    HELP, /STRUCTURE, ev
END

; Simple widget creation routine:
PRO test

    ; The base gets the resource name "test":
    a = WIDGET_BASE(RESOURCE_NAME = 'test')
    b = WIDGET_BUTTON(a, VALUE='Menu', /MENU)

    ; Assign the Item 1 button the resource name "item1":
    c = WIDGET_BUTTON(b, VALUE='Item 1', $
        RESOURCE_NAME='item1')
    c = WIDGET_BUTTON(b, VALUE='Item 2')
    WIDGET_CONTROL, /REALIZE, a
    XMANAGER, 'test', a
END

```

Note that we gave the overall application the resource name “test”, and the “Item 1” button the resource name “item1”. Now we can use these names in the following `.xdefaults` file entries:

```

Idl*test*item1*mnemonic: I
Idl*test*item1*accelerator: Meta<Key>F4
Idl*test*item1*acceleratorText: Meta-F4
Idl*test*item1*background: red

```

Note on Specifying Color Resources

If you wish to specify unique colors for your widgets, it is generally a good idea to use a color name (“red” or “lightblue”, for example) rather than specifying an exact color match with a color string (such as “#b1b122222020”). If IDL is not able to allocate an exact color, the entire operation may fail. Specifying a named color implies “closest color match,” an operation that rarely fails.

If you need an exact color match and IDL fails to allocate the color, try modifying the `Idl.colors` resource in the `$IDL_DIR/resource/X11/lib/app-defaults/Idl` file.

RNAME_MBAR

A string containing an X Window System resource name to be applied to the menubar created by the MBAR keyword. This keyword is identical to the

RESOURCE_NAME keyword except that the resource it specifies applies only to the menubar.

ROW

If this keyword is included, the base lays out its children in rows. The value of this keyword specifies the number of rows to be used. The number of child widgets in each row is calculated by dividing the number of child widgets created by the number of rows specified. When one row is filled, a new one is started.

Specifying both the COLUMN and ROW keywords causes an error.

Row Height

The height of each row is determined by the height of the tallest widget in that row. If the GRID_LAYOUT keyword is set, all rows are as tall as the tallest widget in the base.

Vertical Size of Widgets

If any of the BASE_ALIGN_* keywords to WIDGET_BASE is set, each widget has its “natural” height, determined either by the value of the widget or by the YSIZE keyword. Similarly, if any of the child widgets specifies one of the ALIGN_* keywords, that widget will have its “natural” height. If none of the BASE_ALIGN_* or (ALIGN_*) keywords are set, all widgets in the base are as tall as their row.

Horizontal Placement

Child widgets are placed horizontally one next to the other, with no extra space. If the GRID_LAYOUT keyword is set, each column is as wide as its widest member.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SCROLL

Set this keyword to give the widget scroll bars that allow viewing portions of the widget contents that are not currently on the screen. You must also set [X_SCROLL_SIZE](#) and [Y_SCROLL_SIZE](#) to set the viewport size.

Note

Under UNIX, base widgets with scroll bars do not automatically size themselves to fit their contents. Use the [XSIZE](#) and [YSIZE](#) keywords to explicitly specify the size of the base widget.

Warning

You cannot specify both the SCROLL and MODAL keywords for the same widget. Doing so will cause an error.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

Warning

Under Microsoft Windows, a top-level base that is insensitive does not allow the user to control the window using the system close or minimize/maximize buttons.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

SPACE

The space, in units specified by the UNITS keyword (pixels are the default), between children of a row or column major base. This keyword is ignored if either the EXCLUSIVE or NONEXCLUSIVE keyword is present.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets unless it is explicitly set on an individual widget.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-113: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

After creating the widget hierarchy, you can change tabbing support using the `WIDGET_CONTROL` procedure’s `TAB_MODE` keyword, or query a widget’s support for tabbing using the `WIDGET_INFO` procedure’s `TAB_MODE` keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TITLE

A string containing the title to be used for the widget. Base widgets use the title only if they are top-level widgets or if they are children of a tab widget. For top-level base

widgets, the title appears in the “window dressing” at the top of the base widget; the appearance is platform-specific. For base widgets that have a tab widget as their parent widget, the title is used as the text of the tab.

Note that if the widget base is not wide enough to contain the specified title, the title may appear truncated. If you must be able to see the full title, you have several alternatives:

- Rearrange the widgets in the base so that the base becomes naturally wide enough. This is the best solution.
- Don’t worry about this issue. If the user needs to see the entire label, they can resize the window using the mouse.
- Create the base without using the COLUMN or ROW keywords. Instead, use the XSIZE keyword to explicitly set a usable width. This is an undesirable solution that can lead to strange-looking widget layouts.

For bases that are children of tab widgets, the title may be truncated depending on the width of the largest base that is a child of the tab widget. See the description of the MULTILINE keyword to [WIDGET_TAB](#) for additional details.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

TLB_FRAME_ATTR

Set this keyword to one of the values shown in the table below to suppress certain aspects of a top-level base’s window frame. This keyword applies only to top-level bases. The settings are merely hints to the window system and may be ignored by some window managers. Valid settings are:

Value	Meaning
1	Base cannot be resized, minimized, or maximized.
2	Suppress display of system menu.
4	Suppress title bar.
8	Base cannot be closed.

Table 3-114: Valid Values for TLB_FRAME_ATTR Keyword

Value	Meaning
16	Base cannot be moved.

Table 3-114: Valid Values for TLB_FRAME_ATTR Keyword

This keyword is set bitwise, so multiple effects can be set by adding values together. For example, to make a base that has no title bar (value 4) and cannot be moved (value 16), set the TLB_FRAME_ATTR keyword to 20 (that is, 4+16).

TLB_ICONIFY_EVENTS

Set this keyword when creating a top-level base to make that base return an event when the base is iconified or restored by the user. See the “[Events Returned by Base Widgets](#)” section below for more information.

TLB_KILL_REQUEST_EVENTS

Set this keyword, usable only with top-level bases, to send the top-level base a WIDGET_KILL_REQUEST event if a user tries to destroy the widget using the window manager (by default, widgets are simply destroyed). See the “[Events Returned by Base Widgets](#)” section below for more information.

Use this keyword to perform complex actions before allowing a widget application to exit. Note that widgets that have this keyword set are responsible for killing themselves after receiving a WIDGET_KILL_REQUEST event—they cannot be destroyed using the usual window system controls.

Use a call to TAG_NAMES with the STRUCTURE_NAME keyword set to differentiate a WIDGET_KILL_REQUEST event from other types of widget events. For example:

```
IF TAG_NAMES(event, /STRUCTURE_NAME) EQ $
    'WIDGET_KILL_REQUEST' THEN ...
```

TLB_MOVE_EVENTS

Set this keyword when creating a top-level base to make that base return an event when the base is moved on the screen by the user. See the “[Events Returned by Base Widgets](#)” section below for more information.

TLB_SIZE_EVENTS

Set this keyword, when creating a top-level base, to make that base return an event when the base is resized by the user. See the “[Events Returned by Base Widgets](#)” section below for more information.

TOOLBAR

Set this keyword to indicate that the base is used to hold bitmap buttons that make up a toolbar.

Note

Setting this keyword does not cause any changes in behavior; its only affect is to slightly alter the appearance of the bitmap buttons on the base for cosmetic reasons.

Note

On Motif platforms, if bitmap buttons are on a toolbar base that is also EXCLUSIVE or NONEXCLUSIVE, they will not have a separate “toggle” indicator, they will be grouped closely together, and will have a two-pixel shadow border.

Note

This keyword has no effect on Windows platforms.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer *enters* or *leaves* the region covered by that widget. Widget tracking events are returned as structures with the following definition:

```
{ WIDGET_TRACKING, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ID, TOP, and HANDLER are the standard fields found in every widget event. ENTER is 1 if the tracking event is an entry event, and 0 if it is an exit event.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget

hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set `UNITS` equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If `UVALUE` is not present, the widget’s initial user value is undefined.

The user value for a widget can be accessed and modified at any time by using the `GET_UVALUE` and `SET_UVALUE` keywords to the `WIDGET_CONTROL` procedure.

XOFFSET

The horizontal offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

XPAD

The horizontal space, in units specified by the `UNITS` keyword (pixels are the default), between child widgets and the edges of a row or column major base. The default value of `XPAD` is platform dependent. This keyword is ignored if either the `EXCLUSIVE` or `NONEXCLUSIVE` keyword is present.

XSIZE

The width of the widget in units specified by the `UNITS` keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the

desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

X_SCROLL_SIZE

The XSIZE keyword always specifies the width of a widget. When the SCROLL keyword is specified, this size is not necessarily the same as the width of the visible area. The X_SCROLL_SIZE keyword allows you to set the width of the scrolling viewport independently of the actual width of the widget. X_SCROLL_SIZE must be less than XSIZE for scrollbars to appear.

Use of the X_SCROLL_SIZE keyword implies SCROLL. This means that scroll bars will be added in both the horizontal and vertical directions when X_SCROLL_SIZE is specified. Because the default size of the scrolling viewport may differ between platforms, it is best to specify Y_SCROLL_SIZE when specifying X_SCROLL_SIZE.

Note

X_SCROLL_SIZE and Y_SCROLL_SIZE are ignored if you specify the screen size using the SCR_XSIZE and SCR_YSIZE keywords to the widget creation routine.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

YPAD

The vertical space, in units specified by the UNITS keyword (pixels are the default), between child widgets and the edges of a row or column major base. The default value of YPAD is platform-dependent. This keyword is ignored if either the EXCLUSIVE or NONEXCLUSIVE keyword is present.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the

desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

Y_SCROLL_SIZE

The YSIZE keyword always specifies the height of a widget. When the SCROLL keyword is specified, this size is not necessarily the same as the height of the visible area. The Y_SCROLL_SIZE keyword allows you to set the height of the scrolling viewport independently of the actual height of the widget. Y_SCROLL_SIZE must be less than YSIZE for scrollbars to appear.

Use of the Y_SCROLL_SIZE keyword implies SCROLL. This means that scroll bars will be added in both the horizontal and vertical directions when Y_SCROLL_SIZE is specified. Because the default size of the scrolling viewport may differ between platforms, it is best to specify X_SCROLL_SIZE when specifying Y_SCROLL_SIZE.

Note

X_SCROLL_SIZE and Y_SCROLL_SIZE are ignored if you specify the screen size using the SCR_XSIZE and SCR_YSIZE keywords to the widget creation routine.

Obsolete Keywords

The following keywords are obsolete:

- APP_MBAR

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) procedure affect the behavior of base widgets. In addition to those keywords that affect all widgets, the following are particularly useful: [CANCEL_BUTTON](#), [CONTEXT_EVENTS](#), [DEFAULT_BUTTON](#), [KBRD_FOCUS_EVENTS](#), [TLB_ICONIFY_EVENTS](#), [TLB_MOVE_EVENTS](#), [TLB_SIZE_EVENTS](#).

Keywords to WIDGET_INFO

A number of keywords to the [WIDGET_INFO](#) function return information that applies specifically to base widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [CONTEXT_EVENTS](#),

[KBRD_FOCUS_EVENTS](#), [MODAL](#), [TLB_ICONIFY_EVENTS](#),
[TLB_KILL_REQUEST_EVENTS](#), [TLB_MOVE_EVENTS](#), [TLB_SIZE_EVENTS](#).

Events Returned by Base Widgets

Resize Events

Top-level widget bases return the following event structure only when they are resized by the user and the base was created with the [TLB_SIZE_EVENTS](#) keyword set:

```
{ WIDGET_BASE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. The X and Y fields return the new width of the base, not including any frame provided by the window manager.

Move Events

Top-level widget bases return the following event structure when the base is moved and the base was created with the [TLB_MOVE_EVENTS](#) keyword set:

```
{ WIDGET_TLB_MOVE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. X and Y are the new location of the top left corner of the base.

Note

On Windows, move events are generated while dragging. On UNIX, move events are generated only on the mouse-up.

Note

If both [TLB_SIZE_EVENTS](#) and [TLB_MOVE_EVENTS](#) are enabled, a user resize operation that causes the top left corner of the base to move will generate both a move event and a resize event.

Iconify Events

Top-level widget bases return the following event structure when the base is iconified or restored and the base was created with the [TLB_ICONIFY_EVENTS](#) keyword set:

```
{ WIDGET_TLB_ICONIFY, ID:0L, TOP:0L, HANDLER:0L, ICONIFIED:0 }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. ICONIFIED is 1 (one) if the user iconified the base and 0 (zero) if the user restored the base.

Keyboard Focus Events

Widget bases return the following event structure when the keyboard focus changes and the base was created with the `KBRD_FOCUS_EVENTS` keyword set:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. The ENTER field returns 1 (one) if the base is gaining the keyboard focus, or 0 (zero) if the base is losing the keyboard focus.

Kill Request Events

Top-level widget bases return the following event structure only when a user tries to destroy the widget using the window manager and the base was created with the `TLB_KILL_REQUEST_EVENTS` keyword set:

```
{ WIDGET_KILL_REQUEST, ID:0L, TOP:0L, HANDLER:0L }
```

ID is the widget ID of the base generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine.

Context Menu Events

A base widget created with the `CONTEXT_EVENTS` keyword set returns the following event structure in response to a right mouse button click:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the base widget. The ROW and COL fields return meaningful information for table widgets and values of zero (0) for other widgets.

Version History

Introduced: Pre 4.0

TLB_ICONIFY_EVENTS, TLB_MOVE_EVENTS, TOOLBAR keywords; new event structures for iconify and move events: 5.6

TAB_MODE keyword added: 6.1

ROW and COL fields of WIDGET_CONTEXT event structure added: 6.1

See Also

[Chapter 28, “Widgets”](#) in the *Building IDL Applications* manual.

WIDGET_BUTTON

The WIDGET_BUTTON function creates button widgets.

Exclusive And Non-Exclusive Bases

Buttons placed into exclusive or non-exclusive bases (created via the EXCLUSIVE or NONEXCLUSIVE keywords to WIDGET_BASE function) are created as two-state “toggle” buttons, which are controlled by such bases.

Bitmap Button Labels

Widget buttons can have either a text label (specified as a string value to the VALUE keyword) or a graphic symbol in the form of a bitmap. Bitmap labels are specified in one of the following ways:

1. By setting the VALUE keyword equal to a string containing the name of an image file in BMP format, and setting the BITMAP keyword:

```
button=WIDGET_BUTTON(base, VALUE='mybitmap.bmp', /BITMAP)
```

For 16- and 256-color bitmap files, IDL uses the color of the pixel in the lower left corner as the transparent color. All pixels of this color become transparent, allowing the button color to show through.

Note

Displaying bitmapped buttons on 8-bit color X Windows displays may require using additional X colormap colors to allocate colors used by the bitmaps. If the required colormap colors are not available, the button bitmap may not display properly.

To modify the bitmap after creation, use the /BITMAP keyword with WIDGET_CONTROL:

```
WIDGET_CONTROL, button, SET_VALUE='mybitmap2.bmp', /BITMAP
```

2. By setting the VALUE keyword equal to an $n \times m$ bitmap byte array in which each bit represents a pixel with a value of either zero or one. (Arrays of this type can be produced using the CVTTOBM function.) This method creates a black-and-white bitmap label:

```
button=WIDGET_BUTTON(base, VALUE=bw_arr)
```

To modify the bitmap after creation, simply set a new value using WIDGET_CONTROL:

```
WIDGET_CONTROL, button, SET_VALUE=bw_array2
```

- By setting the VALUE keyword equal to an $n \times m \times 3$ byte array that represents a 24-bit color image, interleaved by plane, with the planes in the order red, green, blue. This method creates a color bitmap label:

```
button=WIDGET_BUTTON(base, VALUE=color_array)
```

To modify the bitmap after creation, simply set a new value using WIDGET_CONTROL:

```
WIDGET_CONTROL, button, SET_VALUE=color_array2
```

See “Using Button Widgets” in Chapter 30 of the *Building IDL Applications* manual for additional details on creating image files and arrays for use as button bitmaps.

Syntax

```
Result = WIDGET_BUTTON( Parent [, ACCELERATOR=string]
  [, /ALIGN_CENTER | , /ALIGN_LEFT | , /ALIGN_RIGHT] [, /BITMAP]
  [, /CHECKED_MENU] [, /DYNAMIC_RESIZE] [, EVENT_FUNC=string]
  [, EVENT_PRO=string] [, FONT=string] [, FRAME=width]
  [, FUNC_GET_VALUE=string] [, GROUP_LEADER=widget_id] [, /HELP]
  [, KILL_NOTIFY=string] [, /MENU] [, /NO_COPY] [, /NO_RELEASE]
  [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
  [, /PUSHBUTTON_EVENTS] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
  [, /SENSITIVE] [, /SEPARATOR] [, TAB_MODE=value] [, TOOLTIP=string]
  [, /TRACKING_EVENTS] [, UNAME=string] [, UNITS={0 | 1 | 2}]
  [, UVALUE=value] [, VALUE=value] [, X_BITMAP_EXTRA=bits]
  [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value] )
```

X Windows Keywords: [, RESOURCE_NAME=string]

Return Value

The returned value of this function is the widget ID of the newly-created button.

Arguments

Parent

The widget ID of the parent for the new button widget. The parent of a button widget must be one of the following:

- A base widget.

2. A widget created using the **MBAR** keyword on a top-level base.
3. A button widget which has the **MENU** keyword set.

Keywords

ACCELERATOR

Set this keyword to assign a key combination that will activate a menu item or button event using the keyboard instead of the mouse. The value of the keyword is a case-insensitive string that specifies zero or more modifier keys (**Ctrl**, **Shift**, or **Alt**) and one other key. (The developer must take special steps to enable **Alt** key accelerators for Macintosh users). See “[Enabling Alt Key Accelerators on Macintosh](#)” in Chapter 30 of the *Building IDL Applications* manual for details.) If there is more than one item in the string, a “+” must separate them. For example:

```
bRun = WIDGET_BUTTON( base, VALUE = "Run", ACCELERATOR = "F5" )
bPause = WIDGET_BUTTON( base, VALUE = "Pause", $
    ACCELERATOR = "Ctrl+F5" )
bResume = WIDGET_BUTTON( base, VALUE = "Resume", $
    ACCELERATOR = "Ctrl+Shift+F5" )
```

Accelerators can be defined for menu items and various types of **WIDGET_BUTTON** on Windows. However, on UNIX, only menu item accelerators are supported. Context menu items do not support accelerators on any platform.

The valid combinations are:

Accelerator Keys	Description
Ctrl , Shift , or Alt plus an alphanumeric key	A modifier key plus any alphabetic character, A-Z (which is case-insensitive), or a number, 0-9, creates a valid accelerator.
Ctrl , Shift , or Alt plus a number pad key	A modifier key plus any key on the number pad can be used as an accelerator. However, the NumLock key must be activated for any accelerator using number pad keys to function properly. Note - On Windows only, a keyboard accelerator using the Shift key and a number key on the number pad will not work.

Table 3-115: Valid **ACCELERATOR** Keyword Combinations

Accelerator Keys	Description
Ctrl, Shift, or Alt plus the BackSpace, Tab or Space key	These miscellaneous keys need a modifier key in the accelerator definition.
Navigation keys (Home, End, PageUp, PageDown, Up, Down, Left, Right)	The navigation keys do not require a modifier in the accelerator definition. Prior and PageUp are equivalent as are Next and PageDown . Up, Down, Left, and Right map to the arrow keys.
Function keys (F1 to F12)	Function keys do not need a modifier key in the accelerator definition. However, not all platforms support the use of all function keys as accelerators. See “Successfully Implementing Keyboard Accelerators” in Chapter 30 of the <i>Building IDL Applications</i> manual for details.
Return, Escape, Insert, Del keys	These miscellaneous keys do not need a modifier key in the accelerator definition. You must specify Return in the accelerator definition to indicate the Enter key on Windows. You must specify Del in the accelerator definition to indicate the Delete key.

Table 3-115: Valid ACCELERATOR Keyword Combinations (Continued)

When an accelerator is defined for a menu item, the ACCELERATOR keyword string is automatically displayed next to the menu item value. The ACCELERATOR keyword string is not included with a button value. Therefore, the VALUE keyword of a WIDGET_BUTTON that is not a menu item should also indicate the accelerator keyboard shortcut so that the user is aware of the option.

See [“Assigning Accelerators in Widget Applications”](#) in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

Note

Ordinarily, accelerators are processed before keyboard events reach widgets that have keyboard focus. Setting IGNORE_ACCELERATORS allows draw widgets and widgets with an editable text area (combobox, property sheet, table and text widgets) to receive keyboard events instead of the accelerator key combinations being captured by the accelerator. See [“Disabling Button Widget Accelerators”](#) in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

ALIGN_CENTER

Set this keyword to center justify the button's text label.

ALIGN_LEFT

Set this keyword to left justify the button's text label.

ALIGN_RIGHT

Set this keyword to right justify the button's text label.

BITMAP

Set this keyword to specify that the VALUE keyword specifies the name of a .bmp file.

The value of a widget button can be a bitmap as described in “[Bitmap Button Labels](#)” on page 2280. If you specify the name of a bitmap file with the VALUE keyword, you must also set the BITMAP keyword.

Note

The use of bitmap values for buttons which have a button widget as their parent widget may cause unexpected behavior and is not supported.

CHECKED_MENU

Set this keyword on a menu entry button to enable the ability to place a check (Windows) or selection box (Motif) next to the menu entry. The parent widget of the button must be either a button widget created with the MENU keyword or a base widget created with the CONTEXT_MENU keyword.

Note

Setting this keyword does not initially “select” the menu item; selection marks are displayed and removed via the [SET_BUTTON](#) keyword to WIDGET_CONTROL.

DYNAMIC_RESIZE

Set this keyword to create a widget that resizes itself to fit its new value whenever its value is changed. Note that this keyword does not take effect when used with the SCR_XSIZE, SCR_YSIZE, XSIZE, or YSIZE keywords. If one of these keywords is also set, the widget will be sized as specified by the sizing keyword and will never resize itself dynamically.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See [“About Device Fonts”](#) on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if `FONT` is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FRAME

The value of this keyword specifies the width of a frame in units specified by the `UNITS` keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

HELP

Set this keyword to tell the widget toolkit that this button is a “help” button for a menubar and should be given that appearance. For example, Motif specifies that the help menubar item is displayed on the far right of the menubar. This keyword is ignored in all other contexts and may be ignored by window managers that have no such special appearance defined.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

MENU

The presence of this keyword indicates that the button will be used to activate a pull-down menu. Such buttons can have button children that are then placed into a pull-down menu.

Under Motif, if the value specified for `MENU` is greater than 1, the button label is enclosed in a box to indicate that this button is a pull-down menu. See the [CW_PDMENU](#) function for a high-level pull-down menu creation utility.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would

otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the UVALUE keyword to WIDGET_BUTTON or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. On a “get” operation (GET_UVALUE keyword to WIDGET_CONTROL), the user value of the widget in question becomes undefined.

NO_RELEASE

Set this keyword to make exclusive and non-exclusive buttons generate only *select* events. This keyword has no effect on regular buttons.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the SET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

PUSHBUTTON_EVENTS

Set this keyword to cause separate button events to be issued for the widget when the left mouse button is pressed and released, or when the spacebar is pressed and released.

Note

This keyword has no effect on exclusive or non-exclusive buttons.

When this keyword is *not* set and the button is selected, pressing and releasing either the left mouse button or the spacebar generates a single button event, with the SELECT field set equal to 1. When this keyword *is* set:

- Pressing the left mouse button generates a button event with the SELECT field set equal to 1.
- Releasing the left mouse button generates a button event with the SELECT field set equal to 0.

- Pressing the spacebar generates a button event with the `SELECT` field set equal to 1 immediately followed by a second button event with the `SELECT` field set equal to 0.
- Pressing and holding the spacebar generates a series of button events, with the value of the `SELECT` field alternating between 1 and 0. The rate at which events are generated is governed by the key-repeat settings of the operating system.

For the structure of button events, see “[Events Returned by Button Widgets](#)” on page 2294.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `XSIZE` keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `YSIZE` keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If `SENSITIVE` is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the `SENSITIVE` keyword with the [WIDGET_CONTROL](#) procedure.

SEPARATOR

Set this keyword to tell the widget toolkit that this button is part of a pulldown menu pane and that a separator line should be added directly above this entry. This keyword is ignored in all other contexts.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets from the parent `WIDGET_BASE` unless it is explicitly set on an individual widget. If the `TAB_MODE` value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Note

For radio buttons, the `TAB_MODE` value is automatically inherited from the parent base and cannot be individually specified. Attempting to set `TAB_MODE` on a menu item or menu base will generate an error.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-116: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

Once a `WIDGET_BUTTON` widget receives focus through tabbing, additional navigation capabilities depend upon the platform as follows:

Widget	Description
<code>WIDGET_BUTTON</code> (Grouped, exclusive button widgets)	<ul style="list-style-type: none"> • On Windows – radio buttons can receive and lose focus through tabbing as long as there is a selected button within the group. Use the arrow keys to change the selection within the group. • On UNIX — the Motif library controls tabbing functionality.
<code>WIDGET_BUTTON</code> (Grouped, non-exclusive button widgets)	<ul style="list-style-type: none"> • On Windows – toggle buttons (or check boxes) can individually receive and lose focus through tabbing. Use the Space key to select or deselect a check box. • On UNIX —the Motif library controls tabbing functionality.

Table 3-117: Tabbing Behavior in Windows and UNIX Widget Applications

After creating the widget hierarchy, you can change tabbing support using the `WIDGET_CONTROL` procedure's `TAB_MODE` keyword, or query a widget's support for tabbing using the `WIDGET_INFO` procedure's `TAB_MODE` keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TOOLTIP

Set this keyword to a string that will be displayed when the cursor hovers over the widget. For UNIX platforms, this string must be non-zero in length.

Note

Tooltips cannot be created for menu sub-items. The topmost button on a menu can, however, have a tooltip.

Note

If your application uses hardware rendering and a RETAIN setting of either zero or one, tooltips will cause draw widgets to generate expose events if the tooltip obscures the drawable area. This is true even if the tooltip is associated with another widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for WIDGET_BASE.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a widget button is the label for that button. You can set VALUE to any of the following:

- a string value, which displays as text;
- an $n \times m$ byte array, which displays as a black-and-white bitmap image;
- an $n \times m \times 3$ byte array, which displays as a 24-bit color bitmap image;
- if the BITMAP keyword is also specified, the name of a bitmap image file that displays as a color bitmap image of the same depth as the image in the file.

See “[Bitmap Button Labels](#)” on page 2280 for additional details on using bitmap images as button labels.

Note

Under Microsoft Windows, including the ampersand character (&) in the value of a button widget causes the window manager to place an underline under the character following the ampersand. (This is a feature of Microsoft Windows, and is generally used to indicate which character is used as a keyboard accelerator for the button.) If you are designing an application that will run on different platforms, you should avoid the use of the ampersand in button value strings.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

X_BITMAP_EXTRA

When creating a bitmap button that is not of a “byte-aligned” size (i.e., a dimension is not a multiple of 8), this keyword specifies how many bits of the supplied bitmap must be ignored (within the end byte). For example, to create a 10 by 8 bitmap, you need to supply a 2 by 8 array of bytes and ignore the bottom 6 bits. Therefore, you would specify `X_BITMAP_EXTRA = 6`.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) procedure affect the behavior of button widgets. In addition to those keywords that affect all widgets, the following are particularly useful: [DYNAMIC_RESIZE](#), [DYNAMIC_RESIZE](#), [GET_VALUE](#), [INPUT_FOCUS](#), [PUSHBUTTON_EVENTS](#), [SET_BUTTON](#), [SET_VALUE](#), [TOOLTIP](#), [X_BITMAP_EXTRA](#).

Keywords to WIDGET_INFO

Some keywords to the [WIDGET_INFO](#) function return information that applies specifically to button widgets. In addition to those keywords that apply to all widgets,

the following are particularly useful: [BUTTON_SET](#), [DYNAMIC_RESIZE](#), [PUSHBUTTON_EVENTS](#), [TOOLTIP](#).

Events Returned by Button Widgets

Pressing the mouse button while the mouse cursor is over a button widget causes the widget to generate an event. The event structure returned by the `WIDGET_EVENT` function is defined by the following statement:

```
{WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0}
```

`ID` is the widget id of the button generating the event. `TOP` is the widget ID of the top level widget containing `ID`. `HANDLER` contains the widget ID of the widget associated with the handler routine. `SELECT` is set to 1 if the button was set, and 0 if released. Unless the [PUSHBUTTON_EVENTS](#) keyword is set, normal buttons do not generate events when released, so `SELECT` will always be 1. Toggle buttons (children of either an exclusive or non-exclusive base) always return separate events for the set and release actions.

Version History

Introduced: Pre 4.0

`CHECKED_MENU` and `TOOLTIP` keywords added: 5.6

`PUSHBUTTON_EVENTS` keyword added: 6.0

`ACCELERATOR`, `TAB_MODE` keywords added: 6.1

See Also

[CW_BGROU](#)P, [CW_PDMENU](#)

WIDGET_COMBOBOX

The WIDGET_COMBOBOX function creates combobox widgets, which are similar to droplist widgets. The main difference between the combobox widget and the droplist widget is that the combobox widget can be created in such a way that the text field is editable, allowing the user to enter a value that is not on the list.

A combobox widget displays a text field and an arrow button. If the combobox is not editable, selecting either the text field or the button reveals a list of options from which to choose. When the user selects a new option from the list, the list disappears and the text field displays the currently selected option. This action generates an event containing the index of the selected item, which ranges from zero to the number of elements in the list minus one.

If the combobox is editable, text can be entered in the text box without causing the list to drop down. This action causes an event in which the index field is set to -1, allowing you to distinguish this event from list selections.

The text of the current selection is returned in the STR field of the WIDGET_COMBOBOX event structure. See [“Widget Events Returned by Combobox Widgets”](#) on page 2304 for details.

Note

WIDGET_COMBOBOX is not currently available on Compaq True64 UNIX platforms due to that platform’s lack of support for the necessary Motif libraries.

Syntax

```
Result = WIDGET_COMBOBOX( Parent [, /DYNAMIC_RESIZE] [, /EDITABLE]
    [, EVENT_FUNC=string] [, EVENT_PRO=string] [, FONT=string]
    [, FRAME=value] [, FUNC_GET_VALUE=string]
    [, GROUP_LEADER=widget_id] [, IGNORE_ACCELERATORS=value]
    [, KILL_NOTIFY=string] [, /NO_COPY] [, NOTIFY_REALIZE=string]
    [, PRO_SET_VALUE=string] [, RESOURCE_NAME=string]
    [, SCR_XSIZE=width] [, SCR_YSIZE=height] [, /SENSITIVE]
    [, TAB_MODE=value] [, /TRACKING_EVENTS] [, UNAME=string]
    [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=value] [, XOFFSET=value]
    [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created combobox widget.

Arguments

Parent

The widget ID of the parent widget for the new combobox widget.

Keywords

DYNAMIC_RESIZE

Set this keyword to create a widget that resizes itself to fit its new value whenever its value is changed.

Note

This keyword does not take effect when used with the `SCR_XSIZE`, `SCR_YSIZE`, `XSIZE`, or `YSIZE` keywords. If one of these keywords is also set, the widget will be sized as specified by the sizing keyword and will never resize itself dynamically.

EDITABLE

Set this keyword to create an editable combobox. If the combobox is editable, users can enter or modify items in the text field. Changes in the combobox text field will cause combobox events with the `INDEX` field of the event structure set to -1. The current text will be ' in the `STR` field of the event structure.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a device font (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 in the *IDL Reference Guide* for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if FONT is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FRAME

The value of this keyword specifies the width of a frame in units specified by the UNITS keyword (pixels are the default) to be drawn around the borders of the widget.

Note

This keyword is only a hint to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as group leader for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. You cannot remove a widget from an existing group.

IGNORE_ACCELERATORS

Set this keyword to specify what accelerators are to be ignored when this combobox widget has keyboard focus. Valid values are:

- A string or string array containing any value that is legal for WIDGET_BUTTON's **ACCELERATOR** keyword
- The number 1, indicating all accelerators should be ignored

Ordinarily, accelerators are processed before keyboard events reach the widget that has the keyboard focus. Consider a case where the accelerator “Ctrl+V” has been mapped to a button that allows the user to paste objects copied from one drawing area into another. In an application that also has a combobox, attempting to use **Ctrl+V** to paste text into the combobox would fail as this event would be stolen by the accelerator. Setting **IGNORE_ACCELERATORS** to “Ctrl+V” allows a combobox widget with an editable text area to receive keyboard events instead of the button with a conflicting accelerator. Once the combobox loses focus, all specified accelerators are automatically re-enabled.

See “[Disabling Button Widget Accelerators](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such callback procedure. This callback procedure can be removed by setting the routine name to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the **WIDGET_CONTROL** procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the **WIDGET_EVENT** function is called.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the **SET_UVALUE** and **GET_UVALUE** keywords to **WIDGET_CONTROL**, IDL makes a second copy of the data being transferred. Although this technique works well for small data, it can have a significant memory cost when the data being copied is large.

If the **NO_COPY** keyword is set, IDL handles these operations differently. Rather than copying the source data, it takes the data away from the source and attaches it

directly to the destination. This feature can be used by compound widgets to obtain state information from a UVALUE without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. Upon a set operation (using the UVALUE keyword to WIDGET_COMBOBOX or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. Upon a get operation (GET_UVALUE keyword to WIDGET_CONTROL), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string containing the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single callback procedure. This callback procedure can be removed by setting the routine name to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the SET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 in the *IDL Reference Guide* for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If `SENSITIVE` is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget.

Note

Some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the `SENSITIVE` keyword with the [WIDGET_CONTROL](#).

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the `TAB_MODE` value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the <code>TAB_MODE</code> Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-118: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

Once the `WIDGET_COMBOBOX` has focus, use the up and down arrow keys to navigate among the individual items.

After creating the widget hierarchy, you can change tabbing support using the [`WIDGET_CONTROL`](#) procedure’s `TAB_MODE` keyword, or query a widget’s support for tabbing using the [`WIDGET_INFO`](#) procedure’s `TAB_MODE` keyword.

See [“Tabbing in Widget Applications”](#) in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see [“`TRACKING_EVENTS`”](#) on page 2273 in the documentation for `WIDGET_BASE`.

UNAME

Set this keyword to a string, which is used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The user value to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a combobox widget is a scalar string or array of strings that contains the text of the list items (one list item per array element). Combobox widgets are sized based on the length (in characters) of the longest item specified in the array of values for the VALUE keyword.

Note

Null strings are not allowed in the combobox widget item list.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, "Using Language Catalogs"](#) in the *Building IDL Applications* manual.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. You should avoid using this style of widget programming.

XSIZE

The desired width of the combobox widget area, in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword does not control the size of the combobox button or of the dropped list. Instead, it controls the size around the combobox button and, as such, is not particularly useful.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. You should avoid using this style of widget programming.

YSIZE

The desired height of the widget, in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword does not control the size of the combobox button or of the dropped list. Instead, it controls the size around the combobox button and, as such, is not particularly useful.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) affect the behavior of combobox widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: [COMBOBOX_ADDITEM](#), [COMBOBOX_DELETEITEM](#),

[COMBOBOX_INDEX](#), [DYNAMIC_RESIZE](#), [GET_VALUE](#),
[SET_COMBOBOX_SELECT](#), [SET_VALUE](#).

Keywords to WIDGET_INFO

A number of keywords to the [WIDGET_INFO](#) return information that applies specifically to combobox widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: [COMBOBOX_GETTEXT](#), [COMBOBOX_NUMBER](#), [DYNAMIC_RESIZE](#).

Widget Events Returned by Combobox Widgets

Pressing the mouse button while the mouse pointer is over an element of a combobox widget causes the widget to change the text field on the combobox and to generate an event. The event structure returned by the [WIDGET_EVENT](#) function is defined by the following statement:

```
{WIDGET_COMBOBOX, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L, STR:""}
```

The first three fields are the standard fields found in every widget event. [INDEX](#) returns the index of the selected item. This can be used to index the array of names originally used to set the widget's value. If the event was caused by text changes in an editable combobox, the [INDEX](#) field will be set to -1. If you are using an editable combobox, it is important to check for the value of -1 prior to using the value of the [INDEX](#) field as an index into the array of items. The text of the current selection is returned in the [STR](#) field, which may eliminate the need to use the index field in many cases.

Note

Platform-specific UI toolkits behave differently if a combobox widget has only a single element. On some platforms, selecting that element again does not generate an event. Events are always generated if the list contains multiple items.

Version History

Introduced: 5.6

[IGNORE_ACCELERATORS](#), [TAB_MODE](#) keywords added: 6.1

See Also

[CW_PDMENU](#), [WIDGET_BUTTON](#), [WIDGET_DROPLIST](#), [WIDGET_LIST](#)

WIDGET_CONTROL

The WIDGET_CONTROL procedure is used to realize, manage, and destroy widget hierarchies. It is often used to change the default behavior or appearance of previously-realized widgets.

Syntax

WIDGET_CONTROL [, *Widget_ID*]

Keywords that apply to all widgets: [, [BAD_ID=variable](#)] [, [/CLEAR_EVENTS](#)] [, [DEFAULT_FONT=string](#){ do not specify *Widget_ID*}] [, [/DELAY_DESTROY](#){ do not specify *Widget_ID*}] [, [/DESTROY](#)] [, [EVENT_FUNC=string](#)] [, [EVENT_PRO=string](#)] [, [FUNC_GET_VALUE=string](#)] [, [GET_UVALUE=variable](#)] [, [GROUP_LEADER=widget_id](#)] [, [/HOURGLASS](#){ do not specify *Widget_ID*}] [, [KILL_NOTIFY=string](#)] [, [/MAP](#)] [, [/NO_COPY](#)] [, [NOTIFY_REALIZE=string](#)] [, [PRO_SET_VALUE=string](#)] [, [/REALIZE](#)] [, [/RESET](#){ do not specify *Widget_ID*}] [, [SCR_XSIZE=width](#)] [, [SCR_YSIZE=height](#)] [, [SEND_EVENT=structure](#)] [, [/SENSITIVE](#)] [, [SET_UNAME=string](#)] [, [SET_UVALUE=value](#)] [, [/SHOW](#)] [, [TIMER=value](#)] [, [TLB_GET_OFFSET=variable](#)] [, [TLB_GET_SIZE=variable](#)] [, [/TLB_KILL_REQUEST_EVENTS](#)] [, [TLB_SET_TITLE=string](#)] [, [TLB_SET_XOFFSET=value](#)] [, [TLB_SET_YOFFSET=value](#)] [, [/TRACKING_EVENTS](#)] [, [UNITS={0 | 1 | 2}](#)] [, [/UPDATE](#)] [, [XOFFSET=value](#)] [, [XSIZE=value](#)] [, [YOFFSET=value](#)] [, [YSIZE=value](#)]

Keywords that apply to widgets created with WIDGET_ACTIVEX:
[, [GET_VALUE=value](#)]

Keywords that apply to widgets created with WIDGET_BASE:
[, [BASE_SET_TITLE=string](#)] [, [CANCEL_BUTTON=widget_id](#){ for modal bases}] [, [/CONTEXT_EVENTS](#)] [, [DEFAULT_BUTTON=widget_id](#){ for modal bases}] [, [/ICONIFY](#)] [, [/KBRD_FOCUS_EVENTS](#)] [, [TAB_MODE=value](#)] [, [/TLB_ICONIFY_EVENTS](#)] [, [/TLB_KILL_REQUEST_EVENTS](#)] [, [/TLB_MOVE_EVENTS](#)] [, [/TLB_SIZE_EVENTS](#)]

Keywords that apply to widgets created with WIDGET_BUTTON: [, [/BITMAP](#)] [, [/DYNAMIC_RESIZE](#)] [, [GET_VALUE=value](#)] [, [/INPUT_FOCUS](#)] [, [/PUSHBUTTON_EVENTS](#)] [, [/SET_BUTTON](#)] [, [SET_VALUE=value](#)] [, [TAB_MODE=value](#)] [, [TOOLTIP=string](#)] [, [X_BITMAP_EXTRA=bits](#)]

Keywords that apply to widgets created with WIDGET_COMBOBOX:

[, COMBOBOX_ADDITEM=*string*] [, COMBOBOX_DELETEITEM=*integer*]
 [, COMBOBOX_INDEX=*integer*] [, /DYNAMIC_RESIZE]
 [, GET_VALUE=*value*] [, SET_COMBOBOX_SELECT=*integer*]
 [, SET_VALUE=*value*] [, TAB_MODE=*value*]

Keywords that apply to widgets created with WIDGET_DRAW:

[, /DRAW_BUTTON_EVENTS] [, /DRAW_EXPOSE_EVENTS]
 [, DRAW_KEYBOARD_EVENTS={0 | 1 | 2}] [, /DRAW_MOTION_EVENTS]
 [, /DRAW_VIEWPORT_EVENTS] [, DRAW_XSIZE=*integer*]
 [, DRAW_YSIZE=*integer*] [, GET_DRAW_VIEW=*variable*]
 [, GET_UVALUE=*variable*] [, GET_VALUE=*variable*] [, /INPUT_FOCUS]
 [, SET_DRAW_VIEW=[*x, y*] [, TOOLTIP=*string*]

Keywords that apply to widgets created with WIDGET_DROPLIST:

[, /DYNAMIC_RESIZE] [, GET_VALUE=*variable*]
 [, SET_DROPLIST_SELECT=*integer*] [, SET_VALUE=*value*]
 [, TAB_MODE=*value*]

Keywords that apply to widgets created with WIDGET_LABEL:

[, /DYNAMIC_RESIZE] [, GET_VALUE=*value*] [, SET_VALUE=*value*]

Keywords that apply to widgets created with WIDGET_LIST:

[, /CONTEXT_EVENTS] [, SET_LIST_SELECT=*value*]
 [, SET_LIST_TOP=*integer*] [, SET_VALUE=*value*] [, TAB_MODE=*value*]

Keywords that apply to widgets created with WIDGET_PROPERTY SHEET:

[, /CONTEXT_EVENTS] [, /EDITABLE] [, /MULTIPLE_PROPERTIES]
 [, PROPERTY SHEET_SETSELECTED=*empty string, string, or array of strings*]
 [, REFRESH_PROPERTY=*string, array of strings, or integer*]

Keywords that apply to widgets created with WIDGET_SLIDER:

[, GET_VALUE=*value*] [, SET_SLIDER_MAX=*value*]
 [, SET_SLIDER_MIN=*value*] [, SET_VALUE=*value*] [, TAB_MODE=*value*]

Keywords that apply to widgets created with WIDGET_TAB:

[, SET_TAB_CURRENT=*index*] [, SET_TAB_MULTILINE=*value*]
 [, TAB_MODE=*value*]

Keywords that apply to widgets created with WIDGET_TABLE:

[, ALIGNMENT={0 | 1 | 2}] [, /ALL_TABLE_EVENTS] [, AM_PM=[*string, string*]] [, COLUMN_LABELS=*string_array*] [, COLUMN_WIDTHS=*array*]
 [, /CONTEXT_EVENTS] [, DAYS_OF_WEEK=*string_array*{7 names}]
 [, /DELETE_COLUMNS {not for row_major mode}] [, /DELETE_ROWS {not for column_major mode}] [, /EDITABLE] [, EDIT_CELL=[*integer, integer*]]
 [, FORMAT=*value*] [, GET_VALUE=*variable*] [, INSERT_COLUMNS=*value*]

```
[, INSERT_ROWS=value] [, /KBRD_FOCUS_EVENTS]
[, MONTHS=string_array{12 names}] [, ROW_LABELS=string_array]
[, ROW_HEIGHTS=array] [, SET_TABLE_SELECT=[left, top, right, bottom]]
[, SET_TABLE_VIEW=[integer, integer]] [, SET_TEXT_SELECT=[integer,
integer]] [, SET_VALUE=value] [, TAB_MODE=value]
[, /TABLE_BLANK=cells] [, /TABLE_DISJOINT_SELECTION]
[, TABLE_XSIZE=columns] [, TABLE_YSIZE=rows]
[, /USE_TABLE_SELECT | , USE_TABLE_SELECT=[left, top, right, bottom]]
[, /USE_TEXT_SELECT]
```

Keywords that apply to widgets created with **WIDGET_TEXT:**

```
[, /ALL_TEXT_EVENTS] [, /APPEND] [, /CONTEXT_EVENTS]
[, /EDITABLE] [, GET_VALUE=variable] [, /INPUT_FOCUS]
[, /KBRD_FOCUS_EVENTS] [, /NO_NEWLINE]
[, SET_TEXT_SELECT=[integer, integer]]
[, SET_TEXT_TOP_LINE=line_number] [, SET_VALUE=value]
[, TAB_MODE=value] [, /USE_TEXT_SELECT]
```

Keywords that apply to widgets created with **WIDGET_TREE:**

```
[, SET_TREE_BITMAP=array] [, /SET_TREE_EXPANDED]
[, SET_TREE_SELECT={0 | 1 | widget ID | array of widget IDs}]
[, /SET_TREE_VISIBLE] [, TAB_MODE=value]
```

Arguments

Widget_ID

The widget ID of the widget to be manipulated. This argument is required by all operations, unless the description of the specific keyword states otherwise. Note that if *Widget_ID* is not provided for a keyword that needs it, that keyword is quietly ignored.

Keywords

Not all keywords to **WIDGET_CONTROL** apply to all combinations of widgets. In the following list, descriptions of keywords that affect only certain types of widgets include a list of the widgets for which the keyword is useful.

ALIGNMENT

This keyword applies to widgets created with the **WIDGET_TABLE** function.

Set this keyword equal to a scalar, 2-D array, or 1-D array specifying the alignment of the contents of each cell. An alignment of 0 (the default) aligns the left edge of the

text with the left edge of the cell. An alignment of 2 right-justifies the text, while 1 results in text centered within the cell.

If the `USE_TABLE_SELECT` keyword is not set:

- If `ALIGNMENT` is set equal to a scalar, all table cells are aligned as specified. If `ALIGNMENT` is set equal to a 2-D array, the alignment of each table cell is governed by the corresponding element of the array.

If the `USE_TABLE_SELECT` keyword is set:

- In standard selection mode, if `ALIGNMENT` is set equal to a scalar, all cells in the selection are aligned as specified. If `ALIGNMENT` is set equal to a 2-D array, the alignment of each selected cell is governed by the corresponding element of the array.
- In disjoint selection mode, if `ALIGNMENT` is set equal to a scalar, all cells in the selection are aligned as specified. If `ALIGNMENT` is set equal to a 1-D array, the alignment of each selected cell is governed by the corresponding element of the array.

ALL_TABLE_EVENTS

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword to cause the table widget to generate events whenever the user changes the contents of a table cell.

Note

If the `EDITABLE` keyword is set, an insert character event (`TYPE=0`) is generated when the user presses the `RETURN` or `ENTER` key in the text widget, even if the `ALL_EVENTS` keyword is not set. End-of-line events such as these could be used by the programmer as an indication to check the cell value or to set the currently selected cell to the next cell. See the table below for details on the interaction between `ALL_TABLE_EVENTS` and `EDITABLE`.

Keywords		Effects	
ALL_TABLE_EVENTS	EDITABLE	Input changes widget contents?	Type of events generated
Not set	Not set	No	None
Not set	Set	Yes	End-of-line insertion
Set	Not set	No	All events
Set	Set	Yes	All events

Table 3-119: Effects of using the ALL_TABLE_EVENTS and EDITABLE keywords

ALL_TEXT_EVENTS

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to cause the text widget to generate events whenever the user changes the contents of the text area.

Note

If the EDITABLE keyword is set, an insert character event (TYPE=0) is generated when the user presses the RETURN or ENTER key in the text widget, even if the ALL_EVENTS keyword is not set. See the table below for details on the interaction between ALL_TEXT_EVENTS and EDITABLE.

Keywords		Effects	
ALL_TEXT_EVENTS	EDITABLE	Input changes widget contents?	Type of events generated
Not set	Not set	No	None
Not set	Set	Yes	End-of-line insertion

Table 3-120: Effects of using the ALL_TEXT_EVENTS and EDITABLE keywords

Keywords		Effects	
ALL_TEXT_EVENTS	EDITABLE	Input changes widget contents?	Type of events generated
Set	Not set	No	All events
Set	Set	Yes	All events

Table 3-120: Effects of using the ALL_TEXT_EVENTS and EDITABLE keywords

AM_PM

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Supplies a string array of two names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the [FORMAT](#) keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

APPEND

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

When using the [SET_VALUE](#) keyword to set the contents of a text widget (as created with the [WIDGET_TEXT](#) procedure), setting this keyword indicates that the supplied text should be appended to the existing contents of the text widget rather than replace it.

BAD_ID

This keyword applies to all widgets.

If *Widget_ID* is not a valid widget identifier, this [WIDGET_CONTROL](#) normally issues an error and causes program execution to stop. However, if [BAD_ID](#) is present and specifies a named variable, the invalid ID is stored into the variable, and this routine quietly returns. If no error occurs, a zero is stored.

BASE_SET_TITLE

This keyword applies to widgets created with the [WIDGET_BASE](#) function.

Set this keyword to a scalar string to change the title of the specified base widget. If the parent of the base widget is a tab widget, the title of the corresponding tab in the tab widget is changed to the provided string. If the parent is not a tab control, this keyword behaves like the keyword [TLB_SET_TITLE](#).

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

BITMAP

This keyword applies to widgets created with the [WIDGET_BUTTON](#) function.

Set this keyword to specify that the [SET_VALUE](#) keyword specifies the name of a .bmp file.

The value of a widget button can be a bitmap as described in “[Bitmap Button Labels](#)” on page 2280. If you specify the name of a bitmap file with the [VALUE](#) keyword, you must also set the [BITMAP](#) keyword.

Note

The use of bitmap values for buttons which have a button widget as their parent widget may cause unexpected behavior and is not supported.

CANCEL_BUTTON

This keyword applies to widgets created with the [WIDGET_BASE](#) function using the [MODAL](#) keyword.

Set this keyword equal to the widget ID of a button widget that will be the cancel button on a modal base widget.

On Motif and Windows platforms, selecting **Close** from the system menu (generally located at the upper left of the base widget) generates a button event for the **Cancel** button.

CLEAR_EVENTS

This keyword applies to all widgets.

If set, any events generated by the widget hierarchy rooted at *Widget_ID* which have arrived but have not been processed (via the `WIDGET_EVENT` procedure) are discarded.

COLUMN_LABELS

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword equal to an array of strings to be used as labels for the columns of the table. If no label is specified for a column, it receives the default label “*n*” where *n* is the column number. If this keyword is set to the empty string (“”), all column labels are set to be empty.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

COLUMN_WIDTHS

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword equal to a one-dimensional array of widths for the columns of the table widget. The widths are given in the units specified with the `UNITS` keyword. If no width is specified for a column, that column is set to the default size, which varies by platform. If `COLUMN_WIDTHS` is set to a scalar value, all of the column widths are set to that value.

If the `USE_TABLE_SELECT` keyword is set (in either standard or disjoint selection mode), the indices of the elements of an array specified as the value of the `COLUMN_WIDTHS` keyword are interpreted as indices into the list of selected columns. For example, if the table is in disjoint selection mode, the current selection includes cells from columns 0, 1, 2, 5, 6, and 7, `USE_TABLE_SELECT` is set equal to one, and the `COLUMN_WIDTHS` keyword is set equal to a four-element array [20, 35, 30, 35], the widths of columns 0, 1, 2, and 5 would be modified. Similarly, if the table is in standard selection mode and the `USE_TABLE_SELECT` keyword is set equal to the array [0, 1, 0, 0], and the `COLUMN_WIDTHS` keyword is set equal to a four-element array [20, 35, 30, 35], only the width of the first column would be altered.

COMBOBOX_ADDITEM

This keyword applies to widgets created with the `WIDGET_COMBOBOX` function.

Set this keyword to a non-null string that specifies a new item to add to the list of the combobox. By default, the item will be added to the end of the list. The item can be added to a specified position in the list by setting the `COMBOBOX_INDEX` keyword in the same call to `WIDGET_CONTROL`.

COMBOBOX_DELETEITEM

This keyword applies to widgets created with the `WIDGET_COMBOBOX` function.

Set this keyword to an integer that specifies the zero-based index of the combobox element to be deleted from the list. If the specified element is outside the range of existing elements, no element is deleted.

COMBOBOX_INDEX

This keyword applies to widgets created with the `WIDGET_COMBOBOX` function.

Set this keyword to an integer that specifies the zero-based index of the position at which a new item will be added to the list when using the `COMBOBOX_ADDITEM` keyword. The value -1 is a special case that indicates the item should be added to the end of the list. Note that this special case is provided for convenience, and that an item can also be added to the end of the list by omitting the `COMBOBOX_INDEX` keyword entirely. If the supplied index is not -1, and is outside the range of zero to the length of the existing list, the item is not added to the list.

Note

You can retrieve the length of the existing list using the `COMBOBOX_NUMBER` keyword to `WIDGET_INFO` function.

CONTEXT_EVENTS

This keyword applies to widgets created with the `WIDGET_BASE` function, `WIDGET_LIST`, `WIDGET_PROPERTY SHEET`, `WIDGET_TABLE`, and `WIDGET_TEXT` function.

Set this keyword to cause *context menu events* (or simply *context events*) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus); pass the context event ID to the `WIDGET_DISPLAYCONTEXTMENU` procedure within your widget program's event handler to display the context menu.

Note

You can also capture right mouse button events generated within draw widgets. See the `BUTTON_EVENTS` keyword to [WIDGET_DRAW](#) function for details.

For more on detecting and handling context menu events, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

DAYS_OF_WEEK

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Supplies a string array of 7 names to be used for the names of the days of the week when processing explicitly formatted dates (`CDWA`, `CDwA`, and `CdwA` format codes) with the `FORMAT` keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

DEFAULT_BUTTON

This keyword applies to widgets created with the [WIDGET_BASE](#) function using the `MODAL` keyword.

Set this keyword equal to the widget ID of a button widget that will be the default button on a modal base widget. The default button is highlighted by the window system.

DEFAULT_FONT

This keyword applies to all widgets. Do not specify a *Widget ID* when using this keyword.

A string containing the name of the default font to be used.

If the font to be used for a given widget is not explicitly specified (via the `FONT` keyword to the widget creation function), a default supplied by the window system or server is used. Use this keyword to change the default. See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

DELAY_DESTROY

This keyword applies to all widgets. Do not specify a *Widget ID* when using this keyword.

Normally, when the user destroys a widget hierarchy using the window manager, it is immediately removed. This can cause problems for applications that use the background task facility provided by the XMANAGER procedure if the hierarchy is destroyed while a background task is using it.

If DELAY_DESTROY is set, attempts to destroy the hierarchy are delayed until the next attempt to obtain an event for it. Setting DELAY_DESTROY to zero restores the default behavior.

XMANAGER uses this keyword automatically when managing background tasks. It is not expected that applications will need to use it directly.

DELETE_COLUMNS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to delete columns that contain selected cells. If a selection is specified via the [USE_TABLE_SELECT](#) keyword (in either standard or disjoint mode), the columns that contain cells specified in the selection array are deleted. If the USE_TABLE_SELECT keyword is *either* absent or set equal to 1, the columns that contain selected cells are deleted.

Warning

You cannot delete columns from a table which displays structure data in /ROW_MAJOR mode (the default).

DELETE_ROWS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to delete rows that contain selected cells. If a selection is specified via the [USE_TABLE_SELECT](#) keyword (in either standard or disjoint mode), the rows that contain cells specified in the selection array are deleted. If the

USE_TABLE_SELECT keyword is *either* absent or set equal to 1, the rows that contain selected cells are deleted.

Warning

You cannot delete rows from a table which displays structure data in /COLUMN_MAJOR mode.

DESTROY

This keyword applies to all widgets.

Set this keyword to destroy the widget and any child widgets in its hierarchy. Any further attempts to use the IDs for these widgets will cause an error.

DRAW_BUTTON_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to enable button press events for draw widgets. Setting a zero value disables such events.

Note

You can use button events generated by draw widgets to simulate the functionality of the CONTEXT_EVENTS keyword to WIDGET_BASE, WIDGET_LIST, and WIDGET_TEXT. See the BUTTON_EVENTS keyword to [WIDGET_DRAW](#) for details.

DRAW_EXPOSE_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to enable viewport expose events for draw widgets. Setting a zero value disables such events.

Note

You must explicitly disable backing store (by setting the RETAIN keyword to WIDGET_DRAW equal to zero) in order to generate expose events.

DRAW_KEYBOARD_EVENTS

This keyword applies to widgets created with [WIDGET_DRAW](#).

Set this keyword equal to 1 (one) or 2 to make the draw widget generate an event when it has the keyboard focus and a key is pressed or released. (The method by

which a widget receives the keyboard focus is dependent on the window manager in use.) The value of the key pressed is reported in either the `CH` or the `KEY` field of the event structure, depending on the type of key pressed. See “[Widget Events Returned by Draw Widgets](#)” on page 2365 for details.

- If this keyword is set equal to 1, the draw widget will generate an event when a “normal” key is pressed. “Normal” keys include all keys except function keys and the modifier keys: `SHIFT`, `CONTROL`, `CAPS LOCK`, and `ALT`. If a modifier key is pressed at the same time as a normal key, the value of the modifier key is reported in the `MODIFIERS` field of the event structure.
- If this keyword is set equal to 2, the draw widget will generate an event when *either* a normal key or a modifier key is pressed. Values for modifier keys are reported in the `KEY` field of the event structure, and the `MODIFIERS` field contains zero.

Note

Keyboard events are never generated for function keys.

DRAW_MOTION_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to enable motion events for draw widgets. Setting a zero value disables such events.

DRAW_VIEWPORT_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to enable viewport motion events for draw widgets. Setting a zero value disables such events.

DRAW_XSIZE

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to an integer that specifies the new horizontal size for the graphics region (the *virtual size*) of a draw widget in units specified by the `UNITS` keyword (pixels are the default). For non-scrollable draw widgets, setting this keyword is the same as setting `SCR_XSIZE` or `XSIZE`. However, for scrolling draw widgets `DRAW_XSIZE` is the only way to change the width of the drawable area (`XSIZE` sets the viewport size).

DRAW_YSIZE

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to an integer that specifies the new vertical size for the graphics region (the *virtual size*) of a draw widget in units specified by the UNITS keyword (pixels are the default). For non-scrollable draw widgets, setting this keyword is the same as setting SCR_YSIZE or YSIZE. However, for scrolling draw widgets DRAW_YSIZE is the only way to change the height of the drawable area (YSIZE sets the viewport size).

DYNAMIC_RESIZE

This keyword applies to widgets created with the [WIDGET_BUTTON](#), [WIDGET_COMBOBOX](#), [WIDGET_DROPLIST](#), and [WIDGET_LABEL](#) functions.

Set this keyword to activate (if set to 1) or deactivate (if set to 0) dynamic resizing of the specified WIDGET_BUTTON, WIDGET_LABEL, or WIDGET_DROPLIST widget (see the documentation for the DYNAMIC_RESIZE keyword to those procedures for more information about dynamic widget resizing).

EDITABLE

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#), [WIDGET_TABLE](#) and [WIDGET_TEXT](#) functions.

Set this keyword to directly edit the contents of a property sheet, text or table widget. The default value depends on the type of widget as follows:

- Text and table widgets are read-only by default (the default value is 0). The text cannot be modified. See the descriptions of the [ALL_TABLE_EVENTS](#) and [ALL_TEXT_EVENTS](#) keywords for additional details.
- Property sheet widgets are editable by default (EDITABLE=1). In this default mode, property values can be modified. When a property sheet is marked as read-only (EDITABLE=0), you can select properties and use any scrollbars, but cannot change any values. Setting this keyword to 0 is roughly equivalent to desensitizing all properties by setting the SENSITIVE attribute of *every* property on the property sheet equal to 0.

Note

To set the SENSITIVE attribute of individual properties within a property sheet, use [IDLitComponent::RegisterProperty](#) or [IDLitComponent::SetPropertyAttribute](#).

EDIT_CELL

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword equal to a two-element integer array containing the y (row) and x (column) coordinates of a table cell to put that cell into edit mode. For example, to put the top cell in the third column (x index=2) into edit mode, use the following commands:

```
row=0
col=2
WIDGET_CONTROL, table, EDIT_CELL=[row, col]
```

where *table* is the Widget ID of the table widget.

EVENT_FUNC

This keyword applies to all widgets.

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy given by *Widget_ID*.

This keyword overwrites any event routine supplied by previous uses of the `EVENT_FUNC` or `EVENT_PRO` keywords. To specify no event routine, set this keyword to a null string (' ').

EVENT_PRO

This keyword applies to all widgets.

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy given by *Widget_ID*.

This keyword overwrites any event routine supplied by previous uses of the `EVENT_FUNC` or `EVENT_PRO` keywords. To specify no event routine, set this keyword to a null string (' ').

FORMAT

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword equal to a single string or a one- or two-dimensional array of strings that specify the format of data displayed within table cells. The string(s) are of the same form as used by the `FORMAT` keyword to the `PRINT` procedure, and the default format is the same as that used by the [PRINT/PRINTF](#) procedures.

If the `USE_TABLE_SELECT` keyword is set equal to one, the format is changed only for the currently selected cells. If `USE_TABLE_SELECT` is set equal to an array, the format is changed for the specified cells.

- In standard selection mode, `FORMAT` can be set either to a single string or to a two-dimensional array of strings of the same size as the selected area.
- In disjoint selection mode, `FORMAT` can be set either to a single string or to a one-dimensional array of strings with the same number of elements as the selected area.

FUNC_GET_VALUE

This keyword applies to all widgets.

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. The function specified by `FUNC_GET_VALUE` is called with the widget ID as an argument. The function specified by `FUNC_GET_VALUE` should return a value for a widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GET_DRAW_VIEW

This keyword applies to widgets created with the `WIDGET_DRAW` function.

Specifies a named variable which will be assigned the current position of a draw widget viewport. The position is returned as a 2-element integer array giving the X and Y position relative to the lower left corner of the graphics area.

GET_UVALUE

This keyword applies to all widgets.

Set this keyword to a named variable to contain the current user value of the widget.

Each widget can contain a user set value of any data type and organization. This value is not used by the widget in any way, and exists entirely for the convenience of the IDL programmer. This keyword allows you to obtain the current user value.

The user value of a widget can be set with the `SET_UVALUE` keyword to this routine, or with the `UVALUE` keyword to the routine that created it.

To improve the efficiency of the data transfer, consider using the `NO_COPY` keyword (described below) with `GET_UVALUE`.

GET_VALUE

This keyword applies to widgets created with the [WIDGET_ACTIVEX](#), [WIDGET_BUTTON](#), [WIDGET_COMBOBOX](#), [WIDGET_DRAW](#), [WIDGET_DROPLIST](#), [WIDGET_LABEL](#), [WIDGET_SLIDER](#), [WIDGET_TABLE](#), and [WIDGET_TEXT](#) functions.

Note

If you would like information about the values returned for a specific compound widget—beginning with the prefix “CW_”—please refer to the description of the compound widget, which may also include a section titled, “Keywords to [WIDGET_CONTROL](#) and [WIDGET_INFO](#)”. Compound widgets are described in the *Reference Guide*.

Set this keyword to a named variable to contain the current value of the widget. The type of value returned depends on the widget type:

- **ActiveX:** An object reference to the IDLcomActiveX object that underlies the ActiveX widget. See [Chapter 5, “Using ActiveX Controls in IDL”](#) in the *External Development Guide* manual for details.
- **Button:** If the button label is text, it is returned as a string. Attempts to obtain the value of a button with a bitmap label is an error.
- **Combobox:** The contents of the list of the combobox widget are returned as a string or string array.
- **Draw:** The value of a draw widget depends on whether the draw widget uses IDL Direct Graphics or IDL Object Graphics. (The type of graphics used is specified by the [GRAPHICS_LEVEL](#) keyword to [WIDGET_DRAW](#).) The two possibilities are:
 - A. By default, draw widgets use IDL Direct Graphics. In this case, the value of a draw widget is the IDL window ID for the drawing area. This ID is used with procedures such as [WSET](#), [WSHOW](#), etc., to direct graphics to the widget. The window ID is assigned to drawing area widgets at the time they are realized. If the widget has not yet been realized, a value of -1 is returned.
 - B. If the draw widget uses IDL Object Graphics (that is, if the [GRAPHICS_LEVEL](#) keyword to [WIDGET_DRAW](#) is set equal to 2), the value of the draw widget is the object reference of the window object used in the draw widget.

- Droplist: The contents of the droplist widget's list are returned as a string or string array.
- Label: The label text is returned as a string.
- Property Sheet: Retrieves the component object(s) that the property sheet is associated with. Use the GET_VALUE keyword to WIDGET_CONTROL.
- Slider: The current value of the slider is returned as an integer.
- Table: Normally, the data for the whole table are returned as a two dimensional array or a vector of structures.

If the **USE_TABLE_SELECT** keyword is set, the value returned is a subset of the whole data.

- If the table is in standard selection mode and the table data is of a single type, the value returned is a two dimensional array. If the table contains structure data, the value returned is a vector of (possibly anonymous) structures.
- If the table is in disjoint selection mode and the table data is a single type, the value returned is a one-dimensional array of values. If the table contains structure data, the value returned is a single structure in which each field corresponds to a selected cell. In either case, use the **TABLE_SELECT** keyword to **WIDGET_INFO** to get the row and column indices of the disjoint values.

If the **USE_TEXT_SELECT** keyword is set, the value returned is a string corresponding to the currently-selected text in the currently-selected cell.

- Text: The current contents of the text widget are returned as a string array. If the **USE_TEXT_SELECT** keyword is also specified, only the contents of the current selection are returned.
- Widget types not listed above do not return a value. Attempting to retrieve the value of such a widget causes an error.

The value of a widget can be set with the **SET_VALUE** keyword to this routine, or with the **VALUE** keyword to the routine that created it.

GROUP_LEADER

This keyword applies to all widgets.

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

HOURLASS

This keyword applies to all widgets. Do not specify a *Widget ID* when using this keyword.

Set this keyword to turn on an “hourglass-shaped” cursor for all IDL widgets and graphics windows. The hourglass remains in place until the `WIDGET_EVENT` function attempts to process the next event. Then the previous cursor is reinstated. If an application starts a time-intensive calculation inside an event-handling routine, the hourglass cursor should be used to indicate that the system is not currently responding to events.

ICONIFY

This keyword applies to all widgets.

Set this keyword to a non-zero value to cause the specified widget to become iconified. Set this keyword to zero to open an iconified widget.

INPUT_FOCUS

This keyword applies to widgets created with the `WIDGET_BUTTON`, `WIDGET_DRAW`, and `WIDGET_TEXT` functions.

If *Widget ID* is a text widget or a draw widget, set this keyword to cause the widget to receive the keyboard focus. If *Widget ID* is a button widget, set this keyword to position the mouse pointer over the button (on Motif), or set the focus to the button so that it can be “pushed” with the spacebar (on Windows). This keyword has no effect for other widget types.

Note

You cannot assign the input focus to an unrealized widget.

INSERT_COLUMNS

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword to the number of columns to be added to the right of the rightmost column of the table. If the `USE_TABLE_SELECT` keyword is set equal to one, the columns are inserted to the left of the current selection. If `USE_TABLE_SELECT` is set equal to an array, the columns are inserted to the left of the specified selection.

Warning

You cannot insert columns into a table which displays structure data in /ROW_MAJOR (default) mode because it would change the structure.

INSERT_ROWS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to the number of rows to be added below the bottommost row of the table. If the [USE_TABLE_SELECT](#) keyword is set equal to one, the rows are inserted above the current selection. If [USE_TABLE_SELECT](#) is set equal to an array, the rows are inserted above the specified selection.

Warning

You cannot insert rows into a table which displays structure data in /COLUMN_MAJOR mode because it would change the structure.

KBRD_FOCUS_EVENTS

This keyword applies to widgets created with the [WIDGET_BASE](#), [WIDGET_TABLE](#), and [WIDGET_TEXT](#) functions.

Set this keyword to cause widget keyboard focus events to be issued for the widget whenever the keyboard focus of that widget changes. See the [KBRD_FOCUS_EVENTS](#) keywords to [WIDGET_BASE](#), [WIDGET_TABLE](#), and [WIDGET_TEXT](#) for details.

KILL_NOTIFY

This keyword applies to all widgets.

Use this keyword to change or remove a previously-specified callback procedure for *Widget_ID*. A previously-defined callback can be removed by setting this keyword to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the [WIDGET_CONTROL](#) procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the [WIDGET_EVENT](#) function is called.

For a top-level base widget in a widget application, the [CLEANUP](#) keyword to [XMANAGER](#) can also be used to specify a procedure that will be called when the base widget dies. Note that the *last* call to any of:

- WIDGET_BASE, KILL_NOTIFY
- XMANAGER, CLEANUP
- WIDGET_CONTROL, KILL_NOTIFY

determines the procedure that is executed when the specified top-level base widget dies. Calling XMANAGER with the CLEANUP keyword overrides any previous setting of KILL_NOTIFY. Similarly, calling WIDGET_CONTROL with KILL_NOTIFY overrides any previous setting of CLEANUP.

MANAGED

This keyword applies to all widgets.

This keyword is used by the XMANAGER procedure to mark those widgets that it is currently managing. User applications should not use this keyword directly.

MAP

This keyword applies to all widgets.

Set this keyword to zero to unmap the widget hierarchy rooted at the widget specified by *Widget_ID*. The hierarchy disappears from the screen, but still exists.

The mapping operation applies only to base widgets. If the specified widget is not a base, IDL searches upward in the widget hierarchy until it finds the closest base widget. The map operation is applied to that base.

Set MAP to a nonzero value to re-map the widget hierarchy and make it visible. Normally, the widget is automatically mapped when it is realized, so use of the MAP keyword is not required.

MONTHS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Supplies a string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMOA, and CmoA format codes) with the FORMAT keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

MULTIPLE_PROPERTIES

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function.

Set this property to 1 to allow multiple properties to be selected at the same time. When this keyword is set, you can hold down the **Ctrl** key while left-clicking with the mouse to make nonadjacent selections, or hold down the **Shift** key to select an adjacent range of properties.

When changing the selection support of a property sheet from multiple selection to single selection (`MULTIPLE_PROPERTIES` equals 0), the current set of selected properties is cleared unless there is only a single selection.

NO_COPY

This keyword applies to all widgets.

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

Note

The `NO_COPY` keyword increases efficiency when sending event structures using the `SEND_EVENT` keyword to `WIDGET_CONTROL`.

NO_NEWLINE

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

When setting the value of a multi-line text widget, newline characters are automatically appended to the end of each line of text. The `NO_NEWLINE` keyword suppresses this action.

NOTIFY_REALIZE

This keyword applies to all widgets.

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. A previously-set callback routine can be removed by setting this keyword to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

This keyword applies to all widgets.

A string containing the name of a procedure to be called when the SET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. The procedure specified by PRO_SET_VALUE is called with 2 arguments— a widget ID and a value. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

PROPERTY SHEET_SELECTED

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function.

This keyword allows programmatic property selection. Set it to a string or an array of strings to identify the properties to appear selected. The strings should match valid property identifiers. When this keyword is set to an empty string, or an array that contains only empty strings, it clears all property selections.

PUSHBUTTON_EVENTS

This keyword applies to widgets created with the [WIDGET_BUTTON](#) function.

Set this keyword to a non-zero value to enable pushbutton events for the widget specified by *Widget_ID*. Set the keyword to 0 to disable pushbutton events for the specified widget. For a description of pushbutton events, see “[PUSHBUTTON_EVENTS](#)” on page 2287 in the documentation for WIDGET_BUTTON.

REALIZE

This keyword applies to all widgets.

If set, the widget hierarchy is realized. Until the realization step, the widget hierarchy exists only within IDL. Realization is the step of actually creating the widgets on the screen (and mapping them if necessary).

When a previously realized widget gets a new child widget, the new child is automatically realized.

Tip

Under Microsoft Windows, when a hidden base is realized, then mapped, a Windows resize message is sent by the windowing system. This “extra” resize event is generated before any manipulation of the base widget by the user.

REFRESH_PROPERTY

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function. Set this keyword to:

- A non-zero numeric value to refresh all properties. Refreshing all properties in this manner is more efficient to use `/REFRESH_PROPERTY` than `WIDGET_CONTROL`'s `SET_VALUE` keyword to reload the property sheet.
- A string or string array containing one or more *property identifiers* identifying individual properties to be refreshed. See “[Property Identifiers](#)” in Chapter 4 of the *iTool Developer's Guide* manual for information on property identifiers.

The `REFRESH_PROPERTY` keyword also updates a property's sensitivity and visibility according to the value of the property held by the related component. Although the property sheet knows about its component, the property sheet and underlying component have a clear boundary and can only affect each other through IDL statements. The following example illustrates this. The first selected property in the property sheet is hidden after a 4 second wait as long as the `REFRESH_PROPERTY` keyword is used. When commented out, there is no notification that the first selected property should be hidden.

```
PRO ExPropSheetRefresh

; Create and initialize the component.
oComp = OBJ_NEW('IDLitVisAxis')

; Create a base and property sheet.
base = WIDGET_BASE(/TLB_SIZE_EVENT, $
    TITLE = 'Single Property Sheet Example')
wPropAxis = WIDGET_PROPERTY SHEET(base, VALUE = oComp, $
    UNAME = 'PropSheet', xsize = 45)
; Pre-select the color and transparency properties of
; axis component.
```

```

WIDGET_CONTROL, wPropAxis, $
    PROPERTYSHEET_SETSELECTED=['Color', 'Transparency']

; Activate the widgets.
WIDGET_CONTROL, base, SET_UVALUE = oComp, /REALIZE
XMANAGER, 'ExSinglePropSheet', base, /NO_BLOCK

; Wait 4 seconds and then hide the first selected property
; and refresh the property sheet.
WAIT, 4
a = WIDGET_INFO(wPropAxis, /propertySheet_selected)
oComp->SetPropertyAttribute, a[0], /HIDE
WIDGET_CONTROL, wPropAxis, /REFRESH_PROPERTY

END

```

RESET

This keyword applies to all widgets. Do not specify a *Widget ID* when using this keyword. Set the RESET keyword to destroy every currently active widget. This keyword should be used with caution.

ROW_LABELS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword equal to an array of strings to be used as labels for the rows of the table. If no label is specified for a row, it receives the default label “*n*” where *n* is the row number. If this keyword is set to the empty string (“”), all row labels are set to be empty.

ROW_HEIGHTS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Note

This keyword is not supported under Microsoft Windows.

Set this keyword equal to a one-dimensional array of heights for the rows of the table widget. The heights are given in the units specified with the UNITS keyword. If no height is specified for a row, that row is set to the default size, which varies by platform. If ROW_HEIGHTS is set to a scalar value, all of the row heights are set to that value.

If the [USE_TABLE_SELECT](#) keyword is set (in either standard or disjoint selection mode), the indices of the elements of the specified array are interpreted as indices

into the list of selected columns. For example, if the table is in disjoint selection mode, the current selection includes cells from rows 0, 1, 2, 5, 6, and 7, `USE_TABLE_SELECT` is set equal to one, and the `ROW_HEIGHTS` keyword is set equal to a four-element array [20, 35, 30, 35], the height of rows 0, 1, 2, and 5 would be modified. Similarly, if the table is in standard selection mode and the `USE_TABLE_SELECT` keyword is set equal to the array [0, 1, 0, 0], and the `ROW_HEIGHTS` keyword is set equal to a four-element array [20, 35, 30, 35], only the height of the first row would be altered.

SCR_XSIZE

This keyword applies to all widgets.

Set this keyword to an integer value that represents the widget's new horizontal size, in units specified by the `UNITS` keyword (pixels are the default). Attempting to change the size of a widget that is part of a menubar or pulldown menu causes an error. This keyword is useful for resizing table, text, list, and scrolling widgets. Note that `[XY]SIZE` sets client area and `SCR_[XY]SIZE` sets total area (title bar, borders, menu, client area).

SCR_YSIZE

This keyword applies to all widgets.

Set this keyword to an integer value that represents the widget's new vertical size, in units specified by the `UNITS` keyword (pixels are the default). Attempting to change the size of a widget that is part of a menubar or pulldown menu causes an error. This keyword is useful for resizing table, text, list, and scrolling widgets. Note that `[XY]SIZE` sets client area and `SCR_[XY]SIZE` sets total area (title bar, borders, menu, client area).

SEND_EVENT

This keyword applies to all widgets.

Set this keyword to a structure containing a valid widget event to be sent to the specified widget. The value of `SEND_EVENT` *must* be a structure and the first three fields must be `ID`, `TOP`, and `HANDLER` (all of `LONG` type). Additional fields can be of any type.

As a convenience, you can set the values of the ID, TOP, and HANDLER fields to the long integer zero. Setting these fields to zero will result in their being populated with the following values:

ID	the widget ID of the widget specified in the call to WIDGET_CONTROL.
TOP	the widget ID of the top-level base containing ID.
HANDLER	the widget ID of the widget associated with the event handling routine for ID.

To improve the efficiency of the data transfer, consider using the NO_COPY keyword with SEND_EVENT.

SENSITIVE

Set this keyword to control the sensitivity state of a widget after creation. This keyword applies to all widgets. Use the SENSITIVE keyword with the widget creation function to control the initial sensitivity state.

When a widget is sensitive, it has normal appearance and can receive user input. For instance, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, and ignores any input directed at it. If SENSITIVE is zero, the widget hierarchy becomes insensitive. If nonzero, it becomes sensitive.

Sensitivity can be used to control when a user is allowed to manipulate a widget. It should be noted that some widgets do not change their appearance when they are made insensitive, and simply cease generating events.

SET_BUTTON

This keyword applies to widgets created with the [WIDGET_BUTTON](#) function.

This keyword changes the current state of toggle buttons. If set equal to zero, every toggle button in the hierarchy specified by *Widget_ID* is set to the unselected state. If set to a nonzero value, the action depends on the type of base holding the buttons:

- For a non-exclusive base:
 - If a single button is specified by *Widget_ID*, that button is set to the selected state, leaving the state of other buttons in the base unchanged.

- If the base itself is specified by *Widget_ID*, all buttons are set to the selected state.
- For an exclusive base:
 - If a single button is specified by *Widget_ID*, that button is set to the selected state, and all other buttons in the base are set to the unselected state.
 - If the base itself is specified by *Widget_ID*, IDL generates an error, since an exclusive base can contain at most one selected button.

If the specified *Widget_ID* is a button widget that was created using the `CHECKED_MENU` keyword, the checked state of the menu item is modified. If the keyword value is nonzero, the menu button is placed in a checked state. If the keyword value is zero, the button is placed in a un-checked state.

SET_COMBOBOX_SELECT

This keyword applies to widgets created with the `WIDGET_COMBOBOX` function.

Set this keyword to an integer that specifies the zero-based index of the combobox list element to be displayed. If the specified element is outside the range of existing elements, the selection remains unchanged.

SET_DRAW_VIEW

This keyword applies to widgets created with the `WIDGET_DRAW` function.

A scrollable draw widget provides a large graphics area which is viewed through a smaller viewport. This keyword allows changing the current position of the viewport. The desired position is specified as a 2-element integer array giving the X and Y position in units specified by the `UNITS` keyword (pixels are the default) relative to the lower left corner of the graphics area. For example, to position the viewport to the lower left corner of the image:

```
WIDGET_CONTROL, widget, SET_DRAW_VIEW=[0, 0]
```

SET_DROPLIST_SELECT

This keyword applies to widgets created with the `WIDGET_DROPLIST` function.

Set this keyword to an integer that specifies the droplist element to be current (i.e., the element that is displayed on the droplist button). Positions start at zero. If the specified element is outside the possible range, no new selection is set.

SET_LIST_SELECT

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword to an integer scalar or vector that specifies the list element or elements to be highlighted. The previous selection (if there is a selection) is cleared. Positions start at zero. If the specified element is outside the possible range, no new selection is set. Note that the [MULTIPLE](#) keyword to [WIDGET_LIST](#) must have been set if more than a single list element is specified.

If the selected position is not currently on the screen, the list widget automatically move the current scrolling viewport to make it visible. The resulting topmost visible element is toolkit specific. If you wish to ensure a certain element is at the top of the list, use the [SET_LIST_TOP](#) keyword (described below) to explicitly set the viewport.

SET_LIST_TOP

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword to an integer that specifies the element of the list widget to the positioned at the top of the scrolling list. If the specified element is outside the range of list elements, nothing happens.

SET_SLIDER_MAX

This keyword applies to widgets created with the [WIDGET_SLIDER](#) function.

Set this keyword to a new maximum value for the specified slider widget.

Note

This keyword does not apply to floating-point sliders created with the [CW_FSLIDER](#) function.

SET_SLIDER_MIN

This keyword applies to widgets created with the [WIDGET_SLIDER](#) function.

Set this keyword to a new minimum value for the specified slider widget.

Note

This keyword does not apply to floating-point sliders created with the [CW_FSLIDER](#) function.

SET_TAB_CURRENT

This keyword applies to widgets created with the `WIDGET_TAB` function.

Set this keyword equal to the zero-based index of the tab to be set as the current (visible) tab. If the index value is invalid, the value is quietly ignored.

SET_TAB_MULTILINE

This keyword applies to widgets created with the `WIDGET_TAB` function.

This keyword controls how tabs appear on the tab widget when all of the tabs do not fit on the widget in a single row. This keyword behaves differently on Windows and UNIX systems.

Windows

Set this keyword to cause tabs to be organized in a multi-line display when the width of the tabs exceeds the width of the largest child base widget. If possible, IDL will create tabs that display the full tab text.

If `MULTILINE=0` and `LOCATION=0` or `1`, tabs that exceed the width of the largest child base widget are shown with scroll buttons, allowing the user to scroll through the tabs while the base widget stays immobile.

If `LOCATION=1` or `2`, a multiline display is always used if the tabs exceed the height of the largest child base widget.

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. The text of the tabs (the titles of the tab widget's child base widgets) may be truncated even if the `MULTILINE` keyword is set.

UNIX

Set this keyword equal to an integer that specifies the maximum number of tabs to display per row in the tab widget. If this keyword is not specified (or is explicitly set equal to zero) all tabs are placed in a single row.

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. The text of the tabs (the titles of the tab widget's child base widgets) is never truncated in order to make the tabs fit the space available. (Tab text may be truncated if the text of a single tab exceeds the

space available, however.) This means that if `MULTILINE` is set to any value other than one, some tabs may not be displayed.

SET_TABLE_SELECT

This keyword applies to widgets created with the `WIDGET_TABLE` function.

- In standard selection mode, specify a four-element array, of the form [*left*, *top*, *right*, *bottom*] specifying the of cells to act upon.
- In disjoint selection mode, specify a 2 x *n* element array of column/row pairs specifying the cells to act upon.

Specifications for cell locations are zero-based (that is, the first data column is column number zero). The value -1 is used to refer to the title row or title column.

See “`USE_TABLE_SELECT`” on page 2347 for details on the selection modes.

Note

To remove a table selection programmatically, set this keyword to either [-1, -1, -1, -1] (in standard selection mode) or [[-1, -1]] (in disjoint selection mode).

If the selected position is not currently on the screen, the table widget automatically moves the current scrolling viewport to make a portion of it visible. The resulting top-left visible cell is toolkit specific. If you wish to ensure a certain element is at the top of the list, use the `SET_TABLE_VIEW` keyword to explicitly set the viewport.

SET_TABLE_VIEW

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword to a two-element array of zero-based cell indices that specifies the cell of the table widget to be positioned at the top-left of the widget. If the specified cell is outside the range of valid cells, nothing happens.

SET_TEXT_SELECT

This keyword applies to widgets created with the `WIDGET_TABLE` and `WIDGET_TEXT` functions.

Use this keyword to clear any current selection in the specified table cell or text widget, and either set a new selection, or simply set the text insertion point. To set a selection, specify a two-element integer array containing the starting position and the

length of the selection. For example, to set a selection covering characters 3 through 23:

```
WIDGET_CONTROL, widgetID, SET_TEXT_SELECT=[3, 20]
```

To move the text insertion point without setting a selection, omit the second element, or set it to zero.

SET_TEXT_TOP_LINE

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to the zero-based line number of the line to be positioned on the topmost visible line in the text widget's viewport. No horizontal scrolling is performed. Note that this is a line number, not a character offset.

SET_TREE_BITMAP

This keyword applies to widgets created with the [WIDGET_TREE](#) function.

Set this keyword equal to a 16x16x3 array representing an RGB image that will be displayed next to the node in the tree widget.

Set this keyword equal to zero to revert to the appropriate default system bitmap.

SET_TREE_EXPANDED

This keyword applies to widgets created with the [WIDGET_TREE](#) function.

Set this keyword equal to a nonzero value to expand the specified tree widget folder. Set this keyword equal to zero to collapse the specified tree widget folder.

SET_TREE_SELECT

This keyword applies to widgets created with the [WIDGET_TREE](#) function.

This keyword has two modes of operation, depending on the widget ID passed to [WIDGET_CONTROL](#):

- If the specified widget ID is for the root node of the tree widget (the tree widget whose *Parent* is a base widget):
 - If the tree widget is in multiple-selection mode and [SET_TREE_SELECT](#) is set to an array of widget IDs corresponding to tree widgets that are nodes in the tree, those nodes are selected.
 - If the tree widget is *not* in multiple-selection mode and [SET_TREE_SELECT](#) is set to a single widget ID corresponding to a tree widget that is a node in the tree, that node is selected.

- If the keyword is set to zero, all selections in the tree widget are cleared.
- If the specified widget ID is a tree widget that is a node in a tree:
 - If the keyword is set to a nonzero value, the specified node is selected.
 - If the keyword is set to zero, the specified node is deselected.

Note

If the tree widget is in multiple-selection mode, the selection changes made to the tree widget via this keyword are additive — that is, the current selections are retained and any additional nodes specified by `SET_TREE_SELECT` are also selected.

SET_TREE_VISIBLE

This keyword applies to widgets created with the `WIDGET_TREE` function and whose parent widget was also created using the `WIDGET_TREE` function (that is, tree widgets that are nodes of another tree).

Set this keyword to make the specified tree node *visible* to the user. Setting this keyword has two possible effects:

1. If the specified node is inside a collapsed folder, the folder and all folders above it are expanded to reveal the node.
2. If the specified node is in a portion of the tree that is not currently visible because the tree has scrolled within the parent base widget, the tree view scrolls so that the selected node is at the top of the base widget.

Note

Use of this keyword does not affect the tree widget selection state.

SET_UNAME

This keyword applies to all widgets.

Set this keyword to a string that can be used to identify the widget. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID. You can set the name at creation time, using the `UNAME` keyword with the creation function.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

SET_UVALUE

This keyword applies to all widgets.

Each widget can contain a user-set value. This value is not used by IDL in any way, and exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value.

To improve the efficiency of the data transfer, consider using the `NO_COPY` keyword with `SET_UVALUE`.

SET_VALUE

This keyword applies to widgets created with the `WIDGET_BUTTON`, `WIDGET_COMBOBOX`, `WIDGET_DROPLIST`, `WIDGET_LABEL`, `WIDGET_LIST`, `WIDGET_SLIDER`, `WIDGET_TABLE`, and `WIDGET_TEXT` functions.

Sets the value of the specified widget. The meaning of the value differs between widget types:

- **Button:** The label to be used for the button. This value can be either a scalar string, or a 2D byte array containing a bitmap.
- **Combobox:** The contents of the combobox widget (string or string array).
- **Droplist:** The contents of the droplist widget (string or string array).
- **Label:** The text to be displayed by the label widget.
- **List:** The contents of the list widget (string or string array).
- **Property Sheet:** Associates one or more component objects with a property sheet. Any existing associations are lost and the property sheet is reloaded with the new list of properties. Use the `SET_VALUE` keyword to `WIDGET_CONTROL`. Use the return value of `OBJ_NEW()` to clear out a property sheet.
- **Slider:** The current position of the slider (integer).
- **Table:** Normally, the data for the whole table is changed to the specified data, which must be of the form of a two dimensional array or a vector of structures. In this form, the table is resized to fit the given data (unless the `TABLE_XSIZE` or `TABLE_YSIZE` keywords are given).

If the `USE_TABLE_SELECT` keyword is set, the value given is treated as a subset of the whole data, and only the given range of cells are updated. Used in this form, the type of data stored in the table cannot be changed. The data

passed in is converted, as appropriate, to the type of the selected cells. If less data is passed in than fits in the current selection, the cells outside the range of data (but inside the selection) are left unchanged. If more data is passed in than fits in the current selection, the extra data is ignored.

- If the table is in standard selection mode, the data should be a two-dimensional array of values for a single-type table, or a vector of structures for a table that contains structure data.
- If the table is in disjoint selection mode, the data should be a one-dimensional array of values for a single-type table, or a single structure with each field corresponding to a cell for a table that contains structure data.

If the `USE_TEXT_SELECT` keyword is present, the value must be a string, which replaces the currently-selected text in the currently-selected cell.

- Text: The text to be displayed. If the `APPEND` keyword is also specified, the text is appended to the current contents instead of completely replacing it (string or string array). If the `USE_TEXT_SELECT` keyword is specified, the new string replaces only the currently-selected text in the text widget.
- Widget types not listed above do not allow the setting of a value. Attempting to set the value of such a widget causes an error.

The value of a widget can also be set with the `VALUE` keyword to the routine that created it.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

SHOW

This keyword applies to all widgets.

Controls the visibility of a widget hierarchy. If set to zero, the hierarchy containing *Widget_ID* is pushed behind any other windows on the screen. If nonzero, the hierarchy is pulled in front.

TAB_MODE

This keyword applies to widgets created with the [WIDGET_BASE](#), [WIDGET_BUTTON](#), [WIDGET_COMBOBOX](#), [WIDGET_DROPLIST](#),

[WIDGET_LIST](#), [WIDGET_SLIDER](#), [WIDGET_TAB](#), [WIDGET_TABLE](#), [WIDGET_TEXT](#), [WIDGET_TREE](#).

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited at creation time by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the `TAB_MODE` value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

When `WIDGET_CONTROL` is used to change the `TAB_MODE` value, the change affects only the widget upon which it is explicitly set. The change is not propagated along the widget hierarchy after it has been created. The `TAB_MODE` keyword also *cannot* be explicitly set for the following widgets:

- Draw widgets
- Label widgets
- Grouped, exclusive button widgets (radio buttons)
- Menu items or menu bases

Attempting to set `TAB_MODE` on these widgets will generate an error.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default value unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-121: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero and any attempt to change it is ignored when running a widget application on the UNIX platform.

After creating the widget hierarchy, you can query a widget's support for tabbing using the `WIDGET_INFO` procedure's `TAB_MODE` keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TABLE_BLANK

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword equal to a nonzero value to cause the specified cells to be blank. Set this keyword equal to zero to cause the specified cells to display values as usual.

Note

Hiding cell contents using this keyword is a *display-only operation*. The cell contents are not actually removed from the table, they are simply hidden from view. Any operation that evaluates or changes the value of a cell will operate in the same way on a cell whose contents are hidden as it will on a cell whose contents are visible.

If the `USE_TABLE_SELECT` keyword is set equal to one, the currently selected cells are blanked or restored. If `USE_TABLE_SELECT` is set equal to an array, the specified cells are blanked or restored. If `USE_TABLE_SELECT` is not set, the entire table is hidden or displayed.

TABLE_DISJOINT_SELECTION

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword to enable the ability to select multiple rectangular regions of cells.

TABLE_XSIZE

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword equal to the number of data columns in the table widget. Note that if the table widget was created with row titles enabled (that is, if the `NO_HEADERS` or `NO_COLUMN_HEADERS` keywords to `WIDGET_TABLE` were *not* set), the table will contain one column more than the number specified by `TABLE_XSIZE`.

If the table is made smaller as a result of the application of the `TABLE_XSIZE` keyword, the data outside the new range persists, but the number of columns and/or rows changes as expected. If the table is made larger, the data type of the cells in the new columns is set according to the following rules:

1. If the table was not created with either the `ROW_MAJOR` or `COLUMN_MAJOR` keywords set (if the table is an array rather than a vector of structures), the new cells have the same type as all the original cells.
2. If the `SET_VALUE` keyword is given, the types of all columns are set according to the new structure.
3. If the table was created with the `ROW_MAJOR` keyword set, and the `SET_VALUE` keyword is not specified, the cells in the new columns inherit their type from the cells to their left.
4. If the table was created with the `COLUMN_MAJOR` keyword set, and the `SET_VALUE` keyword is not specified, any new columns default to type `INT`.

TABLE_YSIZE

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword equal to the number of data rows in the table widget. Note that if the table widget was created with column titles enabled (that is, if the `NO_HEADERS` or `NO_ROW_HEADERS` keywords to `WIDGET_TABLE` were *not* set), the table will contain one row more than the number specified by `TABLE_YSIZE`.

If the table is made smaller as a result of the application of the `TABLE_YSIZE` keyword, the data outside the new range persists, but the number of columns and/or rows changes as expected. If the table is made larger, the data type of the cells in the new rows is set according to the following rules:

1. If the table was not created with either the `ROW_MAJOR` or `COLUMN_MAJOR` keywords set (if the table is an array rather than a vector of structures), the new cells have the same type as all the original cells.
2. If the `SET_VALUE` keyword is given, the types of all rows are set according to the new structure.
3. If the table was created with the `COLUMN_MAJOR` keyword set, and the `SET_VALUE` keyword is not specified, the cells in the new rows inherit their type from the cells above.
4. If the table was created with the `ROW_MAJOR` keyword set, and the `SET_VALUE` keyword is not specified, any new rows default to type `INT`.

TIMER

This keyword applies to all widgets.

If this keyword is present, a `WIDGET_TIMER` event is generated. Set this keyword to a floating-point value that represents the number of seconds before the timer event arrives. Note that this event is identical to any other widget event except that it contains only the 3 standard event tags. These event structures are defined as:

```
{ WIDGET_TIMER, ID:0L, TOP:0L, HANDLER:0L }
```

It is left to the caller to tell the difference between standard widget events and timer events. The standard way to do this is to use a widget that doesn't normally generate events (e.g., a base or label). Alternately, the `TAG_NAMES` function can be called with the `STRUCTURE_NAME` keyword to differentiate a `WIDGET_TIMER` event from other types of events. For example:

```
IF TAG_NAMES(event, /STRUCTURE_NAME) EQ $
    'WIDGET_TIMER' THEN ...
```

Using the `TIMER` keyword is more efficient than the background task functionality found in the `XMANAGER` procedure because it doesn't "poll" like the original background task code. `RSI` will eventually eliminate the background task functionality from `XMANAGER`. We encourage all users to modify their code to use the `TIMER` keyword instead.

TLB_GET_OFFSET

This keyword applies to all widgets.

Set this keyword to a named variable in which the offset of the upper left corner of the top-level base of the specified widget is returned, in units specified by the `UNITS` keyword (pixels are the default). The offset is measured in device coordinates relative to the upper-left corner of the screen.

Note

On Windows and Macintosh platforms, the returned position is the upper-left corner of the window *including* border and title bar. On non-Macintosh Motif platforms, the returned position represents the upper-left corner of the window's client area (not including border and titlebar).

TLB_GET_SIZE

This keyword applies to all widgets.

Set this keyword to a named variable in which the size of the top-level base of the specified widget is returned, in units specified by the UNITS keyword (pixels are the default). The size is returned as a two-element vector that contains the horizontal and vertical size of the base in device coordinates.

TLB_ICONIFY_EVENTS

This keyword applies to widgets created with the [WIDGET_BASE](#) function.

Set this keyword to make the top-level base return an event when the base is iconified or restored by the user.

TLB_KILL_REQUEST_EVENTS

This keyword applies to widgets created with the [WIDGET_BASE](#) function.

Use this keyword to set or clear kill request events for the specified top-level base. For more information on these events see “[TLB_KILL_REQUEST_EVENTS](#)” on page 2272.

TLB_MOVE_EVENTS

This keyword applies to widgets created with the [WIDGET_BASE](#) function.

Set this keyword to make the top-level base return an event when the base is moved by the user. Note that if [TLB_SIZE_EVENTS](#) are also enabled, a user resize operation that causes the top left corner of the base widget to move will generate both a move event and a resize event.

TLB_SET_TITLE

This keyword applies to all widgets.

Set this keyword to a scalar string to change the title of the specified top-level base after it has been created.

TLB_SET_XOFFSET

This keyword applies to all widgets.

Use this keyword to set the horizontal position of the top level base of the specified widget. The offset is measured from the upper-left corner of the screen to the upper-left corner of the base, in units specified by the UNITS keyword (pixels are the default).

TLB_SET_YOFFSET

This keyword applies to all widgets.

Use this keyword to set the vertical position of the top-level base of the specified widget. The offset is measured from the upper-left corner of the screen to the upper-left corner of the base, in units specified by the UNITS keyword (pixels are the default).

TLB_SIZE_EVENTS

This keyword applies to widgets created with the [WIDGET_BASE](#) function.

Set this keyword to make the top-level base return an event when the base is resized by the user. Note that if [TLB_MOVE_EVENTS](#) are also enabled, a user resize operation that causes the top left corner of the base widget to move will generate both a move event and a resize event.

TOOLTIP

This keyword applies to widgets created with the [WIDGET_BUTTON](#) and [WIDGET_DRAW](#) functions.

Set this keyword to a string that will be displayed when the cursor hovers over the specified widget. For UNIX platforms, this string must be non-zero in length, which means that a tooltip can be modified but not be removed on UNIX versions of IDL.

Note

Tooltips cannot be created for menu sub-items. The topmost button on a menu can, however, have a tooltip.

Note

If your application uses hardware rendering and a [RETAIN](#) setting of either zero or one, tooltips will cause draw widgets to generate expose events if the tooltip obscures the drawable area. This is true even if the tooltip is associated with another widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

TRACKING_EVENTS

This keyword applies to all widgets.

Set this keyword to a non-zero value to enable tracking events for the widget specified by *Widget_ID*. Set the keyword to 0 to disable tracking events for the specified widget. For a description of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for `WIDGET_BASE`.

UNITS

This keyword applies to all widgets.

Use this keyword to specify the unit of measurement used for most widget sizing operations. Set `UNITS` equal to 0 to specify that all measurements are in pixels (this is the default), to 1 to specify that all measurements are in inches, or to 2 to specify that all measurements are in centimeters.

Note

This keyword does not affect all sizing operations. Specifically, the value of `UNITS` is ignored when setting the `XSIZE` or `YSIZE` keywords to [WIDGET_LIST](#), [WIDGET_TABLE](#), or [WIDGET_TEXT](#) functions.

UPDATE

This keyword applies to all widgets.

Use this keyword to enable (if set to 1) or disable (if set to 0) screen updates for the widget hierarchy to which the specified widget belongs. This keyword is useful for preventing unwanted intermediate screen updates when changing the values of many widgets at once or when adding several widgets to a previously-realized widget hierarchy. When first realized, widget hierarchies are set to update.

Note

Although this keyword is legal for any widget, it affects every widget in the hierarchy defined for the specified widget’s top-level base widget. In other words, setting `UPDATE` on a particular widget is equivalent to setting `UPDATE` on the top-level base widget that is at the top of the specified widget’s hierarchy.

Warning

Do not attempt to resize a widget on the Windows platform while `UPDATE` is turned off. Doing so may prevent IDL from updating the screen properly when `UPDATE` is turned back on.

USE_TABLE_SELECT

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to modify the behavior of the [ALIGNMENT](#), [COLUMN_WIDTHS](#), [DELETE_COLUMNS](#), [DELETE_ROWS](#), [FORMAT](#), [GET_VALUE](#), [INSERT_COLUMNS](#), [INSERT_ROWS](#), [ROW_HEIGHTS](#), [SET_VALUE](#), and [TABLE_BLANK](#) keywords. If [USE_TABLE_SELECT](#) is set, these other keywords only apply to the currently-selected cells. Normally, these keywords apply to the entire contents of a table widget.

Note

If either the title row or the title column of the table is selected, this keyword only modifies the behavior of the [ROW_HEIGHTS](#) or [COLUMN_WIDTHS](#) keywords.

Warning

The Microsoft Windows platform does not support the [ROW_HEIGHTS](#) keyword. Selecting the title row of a table via [USE_TABLE_SELECT](#) has no effect.

Selection Modes

The table widget supports two *selection modes*:

- In *standard selection mode*, only a single selection can be created at a given time. Creating a new selection causes the existing selection to disappear.
- In *disjoint selection mode*, multiple selections can be created at one time. Creating a new selection does not cause the existing selection(s) to disappear. A table can be created in disjoint selection mode using either the [DISJOINT_SELECTION](#) keyword to [WIDGET_TABLE](#) or the [TABLE_DISJOINT_SELECTION](#) keyword to [WIDGET_CONTROL](#).

The keywords listed above to generate different output or expect different input based on the selection mode. See the description of each keyword for details. For more information on table selection modes, see “[Selection Modes](#)” in Chapter 30 of the *Building IDL Applications* manual

Specifying Selections Programmatically

In many cases, your code will use a selection created by the user manually, with the mouse. You can also create selections programmatically, by setting [USE_TABLE_SELECT](#) equal to an array.

- In standard selection mode, specify a four-element array, of the form [*left*, *top*, *right*, *bottom*] specifying the of cells to act upon.

- In disjoint selection mode, specify a $2 \times n$ element array of column/row pairs specifying the cells to act upon.

Specifications for cell locations are zero-based (that is, the first data column is column number zero). The value -1 is used to refer to the title row or title column.

Note

Setting `USE_TABLE_SELECT` equal to an array does *not* change the current table selection. Operations affect the specified cells without changing the current selection.

is equivalent to first calling `WIDGET_CONTROL` and setting the `SET_TABLE_SELECT` keyword equal to an array, and then calling `WIDGET_CONTROL` with `USE_TABLE_SELECT` set equal to one.

Example

To change the column widths of the title column and the first five data columns of a table widget named `wTable`, leaving the widths of the other columns unchanged:

```
WIDGET_CONTROL, wTable, USE_TABLE_SELECT=[-1,0,4,0], $
    COLUMN_WIDTHS=50
```

For additional examples, see “Using Table Widgets” in Chapter 30 of the *Building IDL Applications* manual.

USE_TEXT_SELECT

This keyword applies to widgets created with the `WIDGET_TABLE` and `WIDGET_TEXT` functions.

Set this keyword to modify the behavior of the `GET_VALUE` and `SET_VALUE` keywords. If `USE_TEXT_SELECT` is set, `GET_VALUE` and `SET_VALUE` apply only to the current text selection. Normally, these keywords apply to the entire contents of a text widget.

Warning

If the target widget is a table, the selected cell must be in *edit mode* for `WIDGET_CONTROL` to return the selected text. (A table cell can be selected without being in edit mode; see “Using Table Widgets” in Chapter 30 of the *Building IDL Applications* manual for a discussion of the table widget’s edit mode.) In most cases, to retrieve the value of a table cell, use the `USE_TABLE_SELECT` keyword rather than `USE_TEXT_SELECT`.

X_BITMAP_EXTRA

This keyword applies to widgets created with the `WIDGET_BUTTON` function.

When the value of a button widget is a bitmap, the usual width is taken to be 8 times the number of columns in the source byte array. This keyword can be used to indicate the number of bits in the last byte of each row that should be ignored. The value can range between 0 and 7.

XOFFSET

This keyword applies to all widgets.

Set this keyword to an integer value that specifies the widget's new horizontal offset, in units specified by the `UNITS` keyword (pixels are the default). Attempting to change the offset of a widget that is the child of a `ROW` or `COLUMN` base or a widget that is part of a menubar or pulldown menu causes an error.

XSIZE

This keyword applies to all widgets.

Set this keyword to an integer or floating-point value that represents the widget's new horizontal size.

- Text and List widgets: Size is specified in characters. The `UNITS` keyword is ignored.
- Table widgets: Size is specified in columns. The width of the row labels is automatically added to this value. The `UNITS` keyword is ignored.
- All other widgets: If the `UNITS` keyword is present, size is in the units specified. If the `UNITS` keyword is not present, the size is specified in pixels.

Note that `[XY]SIZE` sets client area and `SCR_[XY]SIZE` sets total area (title bar, borders, menu, client area). For scrollable widgets (e.g., scrolling bases and scrolling draw widgets), this keyword adjusts the *viewport* size. Use the `DRAW_XSIZE` keyword to change the width of the drawing area in scrolling draw widgets. Attempting to resize a widget that is part of a menubar or pulldown menu causes an error.

YOFFSET

This keyword applies to all widgets.

Set this keyword to an integer value that specifies the widget's new vertical offset, in units specified by the `UNITS` keyword (pixels are the default). Attempting to change

the offset of a widget that is the child of a ROW or COLUMN base or a widget that is part of a menubar or pulldown menu causes an error.

YSIZE

This keyword applies to all widgets.

Set this keyword to an integer or floating-point value that represents the widget's new vertical size

- Text and List widgets: Size is specified in lines. The UNITS keyword is ignored.
- Table widgets: Size is specified in rows. The height of the column labels is automatically added to this value. The UNITS keyword is ignored.
- All other widgets: If the UNITS keyword is present, size is in the units specified. If the UNITS keyword is not present, the size is specified in pixels.

Note that [XY]SIZE sets client area and SCR_[XY]SIZE sets total area (title bar, borders, menu, client area). For scrollable widgets (e.g., scrolling bases and scrolling draw and table widgets), this keyword adjusts the *viewport* size. Use the DRAW_YSIZE keyword to change the height of the drawing area in scrolling draw widgets. Attempting to resize a widget that is part of a menubar or pulldown menu causes an error.

Version History

Introduced: Pre 4.0

COMBOBOX_ADDITEM, COMBOBOX_DELETEITEM, COMBOBOX_INDEX, DRAW_KEYBOARD_EVENTS, SET_BUTTON, SET_COMBOBOX_SELECT, SET_TAB_CURRENT, SET_TAB_MULTILINE, SET_TREE_BITMAP, SET_TREE_EXPANDED, SET_TREE_SELECT, SET_TREE_VISIBLE, TABLE_BLANK, TABLE_DISJOINT_SELECTION, TLB_ICONIFY_EVENTS, TLB_MOVE_EVENTS, TLB_SIZE_EVENTS, TOOLTIP keywords added: 5.6

PUSHBUTTON_EVENTS and REFRESH_PROPERTY keywords added: 6.0

EDITABLE, MULTIPLE_PROPERTIES, PROPERTY SHEET_SETSELECTED, and TAB_MODE keywords added: 6.1

See Also

[Chapter 30, “Widget Application Techniques”](#) in the *Building IDL Applications* manual.

WIDGET_DISPLAYCONTEXTMENU

The `WIDGET_DISPLAYCONTEXTMENU` procedure displays a *context-sensitive menu* (also known as a *pop-up* or *shortcut* menu). Context-sensitive menus appear when the user clicks the right mouse button over a widget for which a context menu is defined. Because the widget programmer must explicitly detect the right-mouse button click and call `WIDGET_DISPLAYCONTEXTMENU` to display the context menu, a widget can have any number of associated context menus, which can be displayed under different circumstances.

Context menus are created by placing one or more button widgets on a base widget constructed with the `CONTEXT_MENU` keyword set. The context menu's base widget must have as its parent a base, draw, list, property sheet, table, tree or text widget; in the case of all but draw widgets, the parent widget must be configured to generate context menu events via the `CONTEXT_EVENTS` keyword. Draw widgets that support context menus must have the `BUTTON_EVENTS` keyword set.

The widget application's event handler routine must detect the presence of a context menu event (base, list, and text widgets) or the right mouse-button event (draw and property sheet widgets) and call `WIDGET_DISPLAYCONTEXTMENU` to display the context menu. Events generated when the user clicks on the context menu's buttons are handled in the normal way. If the user clicks outside the context menu, it is dismissed and no event is generated.

Syntax

`WIDGET_DISPLAYCONTEXTMENU, Parent, X, Y, ContextBase_ID`

Arguments

Parent

The widget ID of the parent widget of the context menu widget.

Note

The parent widget must be either a base, list, or text widget with the `CONTEXT_EVENTS` keyword set or a draw widget with the `BUTTON_EVENTS` keyword set. In most cases the parent widget can be identified via the ID field of the `WIDGET_CONTEXT` or `WIDGET_DRAW` event structure.

X

The x location, relative to the parent widget, at which the menu should be displayed. (In most cases this will be the x location of the user's right mouse-button click, as reported in the X field of the WIDGET_CONTEXT or WIDGET_DRAW event structure.)

Y

The y location, relative to the parent widget, at which the menu should be displayed. (In most cases this will be the y location of the user's right mouse-button click, as reported in the Y field of the WIDGET_CONTEXT or WIDGET_DRAW event structure.)

ContextBase_ID

The widget ID of the context menu base widget that contains the context menu to be displayed. Use the CONTEXT_MENU keyword to WIDGET_BASE to create a context menu base.

Note

The context menu base must be a child of the widget specified by the *Parent* argument.

Keywords

None.

Examples

For examples using WIDGET_DISPLAYCONTEXTMENU, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

Version History

Introduced: 5.5

WIDGET_DRAW

The `WIDGET_DRAW` function is used to create draw widgets. Draw widgets are rectangular areas that IDL treats as standard graphics windows. Draw widgets can use either IDL Direct graphics or IDL Object graphics, depending on the value of the `GRAPHICS_LEVEL` keyword. Any graphical output that can be produced by IDL can be directed to a draw widget. Draw widgets can have optional scroll bars to allow viewing a larger graphics area than could otherwise be displayed in the widget's visible area.

The returned value of this function is the widget ID of the newly-created draw widget.

Note

On some systems, when backing store is provided by the window system (`RETAIN=1`), reading data from a window using `TVRD()` may cause unexpected results. For example, data may be improperly read from the window even when the image displayed on screen is correct. Having IDL provide the backing store (`RETAIN=2`) ensures that the window contents will be read properly.

For a more detailed discussion of the draw widget, along with examples, see [“Using Draw Widgets”](#) in Chapter 30 of the *Building IDL Applications* manual.

Syntax

```
Result = WIDGET_DRAW(Parent [, /APP_SCROLL] [, /BUTTON_EVENTS]
  [, /COLOR_MODEL] [, COLORS=integer] [, EVENT_FUNC=string]
  [, EVENT_PRO=string] [, /EXPOSE_EVENTS] [, FRAME=width]
  [, FUNC_GET_VALUE=string] [, GRAPHICS_LEVEL=2]
  [, GROUP_LEADER=widget_id]
  [, IGNORE_ACCELERATORS=value] [, KEYBOARD_EVENTS={1 | 2}]
  [, KILL_NOTIFY=string] [, /MOTION_EVENTS] [, /NO_COPY]
  [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string] [, RENDERER={0
  | 1}] [, RESOURCE_NAME=string] [, RETAIN={0 | 1 | 2}]
  [, SCR_XSIZE=width] [, SCR_YSIZE=height] [, /SCROLL] [, /SENSITIVE]
  [, TOOLTIP=string] [, /TRACKING_EVENTS] [, UNAME=string] [, UNITS={0
  | 1 | 2}] [, UVALUE=value] [, VALUE=value] [, /VIEWPORT_EVENTS]
  [, XOFFSET=value] [, XSIZE=value] [, X_SCROLL_SIZE=width]
  [, YOFFSET=value] [, YSIZE=value] [, Y_SCROLL_SIZE=height])
```

Arguments

Parent

The widget ID of the parent widget of the new draw widget.

Keywords

APP_SCROLL

Set this keyword to create a scrollable draw widget with horizontal and vertical scrollbars that allow the user to view portions of the widget contents that are not currently on the screen.

The drawable area of a draw widget created with the APP_SCROLL keyword is the same size as the viewable area (the *viewport*). This is useful for displaying very large images, because the memory used by a draw widget is directly related to the size of the drawable area. A draw widget created with the APP_SCROLL keyword also has the concept of a *virtual drawable area*, which is the size of the entire image. The fact that only a portion of the entire image is held in memory means that you must capture events generated when the user adjusts the scroll bars and display the correct portion of the image in your event-handling code.

Note

If the image you are displaying is relatively small or memory is not a concern, consider using the SCROLL keyword rather than APP_SCROLL. With SCROLL, the drawable area is the size of the entire image; this allows IDL to display the appropriate portions of the image automatically when the user adjusts the scroll bars. See “[SCROLL](#)” on page 2360 and “[Scrolling Draw Widgets](#)” in Chapter 30 of the *Building IDL Applications* manual for details.

Specify the size of the viewport using the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords, and the size of the virtual drawable area using the XSIZE and YSIZE keywords. If APP_SCROLL is set, the application generates viewport events even if the VIEWPORT_EVENTS keyword is not set.

If the drawable area uses Direct Graphics (that is, if GRAPHICS_LEVEL is *not* set equal to 2), system backing store is used by default and the viewport is automatically restored after it has been obscured. If you prefer to manually restore the viewport in an event-handling routine, set RETAIN=0 and EXPOSE_EVENTS=1; this allows you to redraw the virtual canvas when your application receives expose events.

If the drawable area uses Object Graphics (that is, if the `GRAPHICS_LEVEL` keyword is set equal to 2), the application will *not* automatically restore the viewport, and will generate expose events as if `RETAIN=0` and `EXPOSE_EVENTS=1` had been set.

BUTTON_EVENTS

Set this keyword to make the draw widget generate events when the mouse buttons are pressed or released (and the mouse pointer is in the draw widget). Normally, draw widgets do not generate events.

You can use the event structure generated when the `BUTTON_EVENTS` keyword is set to emulate the functionality of the `CONTEXT_EVENTS` keyword to `WIDGET_BASE`, `WIDGET_LIST`, and `WIDGET_TEXT`. To determine whether the user clicked the right mouse button, use the test:

```
IF (event.release EQ 4)
```

where `event` holds the widget event structure.

For more on detecting and handling context menu events, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

COLOR_MODEL

Set this keyword equal to 1 (one) to cause the draw widget’s associated `IDLgrWindow` object to use indexed color. If the `COLOR_MODEL` keyword is not set, or is set to a value other than one, the draw widget will use RGB color.

This keyword is only valid when the draw widget uses IDL Object Graphics. (The graphics type used by a draw widget is determined by setting the `GRAPHICS_LEVEL` keyword to `WIDGET_DRAW`.)

COLORS

The maximum number of color table indices to be used. This parameter has effect *only* if it is supplied when the *first* IDL graphics window is created.

If `COLORS` is not specified when the first window is created, all or most of the available color indices are allocated, depending upon the window system in use.

To use monochrome windows on a color display, set `COLORS` equal to 2 when creating the first window. One color table is maintained for all IDL windows. A negative value for `COLORS` specifies that all but the given number of colors from the shared color table should be used.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EXPOSE_EVENTS

Set this keyword to make the draw widget generate event when the visibility of the draw widget changes. This may occur when the widget is hidden behind something else on the screen, brought to the foreground, or when the scroll bars are moved. Normally, draw widgets do not generate events.

If this keyword is set, expose events will be generated only when IDL is unable to restore the contents of the window itself. After the initial draw, expose events are not issued when `GRAPHICS_LEVEL=2` and the software renderer is being used (`RENDERER=1`). In such cases, expose events are not issued because IDL can internally refresh the window itself. On platforms for which OpenGL support is not offered, the software renderer is always being used, and therefore, expose events are not issued after the initial draw.

Note

When using hardware rendering, you must explicitly disable backing store (by setting `RETAIN=0`) in order to generate expose events. Additional expose events may be generated if both `EXPOSE_EVENTS` and `RETAIN=1` are turned on.

Warning

Large numbers of events may be generated when `EXPOSE_EVENTS` is specified. You may wish to compress the events (perhaps using a timer) and only act on a subset.

FRAME

The value of this keyword specifies the width of a frame in units specified by the `UNITS` keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a hint to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GRAPHICS_LEVEL

Set this keyword equal to 2 (two) to use IDL Object Graphics in the draw widget. If the `GRAPHICS_LEVEL` keyword is not set, or is set to a value other than two, the draw widget will use IDL Direct Graphics.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

IGNORE_ACCELERATORS

Set this keyword to specify what `WIDGET_BUTTON` accelerators are to be ignored when this draw widget has keyboard focus. Setting `IGNORE_ACCELERATORS` allows a defined list of accelerators to be processed by the draw widget instead of by the conflicting accelerated button. Valid values are:

- A string or string array containing any value that is legal for `WIDGET_BUTTON`'s [ACCELERATOR](#) keyword
- The number 1, indicating all accelerators should be ignored

Ordinarily, accelerators are processed before keyboard events reach the widget that has the keyboard focus. Setting `IGNORE_ACCELERATORS` allows a defined list of accelerators to be processed by the draw widget instead of by associated buttons or menu items. Once the draw widget loses focus, all specified accelerators are automatically re-enabled.

See “[Disabling Button Widget Accelerators](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

KEYBOARD_EVENTS

Set this keyword equal to 1 (one) or 2 to make the draw widget generate an event when it has the keyboard focus and a key is pressed or released. (The method by which a widget receives the keyboard focus is dependent on the window manager in use.) The value of the key pressed is reported in either the `CH` or the `KEY` field of the event structure, depending on the type of key pressed. See “[Widget Events Returned by Draw Widgets](#)” on page 2365 for details.

- If this keyword is set equal to 1, the draw widget will generate an event when a “normal” key is pressed. “Normal” keys include all keys except function keys and the modifier keys: `SHIFT`, `CONTROL`, `CAPS LOCK`, and `ALT`. If a modifier key is pressed at the same time as a normal key, the value of the modifier key is reported in the `MODIFIERS` field of the event structure.
- If this keyword is set equal to 2, the draw widget will generate an event when *either* a normal key or a modifier key is pressed. Values for modifier keys are reported in the `KEY` field of the event structure, and the `MODIFIERS` field contains zero.

Note

Keyboard events are never generated for function keys.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

MOTION_EVENTS

Set this keyword to make the draw widget generate events when the mouse cursor moves across the widget. Normally, draw widgets do not generate events.

Draw widgets that return motion events can generate a large number of events that can result in poor performance on slower machines.

Note that it is possible to generate motion events with coordinates outside the draw widget. If you position the mouse cursor inside the draw widget, press the mouse

button, and drag the cursor out of the draw widget, the X and Y fields of the widget event will specify coordinates outside the draw widget.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the SET_UVALUE and GET_UVALUE keywords to WIDGET_CONTROL, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the NO_COPY keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a UVALUE without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the UVALUE keyword to WIDGET_DRAW or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. On a “get” operation (GET_UVALUE keyword to WIDGET_CONTROL), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the SET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RENDERER

Set this keyword to an integer value indicating which graphics renderer to use when drawing objects within the window. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL’s software implementation

By default, your platform's native OpenGL implementation is used. If your platform does not have a native OpenGL implementation, IDL's software implementation is used regardless of the value of this property. See [“Hardware vs. Software Rendering”](#) in Chapter 34 of the *Using IDL* manual for details. Your choice of renderer may also affect the maximum size of a draw widget. See [“IDLgrWindow”](#) on page 4135 for details.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See [“RESOURCE_NAME”](#) on page 2266 for a complete discussion of this keyword.

RETAIN

Set this keyword to 0, 1, or 2 to specify how backing store should be handled for the draw widget. RETAIN=0 specifies no backing store. RETAIN=1 requests that the server or window system provide backing store. RETAIN=2 specifies that IDL provide backing store directly. See [“Backing Store”](#) on page 2369 for details on the use of RETAIN with Direct Graphics. For more information on the use of RETAIN with Object Graphics, see [“IDLgrWindow::Init”](#) on page 4164.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SCROLL

Set this keyword to create a scrollable draw widget with horizontal and vertical scrollbars that allow the user to view portions of the widget contents that are not currently on the screen.

Specify the size of the viewport using the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords, and the size of the entire drawable area using the XSIZE and YSIZE keywords.

Note

While the viewport of a draw widget created with the `SCROLL` keyword may be smaller than the entire image, the drawable area itself is the same size as the image, and uses memory commensurately. If you want to display a portion of a very large image, consider using the `APP_SCROLL` keyword rather than `SCROLL`. The drawable area of a draw widget created with `APP_SCROLL` is the same size as the viewport. This saves memory when the displayed image is large, but requires extra work on the programmer's part to handle viewport events and display the appropriate portion of the image. See "[APP_SCROLL](#)" on page 2354 and "[Scrolling Draw Widgets](#)" in Chapter 30 of the *Building IDL Applications* manual for details.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If `SENSITIVE` is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the `SENSITIVE` keyword with the [WIDGET_CONTROL](#) procedure.

TOOLTIP

Set this keyword to a string that will be displayed when the cursor hovers over the widget. For UNIX platforms, this string must be non-zero in length.

Note

If your application uses hardware rendering and a `RETAIN` setting of either zero or one, tooltips will cause draw widgets to generate expose events if the tooltip obscures the drawable area. This is true even if the tooltip is associated with another widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for WIDGET_BASE.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the [FIND_BY_UNAME](#) keyword. The UNAME should be unique to the widget hierarchy because the [FIND_BY_UNAME](#) keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget’s initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a draw widget is the IDL window number for use with Direct Graphics routines, such as WSET. For Object Graphics

routines, it is the draw window object reference. This value cannot be set or modified by the user.

To obtain the window number for a newly-created draw widget, use the `GET_VALUE` keyword to `WIDGET_CONTROL` *after* the draw widget has been realized. Draw widgets do not have a window number assigned to them until they are realized. For example, to return the window number of a draw widget in the variable `win_num`, use the command:

```
WIDGET_CONTROL, my_drawwidget, GET_VALUE = win_num
```

where `my_drawwidget` is the widget ID of the desired draw widget.

When using Object Graphics for the widget draw, the following command returns an object reference to the draw window:

```
WIDGET_CONTROL, my_drawwidget, GET_VALUE = oWindow
```

where `oWindow` is a window object.

VIEWPORT_EVENTS

Set this keyword to enable viewport motion events for draw widgets.

XOFFSET

The horizontal offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

XSIZE

The width of the widget in units specified by the `UNITS` keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations. By default, draw widgets are 100 pixels wide by 100 pixels high.

X_SCROLL_SIZE

The `XSIZE` keyword always specifies the width of a widget. When the `SCROLL` keyword is specified, this size is not necessarily the same as the width of the visible

area. The `X_SCROLL_SIZE` keyword allows you to set the width of the scrolling viewport independently of the actual width of the widget.

Use of the `X_SCROLL_SIZE` keyword implies `SCROLL`.

YOFFSET

The vertical offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

YSIZE

The height of the widget in units specified by the `UNITS` keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations. By default, draw widgets are 100 pixels wide by 100 pixels high.

Y_SCROLL_SIZE

The `YSIZE` keyword always specifies the height of a widget. When the `SCROLL` keyword is specified, this size is not necessarily the same as the height of the visible area. The `Y_SCROLL_SIZE` keyword allows you to set the height of the scrolling viewport independently of the actual height of the widget.

Use of the `Y_SCROLL_SIZE` keyword implies `SCROLL`.

Keywords to WIDGET_CONTROL

A number of keywords to the `WIDGET_CONTROL` procedure affect the behavior of draw widgets. In addition to those keywords that affect all widgets, the following are particularly useful: `DRAW_BUTTON_EVENTS`, `DRAW_EXPOSE_EVENTS`, `DRAW_KEYBOARD_EVENTS`, `DRAW_MOTION_EVENTS`, `DRAW_VIEWPORT_EVENTS`, `DRAW_XSIZE`, `DRAW_YSIZE`, `GET_DRAW_VIEW`, `GET_VALUE`, `INPUT_FOCUS`, `SET_DRAW_VIEW`, `TOOLTIP`.

Keywords to WIDGET_INFO

A number of keywords to the [WIDGET_INFO](#) function return information that applies specifically to draw widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [DRAW_BUTTON_EVENTS](#), [DRAW_EXPOSE_EVENTS](#), [DRAW_KEYBOARD_EVENTS](#), [DRAW_MOTION_EVENTS](#), [DRAW_VIEWPORT_EVENTS](#), [TOOLTIP](#).

Widget Events Returned by Draw Widgets

By default, draw widgets do not generate events. If the [BUTTON_EVENTS](#) keyword is set when the widget is created, pressing or releasing any mouse button while the mouse cursor is over the draw widget causes events to be generated. Specifying the [MOTION_EVENTS](#) keyword causes events to be generated *continuously* as the mouse cursor moves across the draw widget. Specifying the [EXPOSE_EVENTS](#) keyword causes events to be generated whenever the visibility of any portion of the draw window (or viewport) changes. Specifying the [KEYBOARD_EVENTS](#) keyword causes events to be generated when the draw widget has keyboard focus and a keyboard key is pressed.

The event structure returned by the [WIDGET_EVENT](#) function is defined by the following statement:

```
{WIDGET_DRAW, ID:0L, TOP:0L, HANDLER:0L, TYPE: 0, X:0L, Y:0L,
  PRESS:0B, RELEASE:0B, CLICKS:0, MODIFIERS:0L, CH:0, KEY:0L }
```

Note

If you defined your own {WIDGET_DRAW} structures prior to the IDL 5.3 release before the structure was defined by an internal call, the MODIFIERS field will break the existing user code.

Note

If you defined your own {WIDGET_DRAW} structures prior to the IDL 5.6 release before the structure was defined by an internal call, the CH and KEY fields will break the existing user code.

ID, TOP, and HANDLER are the three standard fields found in every widget event. TYPE returns a value that describes the type of draw widget interaction that generated an event. The values for TYPE are shown in the table below.

Value	Meaning
0	Button Press
1	Button Release
2	Motion
3	Viewport Moved (Scrollbars)
4	Visibility Changed (Exposed)
5	Key Press (ASCII character value reported in CH field)
6	Key Press (Non-ASCII key value reported in KEY field)

Table 3-122: Values for the TYPE field

The X and Y fields give the device coordinates at which the event occurred, measured from the lower left corner of the drawing area.

For button and “viewport moved” events (that is, when the TYPE field contains 0, 1, or 3), PRESS and RELEASE are bitmasks that represent which of the left, center, or right mouse button was pressed:

Bitmask	Mouse Button
1	Left
2	Middle
4	Right

Table 3-123: Bitmask for the PRESS and RELEASE Fields during Button Events

For motion events, both PRESS and RELEASE are zero. For keyboard events, PRESS contains 1 (one) if the key is down and 0 (zero) if it is up; RELEASE contains 0 (zero) if the key is down and 1 (one) if it is up.

Note

IDL obtains information about which mouse button was pressed from the operating system, not from the mouse hardware itself. This means that if the operating system

or some extension thereof remaps the mouse buttons, IDL may receive information that one button was pressed even if a different physical button was pressed. For example, if a user remaps the mouse buttons to reverse left and right, and then presses the right physical mouse button, the widget event structure will reflect a left mouse button press.

The `CLICKS` field is set to either 1 or 2 if the event is a button press event. If the time interval between two button-press events is less than the time interval for a double-click event for the platform, the `CLICKS` field returns 2. If the time interval between button-press events is greater than the time interval for a double-click event for the platform, the `CLICKS` field returns 1. This means that if you are writing a widget application that requires the user to double-click on a draw widget, you will need to handle two events. The `CLICKS` field will return a 1 on the first click and a 2 on the second click. If the event is not a button press event, the `CLICKS` field contains a 0.

The `MODIFIERS` field is valid for button press, button release, motion, and “normal” keyboard events. It is a bitmask which returns the current state of several keyboard modifier keys at the time the event was generated. If a bit is zero, the key is up. If the bit is set, the key is depressed.

Note

“Normal” keyboard events are generated when the `KEYBOARD_EVENTS` keyword is set equal to one. If `KEYBOARD_EVENTS` is set equal to two, the keypress event for a modifier key contains zero in the `MODIFIERS` field and the key value is reported in the `KEY` field; see below.

The value in the `MODIFIERS` field is generated by OR-ing the following values together if a key is depressed.

Bitmask	Modifier Key
1	Shift
2	Control
4	Caps Lock
8	Alt (See Note following this table.)

Table 3-124: Bitmask for the MODIFIERS Field

Note

Under UNIX, the Alt key is the currently mapped MOD1 key.

Keyboard events are generated with the value of the TYPE field equal to 5 or 6. If the event was generated by an ASCII keyboard character, the TYPE field will be set to 5 and the ASCII value of the key will be returned in the CH field. (Note that ASCII values can be converted to the string representing the character using the IDL STRING routine.) If the event was generated due to a non-ASCII keyboard character, the type of the event will be set to 6 and a numeric value representing the key will be returned in the KEY field. The following table lists the possible values of the KEY field.

Note that for the key values reported in the KEY field for the SHIFT, CONTROL, CAPS LOCK, and ALT keys are not the same as those reported in the MODIFIER field bit mask, since the KEY field is not a bitmask.

Key Field Value	Keyboard Key
1	Shift
2	Control
3	Caps Lock
4	Alt
5	Left
6	Right
7	Up
8	Down
9	Page Up
10	Page Down
11	Home
12	End

Table 3-125: Modifier key values reported in the KEY field.

Note on using CURSOR

Note that the CURSOR procedure is only for use with IDL graphics windows. It should not be used with draw widgets. To obtain the cursor position and button state information from a draw widget, examine the X, Y, PRESS, and RELEASE fields in the structures returned by the draw widget in response to cursor events.

Backing Store

Draw widgets with scroll bars rely on backing store to repaint the visible area of the window as it is moved. Their performance is best on systems that provide backing store. However, if your system does not automatically provide backing store, you can make IDL supply it with the statement:

```
DEVICE, RETAIN=2
```

or by using the RETAIN keyword to WIDGET_DRAW.

Note

If you are using graphics acceleration, you may wish to turn off backing store entirely and enable expose events (via the EXPOSE_EVENTS keyword) and redraw the draw widget's contents manually. However, because the number of events generated may be quite high, you may wish to enable a timer as well and only redraw the draw widget periodically.

Version History

Introduced: Pre 4.0

KEYBOARD_EVENTS, TOOLTIP keywords added: 5.6

IGNORE_ACCELERATORS keyword added: 6.1

See Also

[SLIDE_IMAGE](#), [WINDOW](#)

WIDGET_DROPLIST

The `WIDGET_DROPLIST` function creates “droplist” widgets. A droplist widget displays a text field and an arrow button. Selecting either the text field or the button reveals a list of options from which to choose. When the user selects a new option from the list, the list disappears and the text field displays the currently selected option. This action generates an event containing the index of the selected item, which ranges from 0 to the number of elements in the list minus one.

Note

The `WIDGET_COMBOBOX` function creates a similar widget that optionally allows users to edit the text displayed by the droplist. The combobox widget is intended to replace the droplist widget; RSI recommends that new code use `WIDGET_COMBOBOX` rather than `WIDGET_DROPLIST`.

Syntax

```
Result = WIDGET_DROPLIST( Parent [, /DYNAMIC_RESIZE]
    [, EVENT_FUNC=string] [, EVENT_PRO=string] [, FONT=string]
    [, FRAME=value] [, FUNC_GET_VALUE=string]
    [, GROUP_LEADER=widget_id] [, KILL_NOTIFY=string] [, /NO_COPY]
    [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
    [, RESOURCE_NAME=string] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
    [, /SENSITIVE] [, TAB_MODE=value] [, TITLE=string]
    [, /TRACKING_EVENTS] [, UNAME=string] [, UNITS={0 | 1 | 2}]
    [, UVALUE=value] [, VALUE=value] [, XOFFSET=value] [, XSIZE=value]
    [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created droplist widget.

Arguments

Parent

The widget ID of the parent widget for the new droplist widget.

Keywords

DYNAMIC_RESIZE

Note

This keyword is quietly ignored under UNIX. On UNIX systems, droplists do not change their size after being realized.

Set this keyword to create a widget that resizes itself to fit its new value whenever its value is changed. Note that this keyword does not take effect when used with the `SCR_XSIZE`, `SCR_YSIZE`, `XSIZE`, or `YSIZE` keywords. If one of these keywords is also set, the widget will be sized as specified by the sizing keyword and will never resize itself dynamically.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See [“About Device Fonts”](#) on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if `FONT` is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FRAME

The value of this keyword specifies the width of a frame in units specified by the UNITS keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a hint to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the WIDGET_CONTROL procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the WIDGET_EVENT function is called.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the SET_UVALUE and GET_UVALUE keywords to WIDGET_CONTROL, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the NO_COPY keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it

directly to the destination. This feature can be used by compound widgets to obtain state information from a UVALUE without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the UVALUE keyword to WIDGET_DROPLIST or the SET_UVALUE keyword to WIDGET_CONTROL), the variable passed as value becomes undefined. On a “get” operation (GET_UVALUE keyword to WIDGET_CONTROL), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the SET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the TAB_MODE value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the TAB_MODE has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-126: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

Once a `WIDGET_DROPLIST` widget receives focus through tabbing, additional navigation capabilities depend upon the platform as follows:

- On Windows – droplist widgets can receive and lose focus through tabbing.
- On UNIX — the Motif library controls tabbing functionality.

After creating the widget hierarchy, you can change tabbing support using the `WIDGET_CONTROL` procedure's `TAB_MODE` keyword, or query a widget's support for tabbing using the `WIDGET_INFO` procedure's `TAB_MODE` keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TITLE

Set this keyword to a string to be used as the title for the droplist widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for `WIDGET_BASE`.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget

hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set `UNITS` equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If `UVALUE` is not present, the widget’s initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a droplist widget is a scalar string or array of strings that contains the text of the list items—one list item per array element. List widgets are sized based on the length (in characters) of the longest item specified in the array of values for the `VALUE` keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

XOFFSET

The horizontal offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The desired width of the droplist widget area, in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword does not control the size of the droplist button or of the dropped list. Instead, it controls the size “around” the droplist button and, as such, is not particularly useful.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The desired height of the widget, in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword does not control the size of the droplist button or of the dropped list. Instead, it controls the size “around” the droplist button and, as such, is not particularly useful.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) procedure affect the behavior of droplist widgets. In addition to those keywords that affect all widgets, the following are particularly useful: [DYNAMIC_RESIZE](#), [GET_VALUE](#), [SET_DROPLIST_SELECT](#), [SET_VALUE](#).

Keywords to WIDGET_INFO

A number of keywords to the [WIDGET_INFO](#) function return information that applies specifically to droplist widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [DROPLIST_NUMBER](#), [DROPLIST_SELECT](#), [DYNAMIC_RESIZE](#).

Widget Events Returned by Droplist Widgets

Pressing the mouse button while the mouse cursor is over an element of a droplist widget causes the widget to change the label on the droplist button and to generate an event. The event structure returned by the `WIDGET_EVENT` function is defined by the following statement:

```
{ WIDGET_DROPLIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L }
```

The first three fields are the standard fields found in every widget event. `INDEX` returns the index of the selected item. This can be used to index the array of names originally used to set the widget's value.

Note

Platform-specific UI toolkits behave differently if a droplist widget has only a single element. On some platforms, selecting that element again does not generate an event. Events are always generated if the list contains multiple items.

Version History

Introduced: 4.0

`TAB_MODE` keyword added: 6.1

See Also

[CW_PDMENU](#), [WIDGET_BUTTON](#), [WIDGET_COMBOBOX](#), [WIDGET_LIST](#)

WIDGET_EVENT

The `WIDGET_EVENT` function returns events for the widget hierarchy rooted at `Widget_ID`. Widgets communicate information by generating events. Events are generated when a button is pressed, a slider position is changed, and so forth.

Note

Widget applications should use the `XMANAGER` procedure in preference to calling `WIDGET_EVENT` directly. See “[Widget Event Processing](#)” in Chapter 29 of the *Building IDL Applications* manual.

Event Processing

Events for a given widget are processed in the order that they are generated. The event processing performed by `WIDGET_EVENT` consists of the following steps, applied iteratively:

1. Wait for an event from one of the specified widgets to arrive.
2. Starting with the widget that generated the event, move up the widget hierarchy looking for a widget that has an associated event-handling procedure or function. Event-handling routines are associated with a widget via the `EVENT_PRO` and `EVENT_FUNC` keywords to the widget creation functions or the `WIDGET_CONTROL` procedure.
3. If an event-handling *procedure* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling procedure. When the procedure returns, `WIDGET_EVENT` returns to the first step above and starts searching for events. Hence, event-handling procedures are said to “swallow” events.
4. If an event-handling *function* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling function.

When the function returns, its value is examined. If the value is not a structure, it is discarded and `WIDGET_EVENT` returns to the first step. This behavior allows event-handling functions to selectively act like event-handling procedures and “swallow” events.

If the returned value is a structure, it is checked to ensure that it has the standard first three fields: `ID`, `TOP`, and `HANDLER`. If any of these fields is missing, IDL issues an error. Otherwise, the returned value replaces the event found in the first step and `WIDGET_EVENT` continues moving up the widget

hierarchy looking for another event handler routine, as described in step 2, above.

In situations where an event structure is returned, event functions are said to “rewrite” events. This ability to rewrite events is the basis of *compound widgets*, which combine several widgets to give the appearance of a single, more complicated widget. Compound widgets are an important widget programming concept. For more information, see “[Compound Widgets](#)” in Chapter 29 of the *Building IDL Applications* manual.

5. If an event reaches the top of a widget hierarchy without being swallowed by an event handler, it is returned as the value of `WIDGET_EVENT`.
6. If `WIDGET_EVENT` was called without an argument, and there are no widgets left on the screen that are being managed (as specified via the `MANAGED` keyword to the `WIDGET_CONTROL` procedure) and could generate events, `WIDGET_EVENT` ends the search and returns an *empty event* (a standard widget event structure with the top three fields set to zero).

Note

Do not interrupt the event loop by placing a `STOP` or `EXIT` command in the event-handler or other callback routine. The presence of either command will cause the widget routine to exit with an error.

Syntax

```
Result = WIDGET_EVENT([Widget_ID]) [, BAD_ID=variable] [, /NOWAIT]
        [, /SAVE_HOURLASS]
```

UNIX Keywords: [, /YIELD_TO_TTY]

Return Value

A widget event is returned in a structure. The exact contents of this structure vary depending upon the type of widget involved. The first three elements of this structure, however, are always the same.

```
{ WIDGET, ID:0L, TOP:0L, HANDLER:0L, ... }
```

Any other elements vary from widget type to type. The three fixed elements are:

Value	Meaning
ID	The widget ID of the widget that generated the event.
TOP	The widget ID of the top level base for the widget hierarchy containing ID.
HANDLER	When an event is passed as the argument to an event handling procedure or function, as discussed in the previous section, this field contains the identifier of the widget associated with the handler routine. When an event is returned from WIDGET_EVENT, this value is always zero to indicate that no handler routine was found.

Table 3-127: Common Event Structure Fields

Arguments

Widget_ID

Widget_ID specifies the widget hierarchy for which events are desired. The first available event for the specified widget or any of its children is returned. If this argument is not specified, WIDGET_EVENT processes events for all existing widgets.

Widget_ID can also be an array of widget identifiers, in which case all of the specified widget hierarchies are searched.

Note

Attempting to obtain events for a widget hierarchy which is not producing events will hang IDL, unless the NOWAIT keyword is used.

Keywords

BAD_ID

If one of the values supplied via *Widget_ID* is not a valid widget identifier, this function normally issues an error and causes program execution to stop. However, if BAD_ID is present and specifies a named variable, the invalid ID is stored into the variable, and this routine quietly returns. If no error occurs, a zero is stored.

This keyword can be used to handle the situation in which IDL is waiting within `WIDGET_EVENT` when the user kills the widget hierarchy.

This keyword has meaning only if *Widget_ID* is explicitly specified.

NOWAIT

When no events are currently available for the specified widget hierarchy, `WIDGET_EVENT` normally waits until one is available and then returns it. However, if `NOWAIT` is set and no events are present, it immediately returns. In this case, the ID field of the returned structure will be zero.

SAVE_HOURLASS

Set this keyword to prevent the hourglass cursor from being cleared by `WIDGET_EVENT`. This keyword can be of use if a program has to poll a widget periodically during a long computation.

YIELD_TO_TTY

This keyword is only available on UNIX platforms.

Unless the `NOWAIT` keyword is specified, `WIDGET_EVENT` does not return until the asked for event is available. If the user should enter a line of input from the keyboard, it cannot be processed until `WIDGET_EVENT` returns. If the `YIELD_TO_TTY` keyword is specified and the user enters a line of input, `WIDGET_EVENT` returns immediately. In this case, the ID field of the returned structure will be zero.

Note

This keyword is supported under UNIX only, and there are no plans to extend it to other operating systems. Do not use it if you intend to use non-UNIX systems.

Version History

Introduced: Pre 4.0

See Also

[XMANAGER](#)

WIDGET_INFO

The WIDGET_INFO function is used to obtain information about the widget subsystem and individual widgets. The specific area for which information is desired is selected by setting the appropriate keyword.

Syntax

Result = WIDGET_INFO([*Widget_ID*])

Keywords that apply to all widgets: [, /ACTIVE] [, /CHILD] [, /EVENT_FUNC] [, /EVENT_PRO] [, FIND_BY_UNAME=*string*] [, /FONTNAME] [, /GEOMETRY] [, /KBRD_FOCUS_EVENTS] [, /MANAGED] [, /MAP] [, /NAME] [, /PARENT] [, /REALIZED] [, /SENSITIVE] [, /SIBLING] [, /SYSTEM_COLORS] [, /TAB_MODE] [, /TRACKING_EVENTS] [, /TYPE] [, UNITS={0 | 1 | 2}] [, /UNAME] [, /UPDATE] [, /VALID_ID] [, /VERSION] [, /VISIBLE]

Keywords that apply to widgets created with WIDGET_BASE:
[, /CONTEXT_EVENTS] [, /MODAL] [, /TLB_ICONIFY_EVENTS] [, /TLB_KILL_REQUEST_EVENTS] [, /TLB_MOVE_EVENTS] [, /TLB_SIZE_EVENTS]

Keywords that apply to widgets created with WIDGET_BUTTON:
[, /BUTTON_SET] [, /DYNAMIC_RESIZE] [, /PUSHBUTTON_EVENTS] [, /TOOLTIP]

Keywords that apply to widgets created with WIDGET_COMBOBOX:
[, /COMBOBOX_GETTEXT] [, /COMBOBOX_NUMBER] [, /DYNAMIC_RESIZE]

Keywords that apply to widgets created with WIDGET_DRAW:
[, /DRAW_BUTTON_EVENTS] [, /DRAW_EXPOSE_EVENTS] [, DRAW_KEYBOARD_EVENTS={0 | 1 | 2}] [, /DRAW_MOTION_EVENTS] [, /DRAW_VIEWPORT_EVENTS] [, /TOOLTIP]

Keywords that apply to widgets created with WIDGET_DROPLIST:
[, /DROPLIST_NUMBER] [, /DROPLIST_SELECT] [, /DYNAMIC_RESIZE]

Keywords that apply to widgets created with WIDGET_LABEL:
[, /DYNAMIC_RESIZE]

Keywords that apply to widgets created with WIDGET_LIST:
[, /CONTEXT_EVENTS] [, /LIST_MULTIPLE] [, /LIST_NUMBER] [, /LIST_NUM_VISIBLE] [, /LIST_SELECT] [, /LIST_TOP]

Keywords that apply to widgets created with WIDGET_PROPERTY SHEET:

[, /CONTEXT_EVENTS] [, COMPONENT=*objref*]
 [, /MULTIPLE_PROPERTIES] [, PROPERTY_VALID=*string*]
 [, PROPERTY_VALUE=*string*] [, /PROPERTY SHEET_NSELECTED]
 [, /PROPERTY SHEET_SELECTED]

Keywords that apply to widgets created with WIDGET_SLIDER:

[, /SLIDER_MIN_MAX]

Keywords that apply to widgets created with WIDGET_TAB:

[, /TAB_CURRENT] [, /TAB_MULTILINE] [, /TAB_NUMBER]

Keywords that apply to widgets created with WIDGET_TABLE:

[, /COLUMN_WIDTHS] [, /CONTEXT_EVENTS] [, /ROW_HEIGHTS {not supported in Windows}] [, /TABLE_ALL_EVENTS]
 [, /TABLE_DISJOINT_SELECTION] [, /TABLE_EDITABLE]
 [, /TABLE_EDIT_CELL] [, /TABLE_SELECT] [, /TABLE_VIEW]
 [, /USE_TABLE_SELECT]

Keywords that apply to widgets created with WIDGET_TEXT:

[, /CONTEXT_EVENTS] [, /TEXT_ALL_EVENTS] [, /TEXT_EDITABLE]
 [, /TEXT_NUMBER] [, TEXT_OFFSET_TO_XY=*integer*] [, /TEXT_SELECT]
 [, /TEXT_TOP_LINE] [, TEXT_XY_TO_OFFSET=*[column, line]*]

Keywords that apply to widgets created with WIDGET_TREE:

[, /CONTEXT_EVENTS] [, /TREE_EXPANDED] [, /TREE_ROOT]
 [, /TREE_SELECT]

Return Value

Returns the specified information for the given widget ID. If the SYSTEM_COLORS keyword is specified, a structure is returned. See “[The WIDGET_SYSTEM_COLORS Structure](#)” on page 2403 for details.

Arguments

Widget_ID

Usually this argument should be the widget ID of the widget for which information is desired. If the ACTIVE or VERSION keywords are specified, this argument is not required.

Widget_ID can also be an array of widget identifiers, in which case the result is an array with the same structure in which information on all the specified widgets is returned.

Keywords

Not all keywords to `WIDGET_INFO` apply to all combinations of widgets. In the following list, descriptions of keywords that affect only certain types of widgets include a list of the widgets for which the keyword is useful.

ACTIVE

This keyword applies to all widgets.

Set this keyword to return 1 if there is at least one realized, managed, top-level widget on the screen. Otherwise, 0 is returned.

BUTTON_SET

This keyword applies to widgets created with the `WIDGET_BUTTON` function.

Set this keyword to return the “set” state of a widget button. If the button is currently set, 1 (one) is returned. If the button is currently not set, 0 (zero) is returned. This keyword is intended for use with exclusive, non-exclusive and checked menu buttons.

CHILD

This keyword applies to all widgets.

Set this keyword to return the widget ID of the first child of the widget specified by *Widget_ID*. If the widget has no children, 0 is returned.

COLUMN_WIDTHS

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword to return an array of long integers giving the width of each column in the table. If `USE_TABLE_SELECT` is set equal to one, only the column widths for columns that contain currently-selected cells are returned. If `USE_TABLE_SELECT` is set equal to an array, only the column widths for columns that contain specified cells are returned.

COMBOBOX_GETTEXT

This keyword applies to widgets created with the `WIDGET_COMBOBOX` function.

Set this keyword to return the current text from the text box of the specified combobox widget. Note that when using an editable combobox, the text displayed in the text box may not be an item from the list of values in the combobox list. To obtain

the index of the selected item, inspect the INDEX field of the event structure returned by the combobox widget.

COMBOBOX_NUMBER

This keyword applies to widgets created with the [WIDGET_COMBOBOX](#) function.

Set this keyword to return the number of elements currently contained in the list of the specified combobox widget.

COMPONENT

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function.

Set this keyword to an object reference to indicate which object to query. This is most useful when the property sheet references multiple objects. If this keyword is not specified, the first (possibly only) object is queried.

CONTEXT_EVENTS

This keyword applies to widgets created with the [WIDGET_BASE](#), [WIDGET_LIST](#), [WIDGET_PROPERTY SHEET](#), [WIDGET_TABLE](#), [WIDGET_TEXT](#) or [WIDGET_TREE](#) functions.

Set this keyword to return 1 (one) if the widget specified by *Widget_ID* is configured to generate context events (that is, the widget was created with the CONTEXT_EVENTS keyword). Otherwise, 0 (zero) is returned.

DRAW_BUTTON_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to return 1 if *Widget_ID* is a draw widget with the BUTTON_EVENTS attribute set. Otherwise, 0 is returned.

DRAW_EXPOSE_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to return 1 if *Widget_ID* is a draw widget with the EXPOSE_EVENTS attribute set. Otherwise, 0 is returned.

DRAW_KEYBOARD_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to return an integer specifying the type of keyboard events currently generated by the draw widget specified by *Widget_ID*. Possible values are:

- 0: No keyboard events are generated
- 1: Keyboard events are generated for “normal” keys (all keys except function keys and modifier keys: SHIFT, CONTROL, CAPS LOCK, and ALT).
- 2: Keyboard events are generated for “normal” keys and modifier keys.

Note

Keyboard events are never generated for function keys.

DRAW_MOTION_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to return 1 if *Widget_ID* is a draw widget with the MOTION_EVENTS attribute set. Otherwise, 0 is returned.

DRAW_VIEWPORT_EVENTS

This keyword applies to widgets created with the [WIDGET_DRAW](#) function.

Set this keyword to return 1 if *Widget_ID* is a draw widget with the VIEWPORT_EVENTS attribute set. Otherwise, 0 is returned.

DROPLIST_NUMBER

This keyword applies to widgets created with the [WIDGET_DROPLIST](#) function.

Set this keyword to return the number of elements currently contained in the specified droplist widget.

DROPLIST_SELECT

This keyword applies to widgets created with the [WIDGET_DROPLIST](#) function.

Set this keyword to return the zero-based number of the currently-selected element (i.e., the currently-displayed element) in the specified droplist widget.

DYNAMIC_RESIZE

This keyword applies to widgets created with the [WIDGET_BUTTON](#), [WIDGET_COMBOBOX](#), [WIDGET_DROPLIST](#), and [WIDGET_LABEL](#) functions.

Set this keyword to return a True value (1) if the widget specified by *Widget_ID* is a button, droplist, or label widget that has had its DYNAMIC_RESIZE attribute set. Otherwise, False (0) is returned.

EVENT_FUNC

This keyword applies to all widgets.

Set this keyword to return a string containing the name of the event handler function associated with *Widget_ID*. A null string is returned if no event handler function exists.

EVENT_PRO

This keyword applies to all widgets.

Set this keyword to return a string containing the name of the event handler procedure associated with *Widget_ID*. A null string is returned if no event handler procedure exists.

FIND_BY_UNAME

This keyword applies to all widgets.

Set this keyword to a UNAME value that will be searched for in the widget hierarchy, and if a widget with the given UNAME is in the hierarchy, its ID is returned. The search starts in the hierarchy with the given widget ID and travels down, and this keyword returns the widget ID of the first widget that has the specified UNAME value.

If a widget is not found, 0 is returned.

FONTNAME

This keyword applies to all widgets.

Set this keyword to return a string containing the name of the font being used by the specified widget. The returned name can then be used when creating other widgets or with the SET_FONT keyword to the [DEVICE](#) procedure. If fonts are not supported by the specified widget, an empty string is returned.

Note that you can use this keyword to retrieve the name of the *default* font used by one type of widget and use the same font in other contexts. For example, the following code retrieves the default font used for buttons and writes into the drawable area of a draw widget using the same font:

```
wBase = WIDGET_BASE(/ROW)
```

```

wButton = WIDGET_BUTTON(wBase, VALUE="Button")
wDraw = WIDGET_DRAW(wBase, XSIZE=200, YSIZE=100)
WIDGET_CONTROL, wBase, /REALIZE
strFont = WIDGET_INFO(wButton, /FONTNAME)
DEVICE, SET_FONT=strFont
WIDGET_CONTROL, wDraw, GET_VALUE=iWindow
WSET, iWindow
XYOUTS, 100, 50, 'Same Font as Button', FONT=0, /DEVICE, $
ALIGNMENT=.5

```

This method allows you to synchronize the fonts in several widgets without knowing in advance which font is in use.

The format of the returned font name is platform dependent. See [“About Device Fonts”](#) on page 4424 for additional details.

GEOMETRY

This keyword applies to all widgets.

Note

Some widgets have no geometry values of their own. For example, only the root node of a tree widget hierarchy has associated geometry values. Similarly, buttons created on a pop-up menu have no geometry. Widgets with no intrinsic geometry values will return a `WIDGET_GEOMETRY` structure containing all zeroes.

Set this keyword to return a `WIDGET_GEOMETRY` structure that describes the offset and size information for the widget specified by *Widget_ID*. This structure has the following definition:

```

{ WIDGET_GEOMETRY,
  XOFFSET:0.0,
  YOFFSET:0.0,
  XSIZE:0.0,
  YSIZE:0.0,
  SCR_XSIZE:0.0,
  SCR_YSIZE:0.0,
  DRAW_XSIZE:0.0,
  DRAW_YSIZE:0.0,
  MARGIN:0.0,
  XPAD:0.0,
  YPAD:0.0,
  SPACE:0.0 }

```

With the exception of `MARGIN`, all of the structure's fields correspond to the keywords of the same name to the various widget routines. `MARGIN` is the width of

any frame added to the widget, in units specified by the UNITS keyword (pixels are the default). Therefore, the actual width of any widget is:

$$\text{SCR_XSIZE} + (2 * \text{MARGIN})$$

The actual height of any widget is:

$$\text{SCR_YSIZE} + (2 * \text{MARGIN})$$

Note

Different window managers may use different *window dressing* (borders, margins, scrollbars, *etc.*). As a result, running a given segment of widget code on different platforms may yield different geometry.

Note

In most cases, setting SCR_XSIZE (or SCR_YSIZE) for a base widget should override the value set via the XSIZE (or YSIZE) keyword. For *unrealized* base widgets on Microsoft Windows platforms, however, the returned WIDGET_GEOMETRY structure will report the specified value of XSIZE (or YSIZE) rather than the value supplied via the SCR_XSIZE (or SCR_YSIZE) keyword.

Note

For table widgets, the YSIZE and XSIZE values return the total number of rows and columns, which may differ from the visible number of rows and columns.

KBRD_FOCUS_EVENTS

This keyword applies to all widgets.

Set this keyword to return the keyboard focus events status of the widget specified by *Widget ID*. WIDGET_INFO returns 1 (one) if keyboard focus events are currently enabled for the widget, or 0 (zero) if they are not. Only base, table, and text widgets can generate keyboard focus events.

LIST_MULTIPLE

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword equal to a named variable that will contain a non-zero value if the list widget supports multiple item selections. See the MULTIPLE keyword to WIDGET_LIST for more on multiple item selections.

LIST_NUMBER

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword to return the number of elements currently contained in the specified list widget.

LIST_NUM_VISIBLE

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword to return the number of elements that can be visible in the scrolling viewport of the specified list widget. Note that this value can be larger than the total number of elements actually in the list.

LIST_SELECT

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword to return the index or indices of the currently selected (highlighted) element or elements in the specified list widget. Note that this offset is zero-based. If no element is currently selected, -1 is returned.

LIST_TOP

This keyword applies to widgets created with the [WIDGET_LIST](#) function.

Set this keyword to return the zero-based offset of the topmost element currently visible in the specified list widget.

MANAGED

This keyword applies to all widgets.

Set this keyword to return 1 if the specified widget is managed, or 0 otherwise. If no widget ID is specified in the call to [WIDGET_INFO](#), the return value will be an array containing the widget IDs of all currently-managed widgets.

MAP

This keyword applies to all widgets.

Set this keyword to return True (1) if the widget specified by *Widget_ID* is mapped (visible), or False (0) otherwise. Note that when a base widget is unmapped, all of its children are unmapped. If [WIDGET_INFO](#) reports that a particular widget is unmapped, it may be because a parent in the widget hierarchy has been unmapped.

MODAL

This keyword applies to widgets created with the [WIDGET_BASE](#) function and the [MODAL](#) keyword.

If this keyword is set, [WIDGET_INFO](#) will return True (1) if the base widget specified by *Widget_ID* is a modal base widget, or False (0) otherwise.

MULTIPLE_PROPERTIES

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function.

Set this keyword to return the property sheet's current selection mode. A value of 1 indicates support for multiple property selection. A value of 0 indicates that only single property selection is permitted.

NAME

This keyword applies to all widgets.

Set this keyword to return the widget type name of the widget specified by *Widget_ID*. The returned value will be one of the following strings: "BASE", "BUTTON", "COMBOBOX", "DRAW", "DROPLIST", "LABEL", "LIST", "PROPERTY SHEET", "SLIDER", "TAB", "TABLE", "TEXT", or "TREE". Set the [TYPE](#) keyword to return the widget's type code.

PARENT

This keyword applies to all widgets.

Set this keyword to return the widget ID of the parent of the widget specified by *Widget_ID*. If the widget is a top-level base (i.e., it has no parent), 0 is returned.

PROPERTY_VALID

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function.

Set this keyword to a scalar string or one-element string array that contains a property identifier. If the specified property is included in the property sheet, [WIDGET_INFO](#) will return 1; if the property is not included in the property sheet, [WIDGET_INFO](#) will return 0. Comparisons are case-insensitive.

Note

Property identifiers returned in the IDENTIFIER field of an event structure generated by a property sheet widget can be assumed to be valid.

PROPERTY_VALUE

This keyword applies to widgets created with the [WIDGET_PROPERTYSHEET](#) function.

Set this keyword to a string that contains a property identifier. If the specified property is included in the property sheet, WIDGET_INFO will return the value of the property. If the specified property is not included in the property sheet, WIDGET_INFO will generate an error; use the [PROPERTY_VALID](#) keyword to WIDGET_INFO to check for validity before retrieving the property value. Comparisons are case-insensitive.

If the property sheet contains multiple components, use the [COMPONENT](#) keyword to indicate which component the property value should be retrieved from.

Note on use of this keyword

When a user changes a value in the property sheet widget, the actual value of the underlying property does not change automatically. Use this keyword to retrieve the value specified by the user, then apply the new value to the underlying component. The following code snippet illustrates this process:

```
PRO prop_event, e

; Get the value of the changed property, using the component and
; property identifier information contained in the widget event.
value = WIDGET_INFO( e.id, COMPONENT = e.component, $
    PROPERTY_VALUE = e.identifier )

; Set the component's property's value
e.component->SetPropertyByIdentifier, e.identifier, value

END
```

PROPERTY_SHEET_NSELECTED

This keyword applies to widgets created with the [WIDGET_PROPERTY_SHEET](#) function.

Set this keyword to return the number of currently selected properties. The [COMPONENT](#) keyword is not required.

PROPERTY SHEET_SELECTED

This keyword applies to widgets created with the [WIDGET_PROPERTY SHEET](#) function.

Set this keyword to return the identifiers of the currently selected properties. The return value depends on the number of rows selected as follows:

Selection	Return Value
No Properties Selected	Returns an empty string ('').
Single Property Selected	Returns a string containing the property identifier.
Multiple Properties Selected	Returns an array of strings containing each selected property's identifier.

Table 9: Possible Return Values

The [COMPONENT](#) keyword is not required.

PUSHBUTTON_EVENTS

This keyword applies to widgets created with the [WIDGET_BUTTON](#) function.

Set this keyword to return the pushbutton events status for the widget specified by *Widget_ID*. [WIDGET_INFO](#) returns 1 if pushbutton events are currently enabled for the widget, or 0 otherwise.

REALIZED

This keyword applies to all widgets.

Set this keyword to return 1 if the widget specified by *Widget_ID* has been realized. If the widget has not been realized, 0 is returned.

ROW_HEIGHTS

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Note

This keyword is not supported under Microsoft Windows.

Set this keyword to return an array of long integers giving the height of each row in the table. If [USE_TABLE_SELECT](#) is set equal to one, only the row heights for rows

that contain currently-selected cells are returned. If `USE_TABLE_SELECT` is set equal to an array, only the row heights for rows that contain specified cells are returned.

SENSITIVE

This keyword applies to all widgets.

Set this keyword to return True (1) if the widget specified by *Widget_ID* is sensitive (enabled), or False (0) otherwise. Note that when a base is made insensitive, all its children are made insensitive. If `WIDGET_INFO` reports that a particular widget is insensitive, it may be because a parent in the widget hierarchy has been made insensitive.

SIBLING

This keyword applies to all widgets.

Set this keyword to return the widget ID of the first sibling of the widget specified by *Widget_ID*. If the widget is the last sibling in the chain, 0 is returned.

SLIDER_MIN_MAX

This keyword applies to widgets created with the `WIDGET_SLIDER` function.

Set this keyword to return the current minimum and maximum values of the specified slider as a two-element integer array. Element 0 is the minimum value and element 1 is the maximum value.

SYSTEM_COLORS

This keyword applies to all widgets.

Set this keyword and supply the widget ID of any widget to cause `WIDGET_INFO` to return an IDL structure that contains RGB values used for 25 IDL display elements.

For more detailed information on the `WIDGET_SYSTEM_COLORS` structure fields and their meaning see the “[The WIDGET_SYSTEM_COLORS Structure](#)” on page 2403.

TAB_CURRENT

This keyword applies to widgets created with the `WIDGET_TAB` function.

Set this keyword to return the zero-based index of the current tab in the tab widget.

TAB_MODE

This keyword applies to all widgets.

Set this keyword to return what support the specified widget has for navigating among widgets using the **Tab** key. Possible return values are:

Value	Description
0	Navigation is not supported onto or off of the widget.
1	Navigation is supported onto and off of the widget.
2	Navigation is supported only onto the widget.
3	Navigation is supported only off of the widget.

Table 3-128: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero and any attempt to change it is ignored when running a widget application on the UNIX platform.

TAB_MULTILINE

This keyword applies to widgets created with the `WIDGET_TAB` function.

Set this keyword to return the current setting of the multi-line mode for the tab widget.

TAB_NUMBER

This keyword applies to widgets created with the `WIDGET_TAB` function.

Set this keyword to return the number of tabs contained in the tab widget.

TABLE_ALL_EVENTS

This keyword applies to widgets created with the `WIDGET_TABLE` function.

Set this keyword to return 1 (one) if `Widget_ID` is a table widget with the `ALL_EVENTS` attribute set. Otherwise, 0 (zero) is returned.

TABLE_DISJOINT_SELECTION

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to return 1 (one) if the widget specified by *Widget_ID* has disjoint selection enabled. Otherwise, 0 (zero) is returned.

TABLE_EDITABLE

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to return 1 (one) if *Widget_ID* is a table widget that allows user editing of its contents. Otherwise, 0 (zero) is returned.

TABLE_EDIT_CELL

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to return a two-element integer array containing the row and column coordinates of the currently editable cell. If none of the cells in the table widget is currently editable, the array [-1, -1] is returned.

TABLE_SELECT

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to return the currently-selected (highlighted) cells in the specified table widget.

- In standard selection mode, this keyword returns an array of the form [*left*, *top*, *right*, *bottom*] containing the zero-based indices of the columns and rows that define the selection.
- In disjoint selection mode (enabled by setting the `DISJOINT_SELECTION` keyword to `WIDGET_TABLE`), this keyword returns a 2 x *n* array of column/row pairs containing the zero-based indices the selected cells.

TABLE_VIEW

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to return a two-element array of the form [*left*, *top*] containing the zero-based offsets of the top-left cell currently visible in the specified table widget.

TEXT_ALL_EVENTS

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to return 1 if *Widget_ID* is a text widget with the ALL_EVENTS attribute set. Otherwise, 0 is returned.

TEXT_EDITABLE

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to return 1 if *Widget_ID* is a text widget that allows user editing of its contents. Otherwise, 0 is returned.

TEXT_NUMBER

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to return the number of characters currently contained in the specified text widget, including end-of-line characters.

Note

On Windows platforms, Carriage Return/Line Feed pairs count as a single character.

TEXT_OFFSET_TO_XY

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Use this keyword to translate a text widget character offset into column and line form. The value of this keyword should be set to the character offset (an integer) to be translated. [WIDGET_INFO](#) returns a two-element integer array giving the column (element 0) and line (element 1) corresponding to the offset. If the offset specified is out of range, the array [-1,-1] is returned.

TEXT_SELECT

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to return the starting character offset and length (in characters) of the selected (highlighted) text in the specified text widget. [WIDGET_INFO](#) returns a two-element integer array containing the starting position of the highlighted text as an offset from character zero of the text in the widget (element 0), and length of the current selection (element 1).

TEXT_TOP_LINE

This keyword applies to widgets created with the [WIDGET_TEXT](#) function.

Set this keyword to return the zero-based line number of the line currently at the top of a text widget's display viewport. Note that this value is different from the zero-based character offset of the characters in the line. The character offset can be calculated from the line offset via the `TEXT_XY_TO_OFFSET` keyword.

TEXT_XY_TO_OFFSET

This keyword applies to widgets created with the `WIDGET_TEXT` function.

Use this keyword to translate a text widget position given in line and column form into a character offset. The value of this keyword should be set to a two-element integer array specifying the column (element 0) and line (element 1) position. `WIDGET_INFO` returns the character offset (as a longword integer) corresponding to the position. If the position specified is out of range, -1 is returned.

TLB_ICONIFY_EVENTS

This keyword applies to widgets created with the `WIDGET_BASE` function.

Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return iconify events. Otherwise, 0 is returned.

TLB_KILL_REQUEST_EVENTS

This keyword applies to widgets created with the `WIDGET_BASE` function.

Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return kill request events. Otherwise, 0 is returned.

TLB_MOVE_EVENTS

This keyword applies to widgets created with the `WIDGET_BASE` function.

Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return move events. Otherwise, 0 is returned.

TLB_SIZE_EVENTS

This keyword applies to widgets created with the `WIDGET_BASE` function.

Set this keyword to return 1 if the top-level base widget specified by *Widget_ID* is set to return resize events. Otherwise, 0 is returned.

TOOLTIP

This keyword applies to widgets created with the `WIDGET_BUTTON` and `WIDGET_DRAW` functions.

Set this keyword to have the `WIDGET_INFO` function return the text of the tooltip of the widget. If the widget does not have a tooltip, a null string will be returned.

TRACKING_EVENTS

This keyword applies to all widgets.

Set this keyword to return the tracking events status for the widget specified by *Widget_ID*. `WIDGET_INFO` returns 1 if tracking events are currently enabled for the widget. Otherwise, 0 is returned.

TREE_EXPANDED

This keyword applies to widgets created with the `WIDGET_TREE` function.

Set this keyword to return 1 (one) if the specified tree widget node is a folder that is expanded, or 0 (zero) if the specified node is a folder that is collapsed.

Note

Only tree widget nodes created with the `FOLDER` keyword can be expanded or collapsed. This keyword will always return 0 (zero) if the specified tree widget node is not a folder.

TREE_ROOT

This keyword applies to widgets created with the `WIDGET_TREE` function.

Set this keyword to return the widget ID of the *root node* of the tree widget hierarchy of which *Widget ID* is a part. The root node is the tree widget whose parent is a base widget.

TREE_SELECT

This keyword applies to widgets created with the `WIDGET_TREE` function.

Set this keyword to return information about the nodes selected in the specified tree widget. This keyword has two modes of operation, depending on the widget ID passed to `WIDGET_INFO`:

- If the specified widget ID is for the root node of the tree widget (the tree widget whose *Parent* is a base widget), this keyword returns either the widget ID of the selected node or (if multiple nodes are selected) an array of widget IDs of the selected nodes. If no nodes are selected, -1 is returned.
- If the specified widget ID is a tree widget that is a node in a tree, this keyword returns 1 (one) if the node is selected or 0 (zero) if it is not selected.

TYPE

This keyword applies to all widgets.

Set this keyword to return the type code of the specified *Widget_ID*. Possible values are given in the following table. Note that you can set the NAME keyword to return string names instead.

Value	Type
0	Base
1	Button
2	Slider
3	Text
4	Draw
5	Label
6	List
8	Droplist
9	Table
10	Tab
11	Tree
12	Combobox

Table 3-129: Widget Type Codes

UNAME

This keyword applies to all widgets.

Set this keyword to have the WIDGET_INFO function return the user name of the widget.

UNITS

This keyword applies to all widgets.

Use this keyword to specify the unit of measurement used when returning dimensions for most widget types. Set UNITS equal to 0 (zero) to specify that all measurements

are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

Note

This keyword does not affect all sizing operations. Specifically, the value of UNITS is ignored when retrieving the XSIZE or YSIZE of a [WIDGET_LIST](#), [WIDGET_TABLE](#), or [WIDGET_TEXT](#) functions.

UPDATE

This keyword applies to all widgets.

Set this keyword to return 1 if the widget hierarchy that contains *Widget_ID* is set to display updates. Otherwise, 0 is returned. See “[UPDATE](#)” on page 2346.

USE_TABLE_SELECT

This keyword applies to widgets created with the [WIDGET_TABLE](#) function.

Set this keyword to modify the behavior of the COLUMN_WIDTHS and ROW_HEIGHTS keywords. If USE_TABLE_SELECT is set, the COLUMN_WIDTHS and ROW_HEIGHTS keywords only apply to the currently-selected cells. Normally, these keywords apply to the entire contents of a table widget.

The USE_TABLE_SELECT keyword can also be specified as a four-element array, of the form [*left, top, right, bottom*], giving the group of cells to act on. In this usage, the value -1 is used to refer to the row or column titles.

VALID_ID

This keyword applies to all widgets.

Set this keyword to return 1 if *Widget_ID* represents a currently-valid widget. Otherwise, 0 is returned.

VERSION

This keyword applies to all widgets.

Set this keyword to return a structure that gives information about the widget implementation. This structure has the following definition:

```
{ WIDGET_VERSION, STYLE:'', TOOLKIT:'', RELEASE:'' }
```

STYLE is the style of widget toolkit used. TOOLKIT is the implementation of the toolkit. RELEASE is the version level of the toolkit. This field can be used to

distinguish between different releases of a given toolkit, such as Motif 1.0 and Motif 1.1.

VISIBLE

This keyword applies to all widgets.

Set this keyword to return True (1) if the widget specified by *Widget_ID* is visible, or False (0) otherwise. A widget is visible if:

- it has been realized,
- it and all of its ancestors are mapped.

Note

The value returned by `WIDGET_INFO` when this keyword is set is not affected if the widget is minimized or obscured by another window. Widgets that are visible in the sense of this keyword may not be immediately apparent to a person viewing a particular display device.

The WIDGET_SYSTEM_COLORS Structure

When the `SYSTEM_COLORS` keyword is specified in a call to `WIDGET_INFO`, IDL returns a `WIDGET_SYSTEM_COLORS` structure. This allows application developers to determine what colors are used in IDL application widgets, so they can design widgets for their application with the same look and feel as the supplied IDL widgets.

The `WIDGET_SYSTEM_COLORS` structure consists of 25 fields, each containing a three-element vector corresponding to the Red, Green, and Blue color values used for various widget elements. The vector elements are integers ranging between 0 and 255 if a color value is available (vector elements contain -1 if the color value is unavailable). The field names and meaning on the Windows and UNIX operating systems are shown in the following table.

Field Names	Windows Platform	UNIX Platform
<code>DARK_SHADOW_3D</code>	Dark shadow color for 3D display elements.	N/A

Table 3-130: `WIDGET_SYSTEM_COLORS` Structure Fields

Field Names	Windows Platform	UNIX Platform
FACE_3D	Face color for 3D display elements and dialog boxes.	Base background color for all widgets.
LIGHT_EDGE_3D	Highlight color for 3D edges that face the light source.	Color of top and left edges of 3D widgets.
LIGHT_3D	Light color for 3D display elements.	Color of highlight rectangle around widgets with the keyboard focus.
SHADOW_3D	Color for 3D edges that face away from the light source.	Color of bottom and right edges of 3D widgets.
ACTIVE_BORDER	Active window's border color.	Push button background color when button is armed.
ACTIVE_CAPTION	Active window's caption color.	N/A
APP_WORKSPACE	Background color of MDI applications.	N/A
DESKTOP	Desktop color.	N/A
BUTTON_TEXT	Text color on push buttons.	Widget text color.
CAPTION_TEXT	Color of text in caption, size box, and scroll bar arrow box.	Widget text color.
GRAY_TEXT	Color of disabled text.	N/A
HIGHLIGHT	Color of item(s) selected in a widget.	Toggle button fill color.

Table 3-130: WIDGET_SYSTEM_COLORS Structure Fields (Continued)

Field Names	Windows Platform	UNIX Platform
HIGHLIGHT_TEXT	Color of text of item(s) selected in a widget.	N/A
INACTIVE_BORDER	Inactive window's border color.	N/A
INACTIVE_CAPTION	Inactive window's caption color.	N/A
INACTIVE_CAPTION_TEXT	Inactive window's caption text color.	N/A
TOOLTIP_BK	Background color for tooltip controls.	N/A
TOOLTIP_TEXT	Text color for tooltip controls.	N/A
MENU	Menu background color.	N/A
MENU_TEXT	Menu text color.	N/A
SCROLLBAR	Color of scroll bar "gray" area.	Color of scroll bar "gray" area.
WINDOW_BK	Window background color.	Base background color for all widgets.
WINDOW_FRAME	Window frame color.	Widget border color.
WINDOW_TEXT	Text color in windows.	Widget text color.

Table 3-130: WIDGET_SYSTEM_COLORS Structure Fields (Continued)

Version History

Introduced: Pre 4.0

BUTTON_SET, COMBOBOX_GETTEXT, COMBOBOX_NUMBER, FONTNAME, MAP, SENSITIVE, TAB_CURRENT, TAB_MULTILINE, TAB_NUMBER, TABLE_DISJOINT_SELECTION, TLB_ICONIFY_EVENTS, TLB_MOVE_EVENTS, TLB_SIZE_EVENTS, TOOLTIP, TREE_EXPANDED, TREE_ROOT, TREE_SELECT, and VISIBLE keywords added: 5.6

COMPONENT, PROPERTY_VALID, PROPERTY_VALUE, and
PUSHBUTTON_EVENTS keywords added: 6.0

MULTIPLE_PROPERTIES, PROPERTY_SHEET_NSELECTED,
PROPERTY_SHEET_SELECTED, TAB_MODE keywords added: 6.1

See Also

[Chapter 29, “Creating Widget Applications”](#) in the *Building IDL Applications* manual.

WIDGET_LABEL

The WIDGET_LABEL function is used to create label widgets.

Syntax

```
Result = WIDGET_LABEL( Parent [, /ALIGN_CENTER | , /ALIGN_LEFT | ,
    /ALIGN_RIGHT] [, /DYNAMIC_RESIZE] [, FONT=string] [, FRAME=width]
    [, FUNC_GET_VALUE=string] [, GROUP_LEADER=widget_id]
    [, KILL_NOTIFY=string] [, /NO_COPY] [, NOTIFY_REALIZE=string]
    [, PRO_SET_VALUE=string] [, RESOURCE_NAME=string]
    [, SCR_XSIZE=width] [, SCR_YSIZE=height] [, /SENSITIVE]
    [, /SUNKEN_FRAME] [, /TRACKING_EVENTS] [, UNAME=string]
    [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=value] [, XOFFSET=value]
    [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created label widget.

Arguments

Parent

The widget ID of the parent widget for the new label widget.

Keywords

ALIGN_CENTER

Set this keyword to center justify the label text.

ALIGN_LEFT

Set this keyword to left justify the label text.

ALIGN_RIGHT

Set this keyword to right justify the label text.

DYNAMIC_RESIZE

Set this keyword to create a widget that resizes itself to fit its new value whenever its value is changed. Note that this keyword cannot be used with the SCR_XSIZE, SCR_YSIZE, XSIZE, or YSIZE keywords. If one of these keywords is also set, the widget will be sized as specified by the sizing keyword and will never resize itself dynamically.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See [“About Device Fonts”](#) on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if FONT is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FRAME

The value of this keyword specifies the width of a frame in units specified by the UNITS keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_LABEL` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

SUNKEN_FRAME

Set this keyword to create a three dimensional, bevelled border around the label widget. The resulting frame gives the label a “sunken” appearance, similar to what is often seen in application status bars.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for WIDGET_BASE.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget’s initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a widget label is a string containing the text for the label.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of

use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) procedure affect the behavior of label widgets. In addition to those keywords that affect all widgets, the following are particularly useful: [DYNAMIC_RESIZE](#), [GET_VALUE](#), [SET_VALUE](#).

Keywords to WIDGET_INFO

Some keywords to the [WIDGET_INFO](#) function return information that applies specifically to label widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [DYNAMIC_RESIZE](#).

Widget Events Returned by Label Widgets

Label widgets do not return an event structure.

Version History

Introduced: Pre 4.0

SUNKEN_FRAME keyword added: 5.6

See Also

[CW_FIELD](#), [WIDGET_TEXT](#)

WIDGET_LIST

The `WIDGET_LIST` function is used to create list widgets. A list widget offers the user a list of text elements from which to choose. The user can select an item by pointing at it with the mouse cursor and pressing a button. This action generates an event containing the index of the selected item, which ranges from 0 to the number of elements in the list minus one.

The returned value of this function is the widget ID of the newly-created list widget.

Syntax

```
Result = WIDGET_LIST( Parent [, /CONTEXT_EVENTS]
  [, EVENT_FUNC=string] [, EVENT_PRO=string] [, FONT=string]
  [, FRAME=width] [, FUNC_GET_VALUE=string]
  [, GROUP_LEADER=widget_id] [, KILL_NOTIFY=string] [, /MULTIPLE]
  [, /NO_COPY] [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
  [, RESOURCE_NAME=string] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
  [, /SENSITIVE] [, TAB_MODE=value] [, /TRACKING_EVENTS]
  [, UNAME=string] [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=value]
  [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value]
```

Return Value

The returned value of this function is the widget ID of the newly-created list widget.

Arguments

Parent

The widget ID of the parent widget for the new list widget.

Keywords

CONTEXT_EVENTS

Set this keyword to cause context menu events (or simply context events) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus); pass the context event ID

to the `WIDGET_DISPLAYCONTEXTMENU` procedure within your widget program's event handler to display the context menu.

For more on detecting and handling context menu events, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if `FONT` is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FRAME

The value of this keyword specifies the width of a frame in units specified by the `UNITS` keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

MULTIPLE

Set this keyword to allow the user to select more than one item from the list in a single operation. Multiple selections are handled using the platform’s native mechanism:

Motif

Holding down the Shift key and clicking an item selects the range from the previously selected item to the new item. Holding down the mouse button when selecting items also selects a range. Holding down the Control key and clicking an item toggles that item between the selected and unselected state.

Windows

Holding down the Shift key and clicking an item selects the range from the previously selected item to the new item. Holding down the Control key and clicking an item toggles that item between the selected and unselected state.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, `IDL`

makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_LIST` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `XSIZE` keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the TAB_MODE value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the <code>TAB_MODE</code> Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-131: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero and any attempt to change it is ignored when running a widget application on the UNIX platform.

Once the `WIDGET_LIST` has focus, use the up and down arrow keys to navigate among the individual items.

After creating the widget hierarchy, you can change tabbing support using the [`WIDGET_CONTROL`](#) procedure’s `TAB_MODE` keyword, or query a widget’s support for tabbing using the [`WIDGET_INFO`](#) procedure’s `TAB_MODE` keyword.

See [“Tabbing in Widget Applications”](#) in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see [“`TRACKING_EVENTS`”](#) on page 2273 in the documentation for `WIDGET_BASE`.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

Note

This keyword does not affect all sizing operations. Specifically, the value of UNITS is ignored when setting the XSIZE or YSIZE keywords to `WIDGET_LIST`.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget’s initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a list widget is a scalar string or array of strings that contains the text of the list items—one list item per array element. List widgets are sized based on the length (in characters) of the longest item specified in the array of values for the VALUE keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Note that the value of a list widget can only be set. It cannot be retrieved using `WIDGET_CONTROL`.

XOFFSET

The horizontal offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The desired width of the widget, in characters. Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. Note that the final size of the widget may be adjusted to include space for scrollbars (which are not always visible), so your widget may be slightly larger than specified.

YOFFSET

The vertical offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The desired height of the widget, in number of list items visible. Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. Note that the final size of the widget may be adjusted to include space for scrollbars (which are not always visible), so your widget may be slightly larger than specified.

Keywords to `WIDGET_CONTROL`

A number of keywords to the `WIDGET_CONTROL` procedure affect the behavior of list widgets. In addition to those keywords that affect all widgets, the following are

particularly useful: [CONTEXT_EVENTS](#), [SET_LIST_SELECT](#), [SET_LIST_TOP](#), [SET_VALUE](#).

Keywords to WIDGET_INFO

A number of keywords to the [WIDGET_INFO](#) function return information that applies specifically to list widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [CONTEXT_EVENTS](#), [LIST_MULTIPLE](#), [LIST_NUMBER](#), [LIST_NUM_VISIBLE](#), [LIST_SELECT](#), [LIST_TOP](#).

Widget Events Returned by List Widgets

Pressing the mouse button while the mouse cursor is over an element of a list widget causes the widget to highlight the appearance of that element and to generate an event. The appearance of any previously selected element is restored to normal at the same time. The event structure returned by the [WIDGET_EVENT](#) function is defined by the following statement:

```
{WIDGET_LIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L, CLICKS:0L}
```

The first three fields are the standard fields found in every widget event. [INDEX](#) returns the index of the selected item. This index can be used to subscript the array of names originally used to set the widget's value. The [CLICKS](#) field returns either 1 or 2, depending upon how the list item was selected. If the list item is double-clicked, [CLICKS](#) is set to 2.

Note

If you are writing a widget application that requires the user to double-click on a list widget, you will need to handle two events. The [CLICKS](#) field will return a 1 on the first click and a 2 on the second click.

Context Menu Events

A list widget created with the [CONTEXT_EVENTS](#) keyword set returns the following event structure in response to a right mouse button click:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The [X](#) and [Y](#) fields give the device coordinates at which the event occurred, measured from the upper left corner of the list widget. The [ROW](#) and [COL](#) fields return meaningful information for table widgets and values of zero (0) for other widgets.

Version History

Introduced: Pre 4.0

TAB_MODE keyword added: 6.1

ROW and COL fields of WIDGET_CONTEXT event structure added: 6.1

See Also

[CW_BGROUP](#), [WIDGET_COMBOBOX](#), [WIDGET_DROPLIST](#)

WIDGET_PROPERTYSHEET

The `WIDGET_PROPERTYSHEET` function creates a property sheet widget, which exposes the *properties* of an IDL object subclassed from the `IDLitComponent` class in a graphical interface. The property sheet widget must be a child of a base or tab widget, and it cannot be the parent of any other widget.

The `IDLitComponent` class was designed for use by the IDL iTools system. As a result, all `IDLit*` objects subclass from `IDLitComponent`, so properties of object classes written for the IDL iTools system can be displayed in a property sheet. In addition, all `IDLgr*` objects subclass from `IDLitComponent`, which means that properties of standard IDL graphics objects can be displayed in a property sheet even if the rest of the iTools framework is not in use.

In order to be shown in a property sheet, object properties must be *registered* and *visible*. In addition, in order for property values shown in a property sheet to be editable by the user, the property must be *sensitive*. For information on registering properties, see “[Registering Properties](#)” in Chapter 4 of the *iTool Developer’s Guide* manual. For information on making properties visible and sensitive, see “[Property Attributes](#)” in Chapter 4 of the *iTool Developer’s Guide* manual.

Syntax

```
Result = WIDGET_PROPERTYSHEET(Parent [, /ALIGN_BOTTOM
    [, /ALIGN_CENTER |, /ALIGN_LEFT |, /ALIGN_RIGHT |, /ALIGN_TOP]
    [, /CONTEXT_EVENTS] [, /EDITABLE] [, EVENT_FUNC=string]
    [, EVENT_PRO=string] [, FONT=string] [, FRAME=width]
    [, FUNC_GET_VALUE=string] [, IGNORE_ACCELERATORS=value]
    [, KILL_NOTIFY=string] [, MULTIPLE_PROPERTIES] [, /NO_COPY]
    [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
    [, SCR_XSIZE=width] [, SCR_YSIZE=height] [, /SENSITIVE]
    [, /SUNKEN_FRAME] [, /TRACKING_EVENTS] [, UNAME=string]
    [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=value] [, XOFFSET=value]
    [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value])
```

Return Value

The returned value of this function is the widget ID of the newly-created property sheet widget.

Arguments

Parent

The widget ID of the parent for the new property sheet widget. *Parent* must be a base or tab widget.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a ROW base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a ROW or COLUMN base. In ROW bases, the new widget will be vertically centered. In COLUMN bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a ROW base.

CONTEXT_EVENTS

Set this keyword to cause *context menu events* (or simply *context events*) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus); pass the context event ID to the WIDGET_DISPLAYCONTEXTMENU procedure within your widget program's event handler to display the context menu.

For more on detecting and handling context menu events, see “Context-Sensitive Menus” in Chapter 26 of the *Building IDL Applications* manual.

Note

With regard to /CONTEXT_EVENTS, the Motif and Windows version of the property sheet differ very slightly. In the Motif version, individually desensitized cells cannot generate context events, though their row label can.

EDITABLE

Set this keyword to 0 (zero) to mark a property sheet as read-only. Property sheet widgets are editable by default (EDITABLE=1). In this default mode, property values can be modified. When a property sheet is marked as read-only (EDITABLE=0), you can select properties and use any scrollbars, but cannot change any values. Setting this keyword to 0 is roughly equivalent to desensitizing all properties by setting the SENSITIVE attribute of *every* property on the property sheet equal to 0.

Three things that determine the appearance of a property sheet data cells are, in order of greatest to least precedence:

1. Sensitivity of the entire widget — If SENSITIVE=0 for WIDGET_PROPERTY SHEET then no selection or scrolling is possible.
2. Editability of the entire widget — If EDITABLE=0 for the property sheet (meaning it is marked as read-only), cells can be selected but cannot be changed. If EDITABLE=1 (the default value meaning properties can be selected and modified), then the editability of individual properties is controlled by their individual sensitivity values.
3. Sensitivity of an individual property — If SENSITIVE=0 for an individual property (set using the RegisterProperty or SetPropertyAttribute methods of IDLitComponent), then the individual property cannot be selected or changed.

Note

To set the SENSITIVE attribute of individual properties within a property sheet, use [IDLitComponent::RegisterProperty](#) or [IDLitComponent::SetPropertyAttribute](#).

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a device font (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “About Device Fonts” in Appendix I of the *IDL Reference Guide* manual for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if `FONT` is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FRAME

The value of this keyword specifies the width of a frame in units specified by the `UNITS` keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances. If `SUNKEN_FRAME` is also set, `FRAME` is ignored.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

IGNORE_ACCELERATORS

Set this keyword to specify what `WIDGET_BUTTON` accelerators are to be ignored when this property sheet widget has keyboard focus. Valid values are:

- A string or string array containing any value that is legal for `WIDGET_BUTTON`'s `ACCELERATOR` keyword
- The number 1, indicating all accelerators should be ignored

Ordinarily, accelerators are processed before keyboard events reach the widget that has the keyboard focus. Consider a case where the accelerator “Ctrl+V” has been mapped to a button that allows the user to paste objects copied from one drawing area into another. In an application that also has a property sheet, attempting to use **Ctrl+V** to paste text into the property sheet would fail as this event would be stolen by the accelerator. Setting `IGNORE_ACCELERATORS` to “Ctrl+V” allows a property sheet widget with an editable text area to receive keyboard events instead of the button with a conflicting accelerator. Once the property sheet text field loses focus, all specified accelerators are automatically re-enabled.

See “[Disabling Button Widget Accelerators](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

Note

A procedure specified via the `CLEANUP` keyword to `XMANAGER` will override a procedure specified for your application’s top-level base with `WIDGET_BASE`, `KILL_NOTIFY`.

MULTIPLE_PROPERTIES

Set this property to 1 to allow multiple properties to be selected at the same time. When this keyword is set, you can hold down the **Ctrl** key while left-clicking with the mouse to make nonadjacent selections, or hold down the **Shift** key to select an adjacent range of properties in a property sheet. Set this property to 0 (zero) to disallow multiple property selection. This is the default.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, `IDL`

makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. During a set operation (using the `UVALUE` keyword to `WIDGET_BASE` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. During a get operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such callback procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently

SCR_XSIZE

Set this keyword to the desired screen width of the widget, in units specified by the `UNITS` keyword (pixels are the default). If neither `XSIZE` nor `SCR_XSIZE` is specified, then the property sheet widget will be naturally sized based on the width of cell contents. See [“Property Sheet Sizing”](#) on page 922 of *Building IDL Applications* for details.

SCR_YSIZE

Set this keyword to the desired screen height of the widget, in units specified by the `UNITS` keyword (pixels are the default). If neither `YSIZE` nor `SCR_YSIZE` is specified, the property sheet will use a default height. This default is based on the number of rows: 10, or the number of visible properties, whichever is less.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If `SENSITIVE` is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget.

Note

Some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the `SENSITIVE` keyword with the `WIDGET_CONTROL` procedure.

SUNKEN_FRAME

Set this keyword to create the appearance of a three-dimensional, beveled border around the property sheet widget. This makes the widget appear slightly inset. If the `FRAME` keyword is also set, `SUNKEN_FRAME` overrides the setting, and the frame's width will be the minimum necessary to give the appearance of indentation.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “`TRACKING_EVENTS`” in the *IDL Reference Guide* manual in the documentation for `WIDGET_BASE`.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The user value to be assigned to the widget. Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

The user value for a widget can be accessed and modified at any time by using the GET_UVALUE and SET_UVALUE keywords to the WIDGET_CONTROL procedure.

VALUE

Set this keyword to the object reference or array of object references to objects that subclass from the IDLitComponent class. Registered properties of the specified objects will be displayed in the property sheet.

If a single object reference is supplied, the property sheet will have a single column containing the object's properties. If an array of object references is supplied, the property sheet will have multiple columns.

Note

Due to limitations of the user interface controls that underlie the property sheet widget, a property sheet can display properties for at most 100 component objects.

Note

All object references must be to objects of the same type.

If no object references are supplied, the property sheet will initially be empty. Object references can be loaded into an existing property sheet using the SET_VALUE keyword to WIDGET_CONTROL.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a bulletin board base widget. Note that it is best to avoid using this style of widget layout.

XSIZE

The desired width, measured in average character widths of the widget's font, not including a possible vertical scrollbar and any frame thickness. If neither XSIZE nor SCR_XSIZE is specified, then the property sheet widget will be naturally sized based on the width of cell contents. See “[Property Sheet Sizing](#)” on page 922 of *Building IDL Applications* for details.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a bulletin board base widget. Note that it is best to avoid using this style of widget layout.

YSIZE

The desired height of the widget, in number of visible properties. The ultimate height of the property sheet in pixels will include the heights of the column header, the possible horizontal scrollbar, and any frame. If neither YSIZE nor SCR_YSIZE is specified, the property sheet will use a default height. This default is based on the number of rows: 10, or the number of visible properties, whichever is less.

Keywords to WIDGET_CONTROL

A number of keywords to the WIDGET_CONTROL affect the behavior of property sheet widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: [CONTEXT_EVENTS](#), [EDITABLE](#), [MULTIPLE_PROPERTIES](#), [PROPERTY SHEET_SELECTED](#), [REFRESH_PROPERTY](#).

Keywords to WIDGET_INFO

Some keywords to WIDGET_INFO return information that applies specifically to property sheet widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: [COMPONENT](#), [CONTEXT_EVENTS](#), [MULTIPLE_PROPERTIES](#), [PROPERTY_VALID](#), [PROPERTY_VALUE](#), [PROPERTY_SHEET_NSELECTED](#), [PROPERTY_SHEET_SELECTED](#).

Widget Events Returned by Property Sheet Widgets

Several variations of the property sheet widget event structure depend upon the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER). The select and change events also include an integer TYPE field that indicates which type of structure has been returned or which type of event was generated. Programs should always check the TYPE field before referencing fields that are not present in all property sheet event structures, unless the CONTEXT_EVENTS keyword is set. When handling context menu events, the event structure name must be used instead of the TYPE field. The different property sheet widget event structures are described below.

Change Event (TYPE=0)

This event is generated whenever the user enters a new value for a property. It is also used to signal that a user-defined property needs to be changed. The following statement defines the event structure returned by the WIDGET_EVENT function:

```
{WIDGET_PROPSHEET_CHANGE, ID:0L, TOP:0L, HANDLER:0L, TYPE:0L,
  COMPONENT:OBJREF, IDENTIFIER:"", PROPTYPE:0L, SET_DEFINED: 0L}
```

The COMPONENT field contains an object reference to the object associated with the property sheet. When multiple objects are associated with the property sheet, this field indicates which object is to change.

The IDENTIFIER field specifies the value of the property's identifier attribute. This identifier is unique among all of the component's properties.

The PROPTYPE field indicates the type of the property (integer, string, etc.). The integer values for these types are:

- 0 = USERDEF
- 1 = BOOLEAN
- 2 = INTEGER
- 3 = FLOAT

- 4 = STRING
- 5 = COLOR
- 6 = LINESTYLE
- 7 = SYMBOL
- 8 = THICKNESS
- 9 = ENUMLIST

The `SET_DEFINED` field indicates whether or not an undefined property is having its value set. In most circumstances, along with its new value, the property should have its `UNDEFINED` attribute set to zero. If a property is never marked as undefined, this field can be ignored.

Select Event (`TYPE=1`)

The select event is generated whenever the current row or column in the property sheet changes. Navigation between cells is performed by clicking on a cell. When the property sheet is realized, no properties are selected by default. However a single property or multiple properties can be selected programmatically using the `PROPERTY SHEET_SETSELECTED` keyword to the `WIDGET_CONTROL` procedure.

The following statement defines the event structure returned by the `WIDGET_EVENT` function:

```
{WIDGET_PROPSHEET_SELECT, ID:0L, TOP:0L, HANDLER:0L, TYPE:0L,
  COMPONENT:OBJREF, IDENTIFIER:"", NSELECTED:0L }
```

The `COMPONENT` tag is a reference to the object associated with the selected property value. When multiple objects (also known as components) are associated with the property sheet, this member indicates which one object had one of its property values selected. If a property (instead of a property value) is selected, the `COMPONENT` tag always contains an object reference to the first object, even if there are multiple objects in the property sheet.

The `IDENTIFIER` field specifies the value of the property's identifier attribute. This identifier is unique among all properties of the component. When multiple selections are made, this field holds the identifier of the first item selected. This is not the first item selected with the mouse, but the first item encountered in the property sheet definition among those which are selected. To return the identifiers of all the selected properties, use `WIDGET_INFO` with the `/PROPERTY SHEET_SELECTED` keyword.

The `NSELECTED` field specifies the total number of currently selected properties. This is equivalent to calling `WIDGET_INFO` with `/PROPERTY SHEET_NSELECTED`.

Context Menu Events

A property sheet widget created with the `CONTEXT_EVENTS` keyword set returns the following event structure in response to a right mouse button click:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The `X` and `Y` fields give the device coordinates at which the event occurred, measured from the upper left corner of the property sheet widget. The `ROW` and `COL` fields return meaningful information for table widgets and values of zero (0) for other widgets.

Note

When working with property sheet context menu events, it is important to notice that the event structure does not have a `TYPE` field, so special code is needed for the property sheet event handler. Instead of keying off the `TYPE` field, use the structure's name. See [“Property Sheet Context Menu Example”](#) on page 2438 for sample code.

Example

Single Property Sheet Example

Enter the following program in the IDL Editor:

```
; ExSinglePropSheet
;
; Creates a base with a property sheet. Only the
; default properties are visible. The property sheet's
; event handler sets values and reveals selection
; changes.

; Handler for selection or value modification events.
PRO PropertyEvent, event

IF (event.type EQ 0) THEN BEGIN ; Value changed.

    ; Get the value of the property identified by
    ; event.identifier.
    value = WIDGET_INFO(event.ID, COMPONENT = event.component, $
        PROPERTY_VALUE = event.identifier)
```

```

; Set the component's property value.
event.component->SetPropertyByIdentifier, event.identifier, $
    value

PRINT, 'Changed: ', event.identifier, ': ', value

ENDIF ELSE BEGIN ; Selection changed.

; Return information about single or multiple property
; selections.
vNumSelected = WIDGET_INFO(event.ID, /PROPERTY SHEET_NSELECTED)
vSelected = WIDGET_INFO(event.ID, /PROPERTY SHEET_SELECTED)
PRINT, 'Number properties selected: ' + STRING(vNumSelected)
PRINT, 'Selected properties: '
PRINT, vSelected

ENDELSE

END

; Handler for property sheet resize event.
PRO ExSinglePropSheet_event, event

prop = WIDGET_INFO(event.top, $
    FIND_BY_UNAME = 'PropSheet')
WIDGET_CONTROL, prop, SCR_XSIZE = event.x, $
    SCR_YSIZE = event.y

END

; Handler for top-level base kill event.
PRO CleanupEvent, baseID

WIDGET_CONTROL, baseID, GET_UVALUE = oComp
OBJ_DESTROY, oComp

END

PRO ExSinglePropSheet

; Create and initialize the component.
oComp = OBJ_NEW('IDLitVisAxis')

; Create a base and property sheet.
base = WIDGET_BASE(/TLB_SIZE_EVENT, $
    TITLE = 'Single Property Sheet Example', $
    KILL_NOTIFY = 'CleanupEvent')
prop = WIDGET_PROPERTY SHEET(base, VALUE = oComp, $

```

```

EVENT_PRO = 'PropertyEvent', UNAME = 'PropSheet', $
YSIZE=20, /MULTIPLE_PROPERTIES)

; Activate the widgets.
WIDGET_CONTROL, base, SET_UVALUE = oComp, /REALIZE

XMANAGER, 'ExSinglePropSheet', base, /NO_BLOCK

END

```

Save this program as `exsinglepropsheet.pro`, then compile and run the program. A property sheet entitled Single Property Sheet Example is displayed:

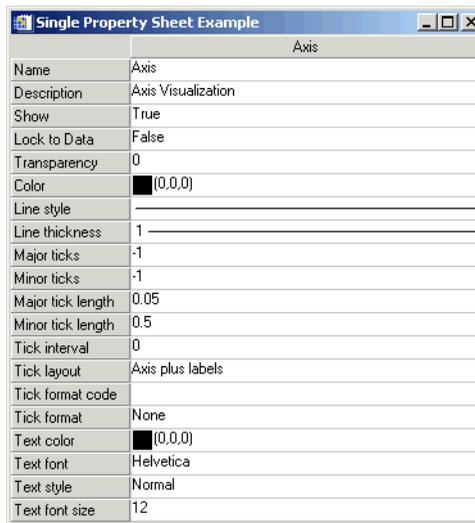


Figure 3-66: Single Property Sheet Example

For examples of the types of settings possible from the property sheet:

- Click the **Show** setting box, click the drop-down button, and select `False` from the list.
- Click the **Major tick length** setting box, click its drop-down button, and move the slider to select a new value.
- Click the **Text color** setting box, click the drop-down button, and select a new color from the color selector.

Property Sheet Context Menu Example

The following simple example shows how to key event handling off of the event structure name instead of the event type. Attempting to use the `CONTEXT_EVENTS` keyword without this type of code will cause an error, because, unlike the `WIDGET_PROPSHEET_CHANGE` and `WIDGET_PROPSHEET_SELECT` event structures, the `WIDGET_CONTEXT` structure does not have a `TYPE` field.

```

; propsheet_context.pro
; For accessing property sheet widget context menus.

PRO prop_event, event

    ; Initiate event based on event structure name, not
    ; event type.
    SWITCH TAG_NAMES( EVENT, /STRUCTURE_NAME ) OF

        'WIDGET_PROPSHEET_CHANGE': $
            ; Placeholder for desired code.
            BEGIN
            BREAK
            END

        'WIDGET_PROPSHEET_SELECT': $
            ; Placeholder for desired code.
            BEGIN
            BREAK
            END

        'WIDGET_CONTEXT': $
            BEGIN
                contextBase = WIDGET_INFO( event.id, $
                    FIND_BY_UNAME = 'contextMenu' )

                ; Display the context menu at the x, y location
                ; of the right mouse click.
                WIDGET_DISPLAYCONTEXTMENU, $
                    event.id, event.x, event.y, contextBase
            BREAK
            END
    ENDSWITCH

END

PRO prop_cleanup, propSheet

    ; Clean up object references.
    WIDGET_CONTROL, propSheet, GET_UVALUE = oAxis
    OBJ_DESTROY, oAxis

```

```

END

PRO yes_event, event

    print, 'Yes'

END

PRO no_event, event

    print, 'No'

END

PRO propsheet_context

    tlb = WIDGET_BASE( /ROW, XOFFSET = 100, YOFFSET = 100 )

    ; Create a property sheet with axis properties.
    oAxis = OBJ_NEW( 'IDLitVisAxis' )

    propSheet = WIDGET_PROPERTY SHEET( tlb, $
        VALUE=oAxis, UVALUE=oAxis, KILL_NOTIFY='prop_cleanup', $
        /CONTEXT_EVENTS, EVENT_PRO = 'prop_event' )

    ; Create a base for the context menu.
    contextBase = WIDGET_BASE( propSheet, /CONTEXT_MENU, $
        UNAME = 'contextMenu' )

    ; Add two buttons to the context menu's base.
    cb1 = WIDGET_BUTTON( contextBase, VALUE = 'Yes', $
        EVENT_PRO = 'yes_event' )
    cb2 = WIDGET_BUTTON( contextBase, VALUE = 'No', $
        EVENT_PRO = 'no_event' )

    ; Activate the widgets.
    WIDGET_CONTROL, tlb, /REALIZE
    XMANAGER, 'propsheet_context', tlb, /NO_BLOCK

END

```

Save and run the example. Left-clicking selects a property. Right-clicking brings up the context menu. Selecting one of the context menu items prints a related string to the Output Log window.

Version History

Introduced: 6.0

EDITABLE, FRAME, IGNORE_ACCELERATORS, MULTIPLE_PROPERTIES,
and SUNKEN_FRAME keywords added: 6.1

NSELECTED field of WIDGET_PROPSHEET_SELECT event structure added: 6.1

ROW and COL fields of WIDGET_CONTEXT event structure added: 6.1

WIDGET_SLIDER

The `WIDGET_SLIDER` function is used to create slider widgets. Slider widgets are used to indicate an integer value from a range of possible values. They consist of a rectangular region which represents the possible range of values. Inside this region is a sliding pointer that displays the current value. This pointer can be manipulated by the user via the mouse, or from within IDL via the `WIDGET_CONTROL` procedure.

To indicated floating-point values, see [CW_FSLIDER](#).

Syntax

```
Result = WIDGET_SLIDER( Parent [, /DRAG] [, EVENT_FUNC=string]
    [, EVENT_PRO=string] [, FONT=string] [, FRAME=width]
    [, FUNC_GET_VALUE=string] [, GROUP_LEADER=widget_id]
    [, KILL_NOTIFY=string] [, MAXIMUM=value] [, MINIMUM=value]
    [, /NO_COPY] [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
    [, RESOURCE_NAME=string] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
    [, SCROLL=units] [, /SENSITIVE] [, /SUPPRESS_VALUE]
    [, TAB_MODE=value] [, TITLE=string] [, /TRACKING_EVENTS]
    [, UNAME=string] [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=value]
    [, /VERTICAL] [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value]
    [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created slider widget.

Arguments

Parent

The widget ID of the parent for the new slider widget.

Keywords

DRAG

Set this keyword to cause events to be generated continuously while the slider is being dragged by the user. Normally, slider widgets generate position events only when the slider comes to rest at its final position and the mouse button is released.

Note

Under Microsoft Windows, this keyword has no effect. IDL sliders only generate events when the mouse button is released.

When a slider widget is set to return drag events, a large number of events can be generated. On slower machines, poor performance can result. Therefore, this option should only be used when detailed or truly interactive control is required.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See [“About Device Fonts”](#) on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if `FONT` is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts.

FRAME

The value of this keyword specifies the width of a frame in units specified by the `UNITS` keyword (pixels are the default) to be drawn around the borders of the widget.

Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

MAXIMUM

The maximum value of the range encompassed by the slider. If this keyword is not supplied, a default of 100 is used.

MINIMUM

The minimum value of the range encompassed by the slider. If this keyword is not supplied, a default of 0 is used.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, `IDL`

makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_SLIDER` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `XSIZE` keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SCROLL

Set the SCROLL keyword to an integer value specifying the number of integer units the scroll bar should move when the user clicks the left mouse button inside the slider area (Motif) or on the slider arrows (Windows), but not on the slider itself. The default on both platforms is $0.1 \times (\text{MAXIMUM} - \text{MINIMUM})$, which is 10% of the slider range.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

SUPPRESS_VALUE

Set this keyword to inhibit the display of the current slider value.

Sliders work only with integer units. This keyword can be used to suppress the actual value of a slider so that a program can present the user with a slider that seems to work in other units (such as floating-point) or with a non-linear scale.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for WIDGET_BASE.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is inherited by lower-level bases and child widgets from the parent `WIDGET_BASE` unless it is explicitly set on an individual widget. If the `TAB_MODE` value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-132: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero and any attempt to change it is ignored when running a widget application on the UNIX platform.

Once the `WIDGET_SLIDER` has focus, use the **Home** and **End** keys, and right and left arrow keys to change the marker value on Windows. The **Page Up** and **Page Down** keys can also be used on Windows to increase or decrease the slider marker value by multiple units at a time. On UNIX, the Motif library controls tabbing functionality.

After creating the widget hierarchy, you can change tabbing support using the [WIDGET_CONTROL](#) procedure's `TAB_MODE` keyword, or query a widget's support for tabbing using the [WIDGET_INFO](#) procedure's `TAB_MODE` keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TITLE

A string containing the title to be used for the slider widget.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set `UNITS` equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If `UVALUE` is not present, the widget's initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a widget slider is the current position of the slider.

VERTICAL

Set this keyword to create a vertical slider. If this keyword is omitted, the slider is horizontal.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

Note

Setting XSIZE for a vertical slider (created with the VERTICAL keyword) does not change the visible width of the slider itself, but does change the amount of horizontal space occupied by the widget within its parent base.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

Note

Setting YSIZE for a horizontal slider does not change the visible height of the slider itself, but does change the amount of vertical space occupied by the widget within its parent base.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) procedure affect the behavior of slider widgets. In addition to those keywords that affect all widgets, the following are particularly useful: [GET_VALUE](#), [SET_SLIDER_MAX](#), [SET_SLIDER_MIN](#), [SET_VALUE](#).

Keywords to WIDGET_INFO

Some keywords to the [WIDGET_INFO](#) function return information that applies specifically to slider widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [SLIDER_MIN_MAX](#).

Widget Events Returned by Slider Widgets

Slider widgets generate events when the mouse is used to change their value. The event structure returned by the [WIDGET_EVENT](#) function is defined by the following statement:

```
{WIDGET_SLIDER, ID:0L, TOP:0L, HANDLER:0L, VALUE:0L, DRAG:0}
```

ID is the widget ID of the button generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. VALUE returns the new value of the slider. DRAG returns integer 1 if the slider event was generated as part of a drag operation, or zero if the event was generated when the user had finished positioning the slider. Note that the slider widget only generates events during the drag operation if the DRAG keyword is set. When the DRAG keyword is set, the DRAG field can be used to avoid computationally expensive operations until the user releases the slider.

Known Implementation Problems

Under Motif 1.0, vertical sliders with a title organized in row bases get horizontally truncated and the slider doesn't show (the title does). Use the `XSIZE` keyword to work around this.

Version History

Introduced: Pre 4.0

`TAB_MODE` keyword added: 6.1

See Also

[CW_FSLIDER](#)

WIDGET_TAB

The WIDGET_TAB function is used to create a tab widget. Tab widgets present a display area on which different pages (base widgets and their children) can be displayed by selecting the appropriate tab. The titles of the tabs are supplied as the values of the TITLE keyword for each of the tab widget's child base widgets.

For a more detailed discussion of the tab widget, along with examples, see “Using Tab Widgets” in Chapter 30 of the *Building IDL Applications* manual.

Syntax

```
Result = WIDGET_TAB( Parent [, /ALIGN_BOTTOM | , /ALIGN_CENTER | ,
  /ALIGN_LEFT | , /ALIGN_RIGHT | , /ALIGN_TOP] [, EVENT_FUNC=string]
  [, EVENT_PRO=string] [, FUNC_GET_VALUE=string]
  [, GROUP_LEADER=widget_id] [, KILL_NOTIFY=string]
  [, LOCATION={0 | 1 | 2 | 3}] [, MULTILINE=0 | 1 (Windows) or num tabs per
  row (Motif)] [, /NO_COPY] [, NOTIFY_REALIZE=string]
  [, PRO_SET_VALUE=string] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
  [, /SENSITIVE] [, TAB_MODE=value] [, /TRACKING_EVENTS]
  [, UNAME=string] [, UNITS={0 | 1 | 2}] [, UVALUE=value]
  [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created tab widget.

Arguments

Parent

The widget ID of the parent for the new tab widget.

Note

Only base widgets can be the parent of a tab widget.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a ROW base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a ROW or COLUMN base. In ROW bases, the new widget will be vertically centered. In COLUMN bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a ROW base.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this

technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as group leader for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. You cannot remove a widget from an existing group.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such callback procedure. This callback procedure can be removed by setting the routine name to the null string (' '). Note that the procedure specified is used only if you are not using the `XMANAGER` procedure to manage your widgets.

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

If you use the `XMANAGER` procedure to manage your widgets, the value of this keyword is overwritten. Use the `CLEANUP` keyword to `XMANAGER` to specify a procedure to be called when a managed widget dies.

LOCATION

Set this keyword equal to an integer that specifies which edge of the tab widget will contain the tabs. The possible values are:

Value	Description
0	The tabs are placed along the top of the widget, which is the default behavior.
1	The tabs are placed along the bottom of the widget.

Table 3-133: LOCATION Keyword Values

Value	Description
2	The tabs are placed along the left edge of the widget. The text label for each tab is displayed vertically. On Windows platforms, setting the keyword to this value implies the MULTILINE keyword.
3	The tabs are placed along the right edge of the widget. The text label for each tab is displayed vertically. On Windows platforms, setting the keyword to this value implies the MULTILINE keyword.

Table 3-133: LOCATION Keyword Values (Continued)

MULTILINE

This keyword controls how tabs appear on the tab widget when all of the tabs do not fit on the widget in a single row. This keyword behaves differently on Windows and Motif systems.

Windows

Set this keyword to cause tabs to be organized in a multiline display when the width of the tabs exceeds the width of the largest child base widget. If possible, IDL will create tabs that display the full tab text.

If MULTILINE = 0 and LOCATION = 0 or 1, tabs that exceed the width of the largest child base widget are shown with scroll buttons, allowing the user to scroll through the tabs while the base widget stays immobile.

If LOCATION = 1 or 2, a multiline display is always used if the tabs exceed the height of the largest child base widget.

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. The text of the tabs (the titles of the tab widget's child base widgets) may be truncated even if the MULTILINE keyword is set.

Motif

Set this keyword equal to an integer that specifies the maximum number of tabs to display per row in the tab widget. If this keyword is not specified (or is explicitly set equal to zero) all tabs are placed in a single row.

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. The text of the tabs (the titles of the tab widget's child base widgets) is never truncated in order to make the tabs fit the space available. However, tab text may be truncated if the text of a single tab exceeds the space available. If `MULTILINE` is set to any value other than one, some tabs may not be displayed.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copying the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. Upon a set operation (using the `UVALUE` keyword to `WIDGET_TAB` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. Upon a get operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string containing the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single callback procedure. This callback procedure can be removed by setting the routine name to the null string (`' '`). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

SCR_XSIZE

Set this keyword to the desired screen width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired screen height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget.

Note

Some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#).

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the TAB_MODE value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the <code>TAB_MODE</code> Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-134: `TAB_MODE` Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero and any attempt to change it is ignored when running a widget application on the UNIX platform.

When a `WIDGET_TAB` widget receives focus through tabbing, the tab title is outlined. Navigate to other tabs using the **End** key, **Home** key, or the right and left arrow keys to switch focus between the tabs. Pressing the **Tab** key brings into focus the first child widget on the tab capable of receiving focus. On UNIX, the Motif library controls tabbing functionality.

After creating the widget hierarchy, you can change tabbing support using the [`WIDGET_CONTROL`](#) procedure’s `TAB_MODE` keyword, or query a widget’s support for tabbing using the [`WIDGET_INFO`](#) procedure’s `TAB_MODE` keyword.

See [“Tabbing in Widget Applications”](#) in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see [“`TRACKING_EVENTS`”](#) on page 2273 in the documentation for `WIDGET_BASE`.

UNAME

Set this keyword to a string, which is used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (which is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The user value to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. You should avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. You should avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) affect the behavior of tab widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: [BASE_SET_TITLE](#), [SET_TAB_CURRENT](#), [SET_TAB_MULTILINE](#).

Keywords to WIDGET_INFO

Some keywords to the [WIDGET_INFO](#) return information that applies specifically to tab widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: [TAB_CURRENT](#), [TAB_MULTILINE](#), [TAB_NUMBER](#).

Widget Events Returned by Tab Widgets

Tab widgets generate events when a new tab is selected. The event structure returned by the [WIDGET_EVENT](#) function is defined by the following statement:

```
{WIDGET_TAB, ID:0L, TOP:0L, HANDLER:0L, TAB:0L}
```

ID is the widget ID of the button generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget associated with the handler routine. TAB returns the zero-based index of the tab selected.

Version History

Introduced: 5.6

TAB_MODE keyword added: 6.1

See Also

[“Using Tab Widgets”](#) in Chapter 30 of the *Building IDL Applications* manual

WIDGET_TABLE

The WIDGET_TABLE function creates table widgets. Table widgets display two-dimensional data and allow in-place data editing. They can have one or more rows and columns, and automatically create scroll bars when viewing more data than can otherwise be displayed on the screen.

For a more detailed discussion of the table widget, along with examples, see “Using Table Widgets” in Chapter 30 of the *Building IDL Applications* manual.

Note on Table Sizing

Table widgets are sized according to the value of the following pairs of keywords to WIDGET_TABLE, in order of precedence: SCR_XSIZE/SCR_YSIZE, XSIZE/YSIZE, X_SCROLL_SIZE/Y_SCROLL_SIZE, VALUE. If either dimension remains unspecified by one of the above keywords, the default value of six (columns or rows) is used when the table is created. If the width or height specified is less than the size of the table, scroll bars are added automatically.

Syntax

```
Result = WIDGET_TABLE( Parent [, ALIGNMENT={0 | 1 | 2}] [, /ALL_EVENTS]
  [, AM_PM=[string, string]] [, COLUMN_LABELS=string_array]
  [, /COLUMN_MAJOR | /ROW_MAJOR] [, COLUMN_WIDTHS=array]
  [, /CONTEXT_EVENTS] [, DAYS_OF_WEEK=string_array{7 names}]
  [, /DISJOINT_SELECTION] [, /EDITABLE] [, EVENT_FUNC=string]
  [, EVENT_PRO=string] [, FONT=string] [, FORMAT=value] [, FRAME=width]
  [, FUNC_GET_VALUE=string] [, GROUP_LEADER=widget_id]
  [, IGNORE_ACCELERATORS=value] [, /KBRD_FOCUS_EVENTS]
  [, KILL_NOTIFY=string] [, MONTHS=string_array{12 names}]
  [, /NO_COLUMN_HEADERS] [, /NO_COPY] [, /NO_HEADERS]
  [, /NO_ROW_HEADERS] [, NOTIFY_REALIZE=string]
  [, PRO_SET_VALUE=string] [, /RESIZEABLE_COLUMNS]
  [, /RESIZEABLE_ROWS{not supported in Windows}]
  [, RESOURCE_NAME=string] [, ROW_HEIGHTS=array{not supported in
  Windows}] [, ROW_LABELS=string_array] [, SCR_XSIZE=width]
  [, SCR_YSIZE=height] [, /SCROLL] [, /SENSITIVE] [, TAB_MODE=value]
  [, /TRACKING_EVENTS] [, UNAME=string] [, UNITS={0 | 1 | 2}]
  [, UVALUE=value] [, VALUE=value] [, XOFFSET=value] [, XSIZE=value]
  [, X_SCROLL_SIZE=width] [, YOFFSET=value] [, YSIZE=value]
  [, Y_SCROLL_SIZE=height] )
```

Return Value

The returned value of this function is the widget ID of the newly-created table widget.

Arguments

Parent

The widget ID of the parent widget for the new table widget.

Keywords

ALIGNMENT

Set this keyword equal to a scalar or 2-D array specifying the alignment of the text within each cell. An alignment of 0 (the default) aligns the left edge of the text with the left edge of the cell. An alignment of 2 right-justifies the text, while 1 results in text centered within the cell. If ALIGNMENT is set equal to a scalar, all table cells are aligned as specified. If ALIGNMENT is set equal to a 2-D array, the alignment of each table cell is governed by the corresponding element of the array.

ALL_EVENTS

Set this keyword to cause the text widget to generate events whenever the user changes the contents of a table cell.

If the EDITABLE keyword is set, an insert character event (TYPE=0) is generated when the user presses the RETURN or ENTER key in the text widget, even if the ALL_EVENTS keyword is not set. See the table below for details on the interaction between ALL_EVENTS and EDITABLE.

Keywords		Effects	
ALL_EVENTS	EDITABLE	Input changes widget contents?	Type of events generated.
Not set	Not set	No	None
Not set	Set	Yes	End-of-line insertion
Set	Not set	No	All events
Set	Set	Yes	All events

Table 3-135: Effects of using the ALL_EVENTS and EDITABLE keywords

AM_PM

Supplies a string array of two names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the `FORMAT` keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

COLUMN_LABELS

Set this keyword equal to an array of strings used as labels for the columns of the table widget. The default labels are of the form “Column *n*”, where *n* is the column number. If this keyword is set to the empty string (' '), all column labels are set to be empty.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

COLUMN_MAJOR

This keyword is only valid if the table data is organized as a vector of structures rather than a two-dimensional array. See the [VALUE](#) keyword for details.

Set this keyword to specify that the data should be read into the table as if each element of the vector is a structure containing one column’s data. Note that the structures must all be of the same type, and must have one field for each row in the table. If this keyword is not set, the table widget behaves as if the `ROW_MAJOR` keyword were set.

COLUMN_WIDTHS

Set this keyword equal to an array of widths for the columns of the table widget. The widths are given in any of the units as specified with the `UNITS` keyword. If no width is specified for a column, that column is set to the default size, which varies by platform. If `COLUMN_WIDTHS` is set to a scalar value, all columns are set to that width.

CONTEXT_EVENTS

Set this keyword to cause context menu events (or simply context events) to be issued when you click the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. This is the default. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus). Pass the context event ID to the [WIDGET_DISPLAYCONTEXTMENU](#) procedure within your widget program's event handler to display the context menu.

For more on detecting and handling context menu events, see [“Context-Sensitive Menus”](#) in Chapter 30 of the *Building IDL Applications* manual.

DAYS_OF_WEEK

Supplies a string array of seven names to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the `FORMAT` keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

DISJOINT_SELECTION

Set this keyword to enable the ability to select multiple rectangular regions of cells. The regions can be overlapping, touching, or entirely distinct.

Setting this keyword changes the data structures returned by the [TABLE_SELECT](#) keyword to `WIDGET_INFO` and the [GET_VALUE](#) keyword to `WIDGET_CONTROL`. Similarly, the data structures you supply via the [SET_TABLE_SELECT](#) and [SET_VALUE](#) keywords to `WIDGET_CONTROL` are different in disjoint mode.

See [“Selection Modes”](#) in Chapter 30 of the *Building IDL Applications* manual for additional details.

EDITABLE

Set this keyword to allow direct user editing of the text widget contents. Normally, the text in text widgets is read-only. See [ALL_EVENTS](#) for a description of how `EDITABLE` interacts with the `ALL_EVENTS` keyword.

Note

The method by which text widgets are placed into edit mode is dependent upon the windowing system. See “[Edit Mode](#)” in the *Building IDL Applications* manual

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

A single font is shared by the row and column labels and by all of the cells in the widget.

Note

On Microsoft Windows platforms, if FONT is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts; in general, the system default font is the font appropriate for the version of Windows in question.

FORMAT

Set this keyword equal to a single string or array of strings that specify the format of data displayed within table cells. The string(s) are of the same form as used by the FORMAT keyword to the PRINT procedure, and the default format is the same as that used by the PRINT procedure.

Warning

If the format specified is incompatible with the data displayed in a table cell, an error message is generated. Since the error is generated *for each cell displayed*, the number of messages printed is potentially large, and can slow execution

significantly. Note also that each time a new cell is displayed (when scroll bars are repositioned, for example), a new error is generated *for each cell displayed*.

FRAME

The value of this keyword specifies the width of a frame in units specified by the UNITS keyword (pixels are the default) to be drawn around the borders of the widget. Note that this keyword is only a “hint” to the toolkit, and may be ignored in some instances.

FUNC_GET_VALUE

A string containing the name of a function to be called when the GET_VALUE keyword to the WIDGET_CONTROL procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The WIDGET_CONTROL procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

IGNORE_ACCELERATORS

Set this keyword to specify what WIDGET_BUTTON accelerators are to be ignored when this table widget has keyboard focus. Valid values are:

- A string or string array containing any value that is legal for WIDGET_BUTTON’s [ACCELERATOR](#) keyword
- The number 1, indicating all accelerators should be ignored

Ordinarily, accelerators are processed before keyboard events reach the widget that has the keyboard focus. Consider a case where the accelerator “Ctrl+V” has been mapped to a button that allows the user to paste objects copied from one drawing area into another. In an application that also has a table, attempting to use **Ctrl+V** to paste text into the table cell would fail as this event would be stolen by the accelerator. Setting IGNORE_ACCELERATORS to “Ctrl+V” allows a table widget with an editable text area to receive keyboard events instead of the button with a conflicting

accelerator. Once a table loses focus, all specified accelerators are automatically re-enabled.

Note

Accelerators are suspended only when a table cell is in edit mode.

See “[Disabling Button Widget Accelerators](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

KBRD_FOCUS_EVENTS

Set this keyword to make the base return keyboard focus events whenever the keyboard focus of the base changes. See “[Widget Events Returned by Table Widgets](#)” on page 2475 for more information.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the WIDGET_CONTROL procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the WIDGET_EVENT function is called.

MONTHS

Supplies a string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the FORMAT keyword.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

NO_COLUMN_HEADERS

Set this keyword to disable the display of the table widget’s column header area. The setting for the NO_HEADERS keyword supersedes any setting for this keyword.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_TABLE` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NO_HEADERS

Set this keyword to disable the display of the table widget’s header area (where row and column labels are normally displayed).

NO_ROW_HEADERS

Set this keyword to disable the display of the table widget’s row header area. The setting for the `NO_HEADERS` keyword supersedes any setting for this keyword.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESIZEABLE_COLUMNS

Set this keyword to allow the user to change the size of columns using the mouse. Note that if the `NO_HEADERS` keyword was set, the columns cannot be resized interactively.

RESIZEABLE_ROWS

Set this keyword to allow the user to change the size of rows using the mouse. Note that if the `NO_HEADERS` keyword was set, the rows cannot be resized interactively.

Under Microsoft Windows, the row size cannot be changed.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See [“RESOURCE_NAME”](#) on page 2266 for a complete discussion of this keyword.

ROW_HEIGHTS

Set this keyword equal to an array of heights for the rows of the table widget. The heights are given in any of the units as specified with the `UNITS` keyword. If no height is specified for a row, that row is set to the default size, which varies by platform. If `ROW_HEIGHTS` is set to a scalar value, all of the row heights are set to that value.

Note

This keyword is not supported under Microsoft Windows.

ROW_LABELS

Set this keyword equal to an array of strings to be used as labels for the rows of the table. If no label is specified for a row, it receives the default label “Row *n*”, where *n* is the row number. If this keyword is set to the empty string (' '), all row labels are set to be empty.

ROW_MAJOR

This keyword is only valid if the table data is organized as a vector of structures rather than a two-dimensional array. See the [VALUE](#) keyword for details.

Set this keyword to specify that the data should be read into the table as if each element of the vector is a structure containing one row’s data. Note that the structures must all be of the same type, and must have one field for each column in the table.

This is the default behavior if neither the `COLUMN_MAJOR` or `ROW_MAJOR` keyword is set.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the `UNITS` keyword (pixels are the default). Note that the screen width of the widget *includes* the width of scroll bars, if any are present. Setting `SCR_XSIZE` overrides values set for the `XSIZE` or `X_SCROLL_SIZE` keywords. See “[Note on Table Sizing](#)” on page 2461.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the `UNITS` keyword (pixels are the default). Note that the screen height of the widget *includes* the height of scroll bars, if any are present. Setting `SCR_YSIZE` overrides values set for the `YSIZE` or `Y_SCROLL_SIZE` keywords. See “[Note on Table Sizing](#)” on page 2461.

SCROLL

Set this keyword to give the widget scroll bars that allow viewing portions of the widget contents that are not currently on the screen. See “[Note on Table Sizing](#)” on page 2461

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If `SENSITIVE` is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the `SENSITIVE` keyword with the [WIDGET_CONTROL](#) procedure.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The `TAB_MODE` setting is

inherited by lower-level bases and child widgets from the parent `WIDGET_BASE` unless it is explicitly set on an individual widget. If the `TAB_MODE` value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the <code>TAB_MODE</code> has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-136: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

When a `WIDGET_TABLE` widget receives focus through tabbing, the result is platform dependent:

- On Windows – table widgets can receive and lose focus through tabbing, but the focus is not clearly depicted.
- On UNIX – the Motif library controls tabbing functionality.

After creating the widget hierarchy, you can change tabbing support using the [WIDGET_CONTROL](#) procedure's `TAB_MODE` keyword, or query a widget's support for tabbing using the [WIDGET_INFO](#) procedure's `TAB_MODE` keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for `WIDGET_BASE`.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the `FIND_BY_UNAME` keyword. The `UNAME` should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set `UNITS` equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

Note

This keyword does not affect all sizing operations. Specifically, the value of `UNITS` is ignored when setting the `XSIZE` or `YSIZE` keywords.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If `UVALUE` is not present, the widget's initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a table widget is either a two-dimensional array or a vector of structures.

If the value is specified as a two-dimensional array, all data must be of the same data type.

If the value is specified as a vector of structures, it can be displayed either in column-major or row-major format by setting either the [COLUMN_MAJOR](#) keyword or the [ROW_MAJOR](#) keyword. All of the structures must be of the same type, and must contain one field for each row (if [COLUMN_MAJOR](#) is set) or column (if [ROW_MAJOR](#) is set) in the table. If neither keyword is set, the data is displayed in row major format.

Note

If the [VALUE](#) keyword is not specified, the data in the created table will be of type [STRING](#).

If none of [\[XY\]SIZE](#), [SCR_\[XY\]SIZE](#), or [\[XY\]_SCROLL_SIZE](#) is present, the size of the table is determined by the size of the array or vector of structures specified by [VALUE](#). See “[Note on Table Sizing](#)” on page 2461.

XOFFSET

The horizontal offset of the widget in units specified by the [UNITS](#) keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The width of the widget in columns. If row labels are present, one column is automatically added to this value. See “[Note on Table Sizing](#)” on page 2461.

X_SCROLL_SIZE

The [XSIZE](#) keyword always specifies the width of a widget, in columns. When the [SCROLL](#) keyword is specified, this size is not necessarily the same as the width of the visible area. The [X_SCROLL_SIZE](#) keyword allows you to set the width of the

scrolling viewport independently of the actual width of the widget. See “[Note on Table Sizing](#)” on page 2461.

Use of the `X_SCROLL_SIZE` keyword implies `SCROLL`. This means that scroll bars will be added in both the horizontal and vertical directions when `X_SCROLL_SIZE` is specified. Because the default size of the scrolling viewport may differ between platforms, it is best to specify `Y_SCROLL_SIZE` when specifying `X_SCROLL_SIZE`.

YOFFSET

The vertical offset of the widget in units specified by the `UNITS` keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The height of the widget in rows. If column labels are present, one row is automatically added to this value. See “[Note on Table Sizing](#)” on page 2461.

Y_SCROLL_SIZE

The `YSIZE` keyword always specifies the height of a widget. in rows. When the `SCROLL` keyword is specified, this size is not necessarily the same as the height of the visible area. The `Y_SCROLL_SIZE` keyword allows you to set the height of the scrolling viewport independently of the actual width of the widget. See “[Note on Table Sizing](#)” on page 2461.

Use of the `Y_SCROLL_SIZE` keyword implies `SCROLL`. This means that scroll bars will be added in both the horizontal and vertical directions when `Y_SCROLL_SIZE` is specified. Because the default size of the scrolling viewport may differ between platforms, it is best to specify `X_SCROLL_SIZE` when specifying `Y_SCROLL_SIZE`.

Keywords to WIDGET_CONTROL

A number of keywords to the `WIDGET_CONTROL` procedure affect the behavior of table widgets. In addition to those keywords that affect all widgets, the following are particularly useful: `ALIGNMENT`, `ALL_TABLE_EVENTS`, `COLUMN_LABELS`,

COLUMN_WIDTHS, CONTEXT_EVENTS, DELETE_COLUMNS,
 DELETE_ROWS, EDITABLE, EDIT_CELL, FORMAT, GET_VALUE,
 INSERT_COLUMNS, INSERT_ROWS, KBRD_FOCUS_EVENTS,
 ROW_LABELS, ROW_HEIGHTS, SET_TABLE_SELECT, SET_TABLE_VIEW,
 SET_TEXT_SELECT, SET_VALUE, TABLE_BLANK,
 TABLE_DISJOINT_SELECTION, TABLE_XSIZE, TABLE_YSIZE,
 USE_TABLE_SELECT, USE_TEXT_SELECT.

Keywords to WIDGET_INFO

A number of keywords to the `WIDGET_INFO` function return information that applies specifically to table widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: `COLUMN_WIDTHS`, `CONTEXT_EVENTS`, `KBRD_FOCUS_EVENTS`, `ROW_HEIGHTS`, `TABLE_ALL_EVENTS`, `TABLE_DISJOINT_SELECTION`, `TABLE_EDITABLE`, `TABLE_EDIT_CELL`, `TABLE_SELECT`, `TABLE_VIEW`, `USE_TABLE_SELECT`.

Widget Events Returned by Table Widgets

There are several variations of the table widget event structure depending on the specific event being reported. All of these structures contain the standard three fields (`ID`, `TOP`, and `HANDLER`). Several contain an integer `TYPE` field that indicates which type of structure has been returned. Others, such as keyboard focus and context menu event structures, do not. If an event structure is known to have a `TYPE` field, check this first before branching to further event handling code. If it is possible that the event structure is a keyboard focus or context menu type, check the `STRUCTURE_NAME` of the event structure first. The different table widget event structures are described below.

Insert Single Character (`TYPE = 0`)

This is the type of structure returned when a single character is typed into a cell of a table widget by a user.

```
{WIDGET_TABLE_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L,  
  CH:0B, X:0L, Y:0L }
```

`OFFSET` is the (zero-based) insertion position that will result after the character is inserted. `CH` is the ASCII value of the character. `X` and `Y` give the zero-based address of the cell within the table.

Insert Multiple Characters (TYPE = 1)

This is the type of structure returned when multiple characters are pasted into a cell by the window system.

```
{WIDGET_TABLE_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L,
  STR:'', X:0L, Y:0L}
```

OFFSET is the (zero-based) insertion position that will result after the text is inserted. STR is the string to be inserted. X and Y give the zero-based address of the cell within the table.

Delete Text (TYPE = 2)

This is the type of structure returned when any amount of text is deleted from a cell of a table widget.

```
{WIDGET_TABLE_DEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:2, OFFSET:0L,
  LENGTH:0L, X:0L, Y:0L}
```

OFFSET is the (zero-based) character position of the first character deleted. It is also the insertion position that will result when the next character is inserted. LENGTH gives the number of characters involved. X and Y give the zero-based address of the cell within the table.

Text Selection (TYPE = 3)

This is the type of structure returned when an area of text is selected (highlighted) by the user.

```
{WIDGET_TABLE_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:3,
  OFFSET:0L, LENGTH:0L, X:0L, Y:0L}
```

The event announces a change in the insertion point. OFFSET is the (zero-based) character position of the first character to be selected. LENGTH gives the number of characters involved. A LENGTH of zero indicates that the widget has no selection, and that the insertion position is given by OFFSET. X and Y give the zero-based address of the cell within the table.

Note

Text insertion, text deletion, or any change in the current insertion point causes any current selection to be lost. In such cases, the loss of selection is implied by the text event reporting the insert/delete/movement and a separate zero length selection event is not sent.

Cell Selection (TYPE = 4)

This is the type of structure returned when range of cells is selected (highlighted) or deselected by the user.

```
{WIDGET_TABLE_CELL_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:4,
  SEL_LEFT:0L, SEL_TOP:0L, SEL_RIGHT:0L, SEL_BOTTOM:0L}
```

The event announces a change in the currently selected cells. The range of cells selected is given by the zero-based indices into the table specified by the SEL_LEFT, SEL_TOP, SEL_RIGHT, and SEL_BOTTOM fields.

Note

If the table is in disjoint selection mode, selecting an additional region will result in a new WIDGET_TABLE_CELL_SEL event that reflects only the newly-selected cells. Use WIDGET_INFO, /TABLE_SELECT to obtain the entire selected region in disjoint selection mode.

When cells are deselected (either by changing the selection or by clicking in the upper left corner of the table) an event is generated in which the SEL_LEFT, SEL_TOP, SEL_RIGHT, and SEL_BOTTOM fields contain the value -1. This means that two WIDGET_TABLE_CELL_SEL events are generated when an existing selection is changed to a new selection. If your code pays attention to WIDGET_TABLE_CELL_SEL events, be sure to differentiate between select and deselect events.

Note

If the table is in disjoint selection mode and a new cell range is selected *starting on a previously-selected cell*, the previously selected cells are deselected but a WIDGET_TABLE_CELL_DESEL event is generated for the deselection rather than a WIDGET_TABLE_CELL_SEL event. See “[Cell Deselection \(Disjoint Mode\) \(TYPE = 9\)](#)” on page 2478 for details.

Row Height Changed (TYPE = 6)

This is the type of structure returned when a row height is changed by the user.

```
{WIDGET_TABLE_ROW_HEIGHT, ID:0L, TOP:0L, HANDLER:0L, TYPE:6,
  ROW:0L, HEIGHT:0L}
```

The event announces that the height of the given row has been changed by the user. The ROW field contains the zero-based row number, and the HEIGHT field contains the new height.

Column Width Changed (TYPE = 7)

This is the type of structure returned when a column width is changed by the user.

```
{WIDGET_TABLE_COL_WIDTH, ID:0L, TOP:0L, HANDLER:0L, TYPE:7,
  COL:0L, WIDTH:0L}
```

The event announces that the width of the given column has been changed by the user. The COL field contains the zero-based column number, and the WIDTH field contains the new width.

Invalid Data (TYPE = 8)

This is the type of structure returned when the text entered by the user does not pass validation, and the user has finished editing the field (by hitting TAB or ENTER).

```
{WIDGET_TABLE_INVALID_ENTRY, ID:0L, TOP:0L, HANDLER:0L, TYPE:8,
  STR:'', X:0L, Y:0L}
```

When this event is generated, the cell's data is left unchanged. The invalid contents entered by the user is given as a text string in the STR field. The cell location is given by the X and Y fields.

Cell Deselection (Disjoint Mode) (TYPE = 9)

This is the type of structure returned when selected cells are de-selected by the user *and the table is in disjoint selection mode*. It is identical to the (TYPE = 4) WIDGET_TABLE_CELL_SEL event structure except for the name and type value.

This event occurs when the user holds down the control key when starting a selection and the cell used to start the selection was already selected. In contrast, if the user starts a selection with the control key down but starts on a cell that was not selected, the normal WIDGET_TABLE_CELL_SEL is generated.

```
{WIDGET_TABLE_CELL_DESEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:9,
  SEL_LEFT:0L, SEL_TOP:0L, SEL_RIGHT:0L, SEL_BOTTOM:0L}
```

The range of cells selected is given by the zero-based indices into the table specified by the SEL_LEFT, SEL_TOP, SEL_RIGHT, and SEL_BOTTOM fields.

Keyboard Focus Events

Table widgets return the following event structure when the keyboard focus changes and the base was created with the KBRD_FOCUS_EVENTS keyword set:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

ID is the widget ID of the table widget generating the event. TOP is the widget ID of the top level widget containing ID. HANDLER contains the widget ID of the widget

associated with the handler routine. The ENTER field returns 1 (one) if the table widget is gaining the keyboard focus, or 0 (zero) if the table widget is losing the keyboard focus.

Context Menu Events

A table widget created with the CONTEXT_EVENTS keyword set returns the following event structure in response to a right mouse button click:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,  
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the table widget. The ROW and COL fields return meaningful information for table widgets and values of zero (0) for other widgets.

For table widgets, ROW and COL indicate the zero-based index of the cell that was clicked on when the context menu was initiated. The upper-left data cell has a row and column index of 0,0. Row and column headers have indices of -1. If the context-menu event takes place outside of all table cells and headers, then both ROW and COL will have values of -1.

Note

When working with table widget context menu events or keyboard focus events, it is important to notice that the event structures do not have a TYPE field, so special code is needed for the table event handler. Instead of keying off the TYPE field, use the event structure's name. See [“Property Sheet Context Menu Example”](#) on page 2438 for sample code dealing with the same issue.

Version History

Introduced: 5.0

DISJOINT_SELECTION keyword, Cell deselection (type 9) event added: 5.6

ROW and COL fields of WIDGET_CONTEXT event structure added: 6.1

CONTEXT_EVENTS, IGNORE_ACCELERATORS, NO_COLUMN_HEADERS, NO_ROW_HEADERS, and TAB_MODE keywords added: 6.1

See Also

[WIDGET_CONTROL](#)

WIDGET_TEXT

The WIDGET_TEXT function creates text widgets. Text widgets display text and optionally get textual input from the user. They can have 1 or more lines, and can optionally contain scroll bars to allow viewing more text than can otherwise be displayed on the screen.

Syntax

```
Result = WIDGET_TEXT( Parent [, /ALL_EVENTS] [, /CONTEXT_EVENTS]
  [, /EDITABLE] [, EVENT_FUNC=string] [, EVENT_PRO=string]
  [, FONT=string] [, FRAME=width] [, FUNC_GET_VALUE=string]
  [, GROUP_LEADER=widget_id] [, IGNORE_ACCELERATORS=value]
  [, /KBRD_FOCUS_EVENTS] [, KILL_NOTIFY=string] [, /NO_COPY]
  [, /NO_NEWLINE] [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
  [, RESOURCE_NAME=string] [, SCR_XSIZE=width] [, SCR_YSIZE=height]
  [, /SCROLL] [, /SENSITIVE] [, TAB_MODE=value] [, /TRACKING_EVENTS]
  [, UNAME=string] [, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=value]
  [, /WRAP] [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value]
  [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created text widget.

Arguments

Parent

The widget ID of the parent widget for the new text widget.

Keywords

ALL_EVENTS

Set this keyword to cause the text widget to generate events whenever the user changes the contents of the text area.

Note

If the EDITABLE keyword is set, an insert character event (TYPE=0) is generated when the user presses the RETURN or ENTER key in the text widget, even if the

ALL_EVENTS keyword is not set. See the table below for details on the interaction between ALL_EVENTS and EDITABLE.

Keywords		Effects	
ALL_EVENTS	EDITABLE	Input changes widget contents?	Type of events generated.
Not set	Not set	No	None
Not set	Set	Yes	End-of-line insertion
Set	Not set	No	All events
Set	Set	Yes	All events

Table 3-137: Effects of using the ALL_EVENTS and EDITABLE keywords

CONTEXT_EVENTS

Set this keyword to cause context menu events (or simply context events) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-sensitive menus (also known as pop-up or shortcut menus); pass the context event ID to the [WIDGET_DISPLAYCONTEXTMENU](#) procedure within your widget program's event handler to display the context menu.

For more on detecting and handling context menu events, see “[Context-Sensitive Menus](#)” in Chapter 30 of the *Building IDL Applications* manual.

EDITABLE

Set this keyword to allow direct user editing of the text widget contents. Normally, the text in text widgets is read-only. See [ALL_EVENTS](#) for a description of how EDITABLE interacts with the ALL_EVENTS keyword.

See “[ALL_TEXT_EVENTS](#)” on page 2309 for a description of how EDITABLE interacts with the ALL_TEXT_EVENTS keyword.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

FONT

The name of the font to be used by the widget. The font specified is a “device font” (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

Note

On Microsoft Windows platforms, if `FONT` is not specified, IDL uses the system default font. Different versions of Windows use different system default fonts.

FRAME

The value of this keyword specifies the width of a frame in units specified by the `UNITS` keyword (pixels are the default) to be drawn around the borders of the widget.

Note

This keyword is only a “hint” to the toolkit, and may be ignored in some instances. Under Microsoft Windows, text widgets *always* have frames.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as “group leader” for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. It is not possible to remove a widget from an existing group.

IGNORE_ACCELERATORS

Set this keyword to specify what WIDGET_BUTTON accelerators are to be ignored when this text widget has keyboard focus. Valid values are:

- A string or string array containing any value that is legal for WIDGET_BUTTON's [ACCELERATOR](#) keyword
- The number 1, indicating all accelerators should be ignored

Ordinarily, accelerators are processed before keyboard events reach the widget that has the keyboard focus. Consider a case where the accelerator “Ctrl+V” has been mapped to a button that allows the user to paste objects copied from one drawing area into another. In an application that also has a text widget, attempting to use **Ctrl+V** to paste text into the text widget would fail as this event would be stolen by the accelerator. Setting IGNORE_ACCELERATORS to “Ctrl+V” allows a text widget with an editable text area to receive keyboard events instead of the button with a conflicting accelerator. Once the text widget loses focus, all specified accelerators are automatically re-enabled.

See “[Disabling Button Widget Accelerators](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

KBRD_FOCUS_EVENTS

Set this keyword to make the base return keyboard focus events whenever the keyboard focus of the base changes. See “[Widget Events Returned by Text Widgets](#)” on page 2489 for more information.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' ').

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the WIDGET_CONTROL procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the WIDGET_EVENT function is called.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the SET_UVALUE and GET_UVALUE keywords to WIDGET_CONTROL, IDL

makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_TEXT` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NO_NEWLINE

Normally, when setting the value of a multi-line text widget, newline characters are automatically appended to the end of each line of text. Set this keyword to suppress this action.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such “callback” procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

RESOURCE_NAME

A string containing an X Window System resource name to be applied to the widget. See “[RESOURCE_NAME](#)” on page 2266 for a complete discussion of this keyword.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the XSIZE keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SCROLL

Set this keyword to give the widget scroll bars that allow viewing portions of the widget contents that are not currently on the screen.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the TAB_MODE value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the TAB_MODE has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-138: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

After creating the widget hierarchy, you can change tabbing support using the [WIDGET_CONTROL](#) procedure’s TAB_MODE keyword, or query a widget’s support for tabbing using the [WIDGET_INFO](#) procedure’s TAB_MODE keyword.

See [“Tabbing in Widget Applications”](#) in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see [“TRACKING_EVENTS”](#) on page 2273 in the documentation for WIDGET_BASE.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

Note

This keyword does not affect all sizing operations. Specifically, the value of UNITS is ignored when setting the `XSIZE` or `YSIZE` keywords to `WIDGET_TEXT`.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget’s initial user value is undefined.

VALUE

The initial value setting of the widget. The value of a text widget is the current text displayed by the widget.

VALUE can be either a string or an array of strings. Note that variables returned by the `GET_VALUE` keyword to `WIDGET_CONTROL` are always string arrays, even if a scalar string is specified in the call to `WIDGET_TEXT`.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, “Using Language Catalogs”](#) in the *Building IDL Applications* manual.

WRAP

Set this keyword to indicate that scrolling or multi-line text widgets should automatically break lines between words to keep the text from extending past the right edge of the text display area. Note that carriage returns are *not* automatically entered when lines wrap; the value of the text widget will remain a single-element array unless you explicitly enter a carriage return.

Note

On Windows, the maximum line length for a multi-line text widget is 1024 characters, regardless of this keyword value.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

XSIZE

The width of the widget in characters. Note that the physical width of the text widget depends on both the value of XSIZE and on the size of the font used. The default value of XSIZE varies according to your windowing system. On Windows, the default size is roughly 20 characters. On Motif, the default size depends on the width of the text widget.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget programming.

YSIZE

The height of the widget in text lines. Note that the physical height of the text widget depends on both the value of YSIZE and on the size of the font used. The default value of YSIZE is one line.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) procedure affect the behavior of text widgets. In addition to those keywords that affect all widgets, the following are particularly useful: [ALL_TEXT_EVENTS](#), [APPEND](#), [CONTEXT_EVENTS](#), [EDITABLE](#), [GET_VALUE](#), [KBRD_FOCUS_EVENTS](#), [INPUT_FOCUS](#), [NO_NEWLINE](#), [SET_TEXT_SELECT](#), [SET_TEXT_TOP_LINE](#), [SET_VALUE](#), [USE_TEXT_SELECT](#).

Keywords to WIDGET_INFO

A number of keywords to the [WIDGET_INFO](#) function return information that applies specifically to text widgets. In addition to those keywords that apply to all widgets, the following are particularly useful: [CONTEXT_EVENTS](#), [KBRD_FOCUS_EVENTS](#), [TEXT_ALL_EVENTS](#), [TEXT_EDITABLE](#), [TEXT_NUMBER](#), [TEXT_OFFSET_TO_XY](#), [TEXT_SELECT](#), [TEXT_TOP_LINE](#), [TEXT_XY_TO_OFFSET](#).

Widget Events Returned by Text Widgets

There are several variations of the text widget event structure depending on the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER) as well as an integer TYPE field that indicates which type of structure has been returned. Programs should always check the type field before referencing fields that are not present in all text event structures. The different text widget event structures are described below.

Insert Single Character (TYPE = 0)

This is the type of structure returned when a single character is typed or pasted into a text widget by a user.

```
{ WIDGET_TEXT_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L,
  CH:0B }
```

OFFSET is the (zero-based) insertion position that will result after the character is inserted. CH is the ASCII value of the character.

Insert Multiple Characters (TYPE = 1)

This is the type of structure returned when multiple characters are pasted into a text widget by the window system.

```
{ WIDGET_TEXT_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L,
  STR:' ' }
```

OFFSET is the (zero-based) insertion position that will result after the text is inserted. STR is the string to be inserted.

Delete Text (TYPE = 2)

This is the type of structure returned when any amount of text is deleted from a text widget.

```
{ WIDGET_TEXT_DEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:2, OFFSET:0L,
  LENGTH:0L }
```

OFFSET is the (zero-based) character position of the first character to be deleted. It is also the insertion position that will result when the characters have been deleted. LENGTH gives the number of characters involved. A LENGTH of zero indicates that no characters were deleted.

Selection (TYPE = 3)

This is the type of structure returned when an area of text is selected (highlighted) by the user.

```
{ WIDGET_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:3, OFFSET:0L,
  LENGTH:0L }
```

The event announces a change in the insertion point. OFFSET is the (zero-based) character position of the first character to be selected. LENGTH gives the number of characters involved. A LENGTH of zero indicates that no characters are selected, and the new insertion position is given by OFFSET.

Note that text insertion, text deletion, or any change in the current insertion point causes any current selection to be lost. In such cases, the loss of selection is implied by the text event reporting the insert/delete/movement and a separate zero length selection event is *not* sent.

Keyboard Focus Events

Text widgets return the following event structure when the keyboard focus changes and the base was created with the `KBRD_FOCUS_EVENTS` keyword set:

```
{ WIDGET_KBRD_FOCUS, ID:0L, TOP:0L, HANDLER:0L, ENTER:0 }
```

The first three fields are the standard fields found in every widget event. The `ENTER` field returns 1 (one) if the text widget is gaining the keyboard focus, or 0 (zero) if the text widget is losing the keyboard focus.

Context Menu Events

A text widget created with the `CONTEXT_EVENTS` keyword set returns the following event structure in response to a right mouse button click:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,  
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The `X` and `Y` fields give the device coordinates at which the event occurred, measured from the upper left corner of the text widget. The `ROW` and `COL` fields return meaningful information for table widgets and values of zero (0) for other widgets.

Version History

Introduced: Pre 4.0

`IGNORE_ACCELERATORS`, `TAB_MODE` keyword added: 6.1

`ROW` and `COL` fields of `WIDGET_CONTEXT` event structure added: 6.1

See Also

[CW_FIELD](#), [XDISPLAYFILE](#)

WIDGET_TREE

The WIDGET_TREE function is used to create and populate a tree widget. The tree widget presents a hierarchical view that can be used to organize a wide variety of data structures and information.

The WIDGET_TREE function performs two separate tasks: creating the tree widget and populating the tree widget with *nodes* (branches and leaves).

For a more detailed discussion of the tree widget, along with examples, see “Using Tree Widgets” in Chapter 30 of the *Building IDL Applications* manual.

Syntax

```
Result = WIDGET_TREE( Parent [, /ALIGN_BOTTOM | , /ALIGN_CENTER |
, /ALIGN_LEFT | , /ALIGN_RIGHT | , /ALIGN_TOP] [, BITMAP=array]
[, /CONTEXT_EVENTS] [, EVENT_FUNC=string] [, EVENT_PRO=string]
[, /EXPANDED] [, /FOLDER] [, FUNC_GET_VALUE=string]
[, GROUP_LEADER=widget_id] [, KILL_NOTIFY=string] [, /MULTIPLE]
[, /NO_COPY] [, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string]
[, SCR_XSIZE=width] [, SCR_YSIZE=height] [, /SENSITIVE]
[, TAB_MODE=value] [, /TRACKING_EVENTS] [, /TOP] [, UNAME=string]
[, UNITS={0 | 1 | 2}] [, UVALUE=value] [, VALUE=string] [, XOFFSET=value]
[, XSIZE=value] [, YOFFSET=value] [, YSIZE=value] )
```

Return Value

The returned value of this function is the widget ID of the newly-created tree widget.

Arguments

Parent

The widget ID of the parent for the new tree widget. *Parent* can be either a base widget or a tree widget.

- If *Parent* is a base widget, WIDGET_TREE will create a tree widget that contains no other tree widgets. This type of tree widget is referred to as a *root node*.
- If *Parent* is a tree widget, WIDGET_TREE will create a new tree widget (called a *node*) in the specified tree widget.

Note

With the exception of the first tree widget created (the *root node*, whose *Parent* is a base widget), a tree widget (or *node*) must be created with the **FOLDER** keyword in order to serve as the *Parent* for other tree widgets.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a **ROW** base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a **ROW** or **COLUMN** base. In **ROW** bases, the new widget will be vertically centered. In **COLUMN** bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a **COLUMN** base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a **COLUMN** base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a **ROW** base.

BITMAP

Set this keyword equal to a 16x16x3 array representing an RGB image that will be displayed next to the node in the tree widget.

CONTEXT_EVENTS

Set this keyword to cause context menu events (or simply context events) to be issued when the user clicks the right mouse button over the widget. Set the keyword to 0 (zero) to disable such events. Context events are intended for use with context-

sensitive menus (also known as pop-up or shortcut menus); pass the context event ID to the `WIDGET_DISPLAYCONTEXTMENU` within your widget program's event handler to display the context menu.

For more on detecting and handling context menu events, see “Context-Sensitive Menus” in Chapter 30 of the *Building IDL Applications* manual.

This keyword is only valid if the *Parent* of the tree widget is a base widget.

EVENT_FUNC

A string containing the name of a function to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EXPANDED

If the tree node being created is a *folder* (specified by the `FOLDER` keyword), set this keyword to cause the folder to be initially displayed expanded, showing all of its immediate child entries. By default, folders are initially displayed collapsed.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

FOLDER

Set this keyword to cause the tree node being created to act as a *folder* (that is, as a branch of the tree rather than a leaf).

Note

With the exception of the root node (the tree widget whose *Parent* widget is a base widget), only tree nodes that have the `FOLDER` keyword set can act as the parent for other tree widgets.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this

technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

GROUP_LEADER

The widget ID of an existing widget that serves as group leader for the newly-created widget. When a group leader is killed, for any reason, all widgets in the group are also destroyed.

A given widget can be in more than one group. The `WIDGET_CONTROL` procedure can be used to add additional group associations to a widget. You cannot remove a widget from an existing group.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such callback procedure. This callback procedure can be removed by setting the routine name to the null string (' '). Note that the procedure specified is used only if you are not using the `XMANAGER` procedure to manage your widgets.

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

If you use the `XMANAGER` procedure to manage your widgets, the value of this keyword is overwritten. Use the `CLEANUP` keyword to `XMANAGER` to specify a procedure to be called when a managed widget dies.

MULTIPLE

Set this keyword to enable multiple selection operations in the tree widget. If enabled, multiple elements in the tree widget can be selected at one time by holding down the **Control** or **Shift** key while clicking the left mouse button.

This keyword is only valid if the *Parent* of the tree widget is a base widget.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique works well for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copying the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. Upon a set operation (using the `UVALUE` keyword to `WIDGET_TAB` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. Upon a get operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string containing the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such callback procedure. This callback procedure can be removed by setting the routine name to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

SCR_XSIZE

Set this keyword to the desired screen width of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `XSIZE` keyword.

SCR_YSIZE

Set this keyword to the desired screen height of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `YSIZE` keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget.

Note

Some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#).

TAB_MODE

Set this keyword to one of the values shown in the table below to determine how the widget hierarchy can be navigated using the **Tab** key. The TAB_MODE setting is inherited by lower-level bases and child widgets from the parent [WIDGET_BASE](#) unless it is explicitly set on an individual widget. If the TAB_MODE value of the widget differs from that of the base, the setting on the widget will be respected when the widget has focus. For example, if a base does not support tabbing, but an individual child widget does support tabbing, this functionality will be enabled when the child widget has focus.

Note

It is not possible to tab to disabled (SENSITIVE=0) or hidden (MAP=0) widgets.

Valid settings are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the TAB_MODE has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “Inheriting the TAB_MODE Value” in Chapter 30 of the <i>Building IDL Applications</i> manual.
1	Enable navigation onto and off of the widget.

Table 3-139: TAB_MODE Keyword Options

Value	Description
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-139: TAB_MODE Keyword Options

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform.

Once a WIDGET_TREE widget receives focus through tabbing, additional navigation capabilities are platform dependent:

- On Windows – tree widgets can receive and lose focus through tabbing.
- On UNIX - the Motif library controls tabbing functionality.

Use the arrow keys to select higher or lower level nodes once a tree widget has focus. The TAB_MODE specified for the root tree widget defines the tabbing support for all of its nodes.

After creating the widget hierarchy, you can change tabbing support using the [WIDGET_CONTROL](#) procedure's TAB_MODE keyword, or query a widget's support for tabbing using the [WIDGET_INFO](#) procedure's TAB_MODE keyword.

See “[Tabbing in Widget Applications](#)” in Chapter 30 of the *Building IDL Applications* manual for usage details and examples.

TRACKING_EVENTS

Set this keyword to cause widget tracking events to be issued for the widget whenever the mouse pointer enters or leaves the region covered by that widget. For the structure of tracking events, see “[TRACKING_EVENTS](#)” on page 2273 in the documentation for WIDGET_BASE.

TOP

Set this keyword to cause the tree node being created to be inserted as the parent node's top entry. By default, new nodes are inserted as the parent node's bottom entry.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) with the `FIND_BY_UNAME` keyword. The UNAME should be unique to the widget hierarchy because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (which is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The user value to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

VALUE

Set this keyword equal to a string containing the text that will be displayed next to the tree node. If this keyword is not set, the default value `Tree` is used.

Note

You can use language catalogs to internationalize this value with strings in particular languages. For more information, see [Chapter 24, "Using Language Catalogs"](#) in the *Building IDL Applications* manual.

This keyword is only valid if the *Parent* of the tree widget is another tree widget.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. You should avoid using this style of widget programming.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. You should avoid using this style of widget programming.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a hint to the toolkit and may be ignored in some situations.

Keywords to WIDGET_CONTROL

A number of keywords to the [WIDGET_CONTROL](#) affect the behavior of tree widgets. In addition to those keywords that affect all widgets, the following keywords are particularly useful: [SET_TREE_BITMAP](#), [SET_TREE_EXPANDED](#), [SET_TREE_SELECT](#), [SET_TREE_VISIBLE](#).

Keywords to WIDGET_INFO

Some keywords to the [WIDGET_INFO](#) return information that applies specifically to tree widgets. In addition to those keywords that apply to all widgets, the following keywords are particularly useful: [TREE_EXPANDED](#), [TREE_SELECT](#), and [TREE_ROOT](#).

Widget Events Returned by Tree Widgets

Several variations of the tree widget event structure depend upon the specific event being reported. All of these structures contain the standard three fields (ID, TOP, and HANDLER) as well as an integer TYPE field that indicates which type of structure has been returned. Programs should always check the type field before referencing fields that are not present in all tree event structures. The different tree widget event structures are described below.

Select (TYPE = 0)

This structure is returned when the currently selected node in the tree widget changes:

```
{WIDGET_TREE_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, CLICKS:0L}
```

The CLICKS field indicates the number of mouse-button clicks that occurred when the event took place. This field contains 1 (one) when the item is selected, or 2 when the user double-clicks on the item.

Expand (TYPE = 1)

This structure is returned when a folder in the tree widget expands or collapses:

```
{WIDGET_TREE_EXPAND, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, EXPAND:0L}
```

The EXPAND field contains 1 (one) if the folder expanded or 0 (zero) if the folder collapsed.

Context Menu Events

A tree widget created with the CONTEXT_EVENTS keyword set returns the following event structure in response to a right mouse button click:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,  
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The X and Y fields give the device coordinates at which the event occurred, measured from the upper left corner of the tree widget. The ROW and COL fields return meaningful information for table widgets and values of zero (0) for other widgets.

Version History

Introduced: 5.6

TAB_MODE keyword added: 6.1

ROW and COL fields of WIDGET_CONTEXT event structure added: 6.1

See Also

[“Using Tree Widgets”](#) in Chapter 30 of the *Building IDL Applications* manual

WINDOW

The WINDOW procedure creates a window for the display of graphics or text. It is only necessary to use WINDOW if more than one simultaneous window or a special size window is desired because a window is created automatically the first time any display procedure attempts to access the window system. The newly-created window becomes the current window, and the system variable !D.WINDOW is set to that window's window index. (See the description of the WSET procedure for a discussion of the current IDL window.)

The behavior of WINDOW varies slightly depending on the window system in effect. See the discussion of IDL graphics devices in [Appendix A, "IDL Graphics Devices"](#) for additional details.

Syntax

```
WINDOW [, Window_Index] [, COLORS=value] [, /FREE] [, /PIXMAP]
      [, RETAIN={0 | 1 | 2}] [, TITLE=string] [, XPOS=value] [, YPOS=value]
      [, XSIZE=pixels] [, YSIZE=pixels]
```

Arguments

Window_Index

The window index for the newly-created window. A window index is an integer value between 0 and 31 that is used to refer to the window. If this parameter is omitted, window index 0 is used. If the value of *Window_Index* specifies an existing window, the existing window is deleted and a new one is created. If you need to create more than 32 windows, use the FREE keyword described below.

Keywords

COLORS

Note

This keyword is ignored under Microsoft Windows.

The maximum number of color table indices to be used when drawing. This keyword has an effect only if supplied when the first window is created. If COLORS is not present when the first window is created, all or most of the available color indices are allocated depending upon the window system in use.

To use monochrome windows on a color display in X Windows, use `COLORS = 2` when creating the first window. One color table is maintained for all windows. A negative value for `COLORS` specifies that all but the given number of colors from the shared color table should be allocated.

Although this keyword is ignored under Microsoft Windows, we could use the following code to use a monochrome window on all platforms:

```
WINDOW, COLORS=2 ; ignored on Windows
white=!D.N_COLORS-1
PLOT, FINDGEN(20), COLOR=white
```

FREE

Set this keyword to create a window using the smallest unused window index above 32. If this keyword is present, the *Window_Index* argument can be omitted. The default position of the new window is opposite that of the current window. Using the `FREE` keyword allows the creation of a large number of windows. The system variable `!D.WINDOW` is set to the index of the newly-created window.

PIXMAP

Set the `PIXMAP` keyword to specify that the window being created is actually an invisible portion of the display memory called a pixmap.

RETAIN

Set this keyword to 0, 1, or 2 to specify how backing store should be handled for the window:

- 0 = no backing store
- 1 = requests that the server or window system provide backing store
- 2 = specifies that IDL provide backing store directly

See “[Backing Store](#)” on page 4286 for details.

TITLE

A scalar string that contains the window’s label. If not specified, the window is given a label of the form “IDL *n*”, where *n* is the index number of the window. For example, to create a window with the label “IDL Graphics”, enter:

```
WINDOW, TITLE='IDL Graphics'
```

XPOS

The X position of the window, specified in device coordinates. On Motif platforms, XPOS specifies the X position of the *lower* left corner and is measured from the lower left corner of the screen. On Windows platforms, XPOS specifies the X position of the *upper* left corner and is measured from the upper left corner of the screen. That is, specifying

```
WINDOW, XPOS = 0, YPOS = 0
```

will create a window in the lower left corner on Motif machines and in the upper left corner on Windows machines.

If no position is specified, the position of the window is determined from the value of Window Index using the following rules:

- Window 0 is placed in the upper right hand corner.
- Even numbered windows are placed on the top half of the screen and odd numbered windows are placed on the bottom half.
- Windows 0,1,4,5,8, and 9 are placed on the right side of the screen and windows 2,3,6, and 7 are placed on the left.

Note

The order of precedence (highest to lowest) for positioning windows is: XPOS/YPOS keywords to WINDOW, Tile/Cascade IDE graphics (user system) preferences, optional index argument to WINDOW. Also realize that setting LOCATION is only a request to the Window manager and may not always be honored due to system peculiarities.

YPOS

The Y position of the window, specified in device coordinates. See the description of XPOS for details.

XSIZE

The width of the window in pixels.

YSIZE

The height of the window in pixels.

Examples

Create graphics window number 0 with a size of 400 by 400 pixels and a title that reads “Square Window” by entering:

```
WINDOW, 0, XSIZE=400, YSIZE=400, TITLE='Square Window'
```

Version History

Introduced: Original

See Also

[WDELETE](#), [WSET](#), [WSHOW](#)

WRITE_BMP

The `WRITE_BMP` procedure writes an image and its color table vectors to a Microsoft Windows Version 3 device independent bitmap file (.BMP).

`WRITE_BMP` does not handle 1-bit-deep images or compressed images, and is not fast for 4-bit images. The algorithm works best on images where the number of bytes in each scan-line is evenly divisible by 4.

This routine is written in the IDL language. Its source code can be found in the file `write_bmp.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WRITE_BMP, Filename, Image [, R, G, B] [, /FOUR_BIT] [, IHDR=structure]  
          [, HEADER_DEFINE=h{define h before call}] [, /RGB]
```

Arguments

Filename

A scalar string containing the full pathname of the bitmap file to write.

Image

The array to write into the new bitmap file. The array should be scaled into a range of bytes for 8- and 24-bit deep images. Scale to 0-15 for 4-bit deep images. If the image has 3 dimensions and the first dimension is 3, a 24-bit deep bitmap file is created.

Note

For 24-bit images, color interleaving is blue, green, red: $Image[0, i, j]$ = blue, $Image[1, i, j]$ = green, $Image[2, i, j]$ = red.

R, G, B

Color tables. If omitted, the colors loaded in the `COLORS` common block are used.

Keywords

FOUR_BIT

Set this keyword to write as a 4-bit device independent bitmap. If omitted or zero, an 8-bit deep bitmap is written.

IHDR

Set this keyword to a BITMAPINFOHEADER structure containing the file header fields that are not obtained from the image itself. The fields in this structure that can be set are: `bi{XY}PelsPerMeter`, `biClrUsed`, and `biClrImportant`.

HEADER_DEFINE

If this keyword is set, `WRITE_BMP` returns an empty BITMAPINFOHEADER structure, containing zeros. No other actions are performed. This structure may be then modified with the pertinent fields and passed in via the `IHDR` keyword parameter. See the Microsoft Windows Programmers Reference Guide for a description of each field in the structure.

Note: this parameter must be defined *before* the call. For example:

```
H = 0
WRITE_BMP, HEADER_DEFINE = H
```

RGB

Set this keyword to reverse the color interleaving for 24-bit images to red, green, blue: $Image[0, i, j] = \text{red}$, $Image[1, i, j] = \text{green}$, $Image[2, i, j] = \text{blue}$. By default, 24-bit images are written with color interleaving of blue, green, red.

Examples

The following command captures the contents of the current IDL graphics window and saves it to a Microsoft Windows Bitmap file with the name `test.bmp`. Note that this works only on a PseudoColor (8-bit) display:

```
WRITE_BMP, 'test.bmp', TVRD()
```

The following commands scale an image to 0-15, and then write a 4-bit BMP file, using a grayscale color table:

```
; Create a ramp from 0 to 255:
r = BYTSCL(INDGEN(16))

WRITE_BMP, 'test.bmp', BYTSCL(Image, MAX=15), r, r, r, /FOUR
```

Version History

Introduced: Pre 4.0

See Also

[READ_BMP](#), [QUERY_*](#) Routines

WRITE_GIF

The WRITE_GIF procedure writes an image and its color table vectors to a Graphics Interchange Format (GIF) file.

WRITE_GIF produces 8-bit GIF files of the standard type: non-interlaced, global colormap.

Note

The GIF format uses compression technologies that may be subject to license restrictions. If you receive a warning when attempting to use this routine, contact RSI for information about licensing options.

This routine is written in the IDL language. Its source code can be found in the file `write_gif.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WRITE_GIF, Filename, Image [, R, G, B] [, /MULTIPLE [, /CLOSE]]
```

Arguments

Filename

A scalar string containing the full pathname of the GIF file to write.

Image

The array to write into the new GIF file.

R, G, B

The Red, Green, and Blue color vectors to be written with to the GIF file. If *R*, *G*, *B* values are not provided, the last color table established using LOADCT is saved, and the table is padded to 256 entries. If no color table has been established, WRITE_GIF calls LOADCT to load the grayscale entry (table 0).

Keywords

CLOSE

Set this keyword to close any open files. The CLOSE keyword is only useful if a file containing multiple images (as specified by the MULTIPLE keyword) is being written. If the CLOSE keyword is specified, nothing is written to the file, and all other parameters are ignored.

MULTIPLE

Set this keyword to write multiple images to a file. Each call to WRITE_GIF writes the next image, with the file remaining open between calls. The *Filename* argument is ignored after the first call, *but must be supplied*. After the first image has been written, any R, G, and B color vectors supplied are ignored. All images written to a GIF file must be the same size.

Example

The following command captures the contents of the current IDL graphics window and saves it to a GIF file named `test.gif`. Note that this works only on a PseudoColor (8-bit) display:

```
WRITE_GIF, 'test.gif', TVRD()
```

Version History

Introduced: Pre 4.0

See Also

[READ_GIF](#), [WRITE_JPEG](#), [QUERY_*](#) Routines

WRITE_IMAGE

The WRITE_IMAGE procedure writes an image and its color table vectors, if any, to a file of a specified type. WRITE_IMAGE can write most types of image files supported by IDL.

Syntax

```
WRITE_IMAGE, Filename, Format, Data [, Red, Green, Blue] [, /APPEND]
```

Arguments

Filename

A scalar string containing the name of the file to write.

Format

A scalar string containing the name of the file format to write. The following are the supported formats:

- BMP
- GIF
- JPEG
- PNG
- PPM
- SRF
- TIFF

Data

An IDL variable containing the image data to write to the file.

Red

An optional vector containing the red channel of the color table if a colortable exists.

Green

An optional vector containing the green channel of the color table if a colortable exists.

Blue

An optional vector containing the blue channel of the color table if a colortable exists.

Keywords

APPEND

Set this keyword to force the image to be appended to the file instead of overwriting the file. APPEND may be used with image formats that supports multiple images per file and is ignored for formats that support only a single image per file.

Version History

Introduced: 5.3

WRITE_JPEG

The `WRITE_JPEG` procedure writes compressed images to files. JPEG (Joint Photographic Experts Group) is a standardized compression method for full-color and gray-scale images. This procedure is based in part on the work of the Independent JPEG Group.

As the Independent JPEG Group states, JPEG is intended for real-world scenes (such as digitized photographs). Line art, such as drawings or IDL plots, and other unrealistic images are not its strong suit. Note also that JPEG is a lossy compression scheme. That is, the output image is *not* identical to the input image. Hence you cannot use JPEG if you must have identical output bits. On typical images of real-world scenes, however, very good compression levels can be obtained with no visible change, and amazingly high compression levels are possible if you can tolerate a low-quality image. You can trade off output image quality against compressed file size by adjusting a compression parameter. Files are encoded in JFIF, the JPEG File Interchange Format; however, such files are usually simply called JPEG files.

If you need to store images in a format that uses lossless compression, consider using the `WRITE_PNG` procedure. This procedure writes a Portable Network Graphics (PNG) file using lossless compression with either 8 or 16 data bits per channel. To store 8-bit or 24-bit images without compression, consider using `WRITE_BMP` (for Microsoft Bitmap format files) or `WRITE_TIFF` (to write Tagged Image Format Files).

For a short technical introduction to the JPEG compression algorithm, see: Wallace, Gregory K. "The JPEG Still Picture Compression Standard", *Communications of the ACM*, April 1991 (vol. 34, no. 4), pp. 30-44.

Note

All JPEG files consist of byte data. Input data is converted to bytes before being written to a JPEG file.

Syntax

```
WRITE_JPEG [, Filename | , UNIT=lun] , Image [, /ORDER] [, /PROGRESSIVE]
           [, QUALITY=value{0 to 100}] [, TRUE={1 | 2 | 3}]
```

Arguments

Filename

A string containing the name of file to be written in JFIF (JPEG) format. If this parameter is not present, the UNIT keyword must be specified.

Image

A byte array of either two or three dimensions, containing the image to be written. Grayscale images must have two dimensions. TrueColor images must have three dimensions with the index of the dimension that contains the color specified with the TRUE keyword.

Keywords

ORDER

JPEG/JFIF images are normally written in top-to-bottom order. If the image array is in the standard IDL order (i.e., from bottom-to-top) set ORDER to 0, its default value. If the image array is in top-to-bottom order, ORDER must be set to 1.

PROGRESSIVE

Set this keyword to write the image as a series of scans of increasing quality. When used with a slow communications link, a decoder can generate a low-quality image very quickly, and then improve its quality as more scans are received.

Warning

Not all JPEG applications can handle progressive JPEG files, and it is up to the JPEG reader to progressively display the JPEG image. For example, IDL's READ_JPEG routine ignores the progressive readout request and reads the entire image in at the first reading.

QUALITY

This keyword specifies the quality index, in the range of 0 (terrible) to 100 (excellent) for the JPEG file. The default value is 75, which corresponds to very good quality. Lower values of QUALITY produce higher compression ratios and smaller files.

TRUE

This keyword specifies the index, starting at 1, of the dimension over which the color is interleaved. For example, for an image that is pixel interleaved and has dimensions of $(3, m, n)$, set TRUE to 1. Specify 2 for row-interleaved images $(m, 3, n)$; and 3 for band-interleaved images $(m, n, 3)$. If TRUE is not set, the image is assumed to have no interleaving (it is not a TrueColor image).

UNIT

This keyword designates the logical unit number of an already open file to receive the output, allowing multiple JFIF images per file or the embedding of JFIF images in other data files. If this keyword is used, *Filename* should not be specified.

Note

When opening a file intended for use with the UNIT keyword, if the filename does not end in `.jpg`, or `.jpeg`, you must specify the `STDIO` keyword to `OPEN` in order for the file to be compatible with `WRITE_JPEG`.

Examples

Write the image contained in the array A, using JPEG compression with a quality index of 25. The image is stored in bottom-to-top order:

```
image = DIST(100)
WRITE_JPEG, 'test1.jpg', image, QUALITY=25
```

Write a TrueColor image to a JPEG file. The image is contained in the band-interleaved array A with dimensions $(m, n, 3)$. Assume it is stored in top-to-bottom order:

```
WRITE_JPEG, 'test2.jpg', image, TRUE=3, /ORDER
```

Version History

Introduced: Pre 4.0

See Also

[READ_JPEG](#), [WRITE_GIF](#), [QUERY_* Routines](#)

WRITE_JPEG2000

The `WRITE_JPEG2000` procedure writes image data into a JPEG2000 file. This procedure is a wrapper around the `IDLffJPEG2000` object interface that presents JPEG2000 image writing in a familiar way to users of the `WRITE_*` image routines. However this procedure is not as efficient as the object interface and the object interface should be used whenever possible. See “`IDLffJPEG2000`” in Chapter 6 of the *IDL Reference Guide* manual for information about the object interface.

Syntax

```
WRITE_JPEG2000, Filename, Image [, Red, Green, Blue] [, N_LAYERS=value]  
[, N_LEVELS=value] [, /ORDER] [, /REVERSIBLE] )
```

Arguments

Filename

A scalar string containing the full path and filename of the JPEG2000 file to write. If the file suffix is `.jp2` or `.jpx` (case insensitive), the file is written using the JP2 file format. Otherwise the file is written as a raw JPEG2000 code stream.

Image

A byte or integer array of either two or three dimensions, containing the image to be written. Single channel images must have dimensions of [*Width*, *Height*]. Multi-channel images must have dimensions of [*nComponents*, *Width*, *Height*], where *nComponents* is the number of components present in the image.

Red, Green, Blue

For single-channel images, *Red*, *Green*, and *Blue* should contain the red, green, and blue color vectors to be stored within the JPEG2000 palette. For multi-channel images, these arguments are ignored.

Keywords

N_LAYERS

Set this keyword to a positive integer specifying the number of quality layers. Each layer contains the information required to represent the image at a higher quality, given the information from all the previous layers. A larger number of layers takes

longer to encode and produces a larger file, but provides more flexibility when decoding. The default is 1. If `N_LAYERS` is greater than 1, the layers will be assigned roughly logarithmically-spaced bit rates, with the last layer containing all of the compressed bits.

N_LEVELS

Set this keyword to an integer specifying the number of wavelet decomposition levels, or stages, in the range of 0...15. A value of zero indicates that no wavelet transform will be used. Higher values will typically give better compression. If this keyword is omitted, then the default value of 5 is used. If the file is in JP2 format, and a *Red, Green, Blue* palette is present, then the default is `N_LEVELS=0`.

ORDER

JPEG2000 images are assumed to be stored in top-to-bottom order, while IDL usually assumes images are in bottom-to-top order. By default (`ORDER = 0`), `WRITE_JPEG2000` will automatically flip the input arrays to match JPEG2000's top-to-bottom order. Setting `ORDER` to a nonzero value will save the image directly to the JPEG2000 file without doing the vertical flip.

REVERSIBLE

Set this keyword to specify that reversible (lossless) compression should be used. The default is to use irreversible (lossy) compression. If the file is in JP2 format, and a Red, Green, Blue palette is present, then the default is `REVERSIBLE=1`.

Example

For an example, see [READ_JPEG2000](#).

Version History

Introduced: 6.1

See Also

[QUERY_JPEG2000](#), [READ_JPEG2000](#)

WRITE_NRIF

The `WRITE_NRIF` procedure writes an image and its color table vectors to an NCAR Raster Interchange Format (NRIF) rasterfile.

`WRITE_NRIF` only writes 8- or 24-bit deep rasterfiles of types “Indexed Color” (8-bit) and “Direct Color integrated” (24-bit). The color map is included only for 8-bit files.

See the document “NCAR Raster Interchange Format and TAGS Raster Reference Manual,” available from the Scientific Computing Division, National Center for Atmospheric Research, Boulder, CO, 80307-3000, for information on the structure of NRIF files.

This routine is written in the IDL language. Its source code can be found in the file `write_nrif.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WRITE_NRIF, File, Image [, R, G, B]
```

Arguments

File

A scalar string containing the full path name of the NRIF file to write.

Image

The byte array to be written to the NRIF file. If *Image* has the dimensions (n,m) , an 8-bit NRIF file with color tables is created. If *Image* has the dimensions $(3,n,m)$, a 24-bit NRIF file is created, where each byte triple represents the red, green, and blue intensities at (n,m) on a scale from 0 to 255. The NRIF image will be rendered from bottom to top, in accordance with IDL standards.

R, G, B

The Red, Green, and Blue color vectors to be used as a color table with 8-bit images. If color vectors are supplied, they are included in the output (8-bit images only). If *R*, *G*, *B* values are not provided, the last color table established using `LOADCT` is included. If no color table has been established, `WRITE_NRIF` calls `LOADCT` to load the grayscale entry (table 0).

Note

WRITE_NRIF does not recognize color vectors loaded directly using TVLCT, so if a custom color table is desired and it is not convenient to use XPALETTE, include the R, G, and B vectors that were used to create the color table.

Version History

Introduced: Pre 4.0

WRITE_PICT

The `WRITE_PICT` procedure writes an image and its color table vectors to a PICT (version 2) format image file. The PICT format is used by Apple Macintosh computers.

Note: `WRITE_PICT` only works with 8-bit displays

This routine is written in the IDL language. Its source code can be found in the file `write_pict.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WRITE_PICT, Filename [, Image, R, G, B]
```

Arguments

Filename

A scalar string containing the full pathname of the PICT file to write.

Image

The byte array to be written to the PICT file. If *Image* is omitted, the entire current graphics window is read into an array and written to the PICT file.

R, G, B

The Red, Green, and Blue color vectors to be written to the PICT file. If *R*, *G*, *B* values are not provided, the last color table established using `LOADCT` is included. If no color table has been established, `WRITE_PICT` calls `LOADCT` to load the grayscale entry (table 0).

Examples

Create a pseudo screen dump from the current window. Note that this works only on a PseudoColor (8-bit) display:

```
WRITE_PICT, 'test.pict', TVRD()
```

Version History

Introduced: Pre 4.0

See Also

[READ_PICT](#), [QUERY_*](#) Routines

WRITE_PNG

The `WRITE_PNG` procedure writes a 2-D or 3-D IDL variable into a Portable Network Graphics (PNG) file. The data in the file is stored using lossless compression with either 8 or 16 data bits per channel, based on the input IDL variable type. 3-D IDL variables must have the number of channels as their leading dimension (pixel interleaved). For `BYTE` format 2-D IDL variables, an optional palette may be stored in the image file along with a list of pixel values which are to be considered transparent by a reading program.

Note

IDL supports version 1.0.5 of the PNG Library.

Syntax

```
WRITE_PNG, Filename, Image [, R, G, B] [, /ORDER] [, /VERBOSE]  
          [, TRANSPARENT=array]
```

Arguments

Filename

A scalar string containing the full pathname of the PNG file to write.

Image

The array to write into the new PNG file. If *Image* is one of the integer data types, it is converted to type unsigned integer (UINT) and written out at 16 data bits per channel. All other data types are converted to bytes and written out at 8-bits per channel.

Note

If *Image* is two-dimensional (single-channel) and *R*, *G*, and *B* are provided, all input data types (including integer) are converted to bytes and written out as 8-bit data.

R, G, B

For single-channel images, *R*, *G*, and *B* should contain the red, green, and blue color vectors, respectively. For multi-channel images, these arguments are ignored.

Keywords

ORDER

Set this keyword to indicate that the rows of the image should be written from bottom to top. The rows are written from top to bottom by default. ORDER provides compatibility with PNG files written using versions of IDL prior to IDL 5.4, which wrote PNG files from bottom to top.

VERBOSE

Produces additional diagnostic output during the write.

TRANSPARENT

Set this keyword to an array of pixel index values which are to be treated as “transparent” for the purposes of image display. This keyword is valid only if *Image* is a single-channel (color indexed) image and the R, G, B palette is provided.

Examples

Create an RGBA (16-bits/channel) and a Color Indexed (8-bits/channel) image with a palette.

```

rgbdata = UINDGEN(4,320,240)
cidata = BYTSCL(DIST(256))
red = INDGEN(256)
green = INDGEN(256)
blue = INDGEN(256)
WRITE_PNG, 'rgb_image.png', rgbdata
WRITE_PNG, 'ci_image.png', cidata, red, green, blue

; Query and Read the data:
names = ['rgb_image.png', 'ci_image.png', 'unknown.png']
FOR i=0,N_ELEMENTS(names)-1 DO BEGIN
  ok = QUERY_PNG(names[i],s)
  IF (ok) THEN BEGIN
    HELP,s,/STRUCTURE
    IF (s.HAS_PALETTE) THEN BEGIN
      img = READ_PNG(names[i],rpal,gpal,bpal)
      HELP,img,rpal,gpal,bpal
    ENDIF ELSE BEGIN
      img = READ_PNG(names[i])
      HELP,img
    ENDELSE
  ENDIF ELSE BEGIN

```

```
        PRINT,names[i],' is not a PNG file'  
    ENDELSE  
ENDFOR
```

Version History

Introduced: 5.2

See Also

[READ_PNG](#), [QUERY_*](#) Routines

WRITE_PPM

The `WRITE_PPM` procedure writes an image to a PPM (TrueColor) or PGM (gray scale) file. This routine is written in the IDL language. Its source code can be found in the file `write_ppm.pro` in the `lib` subdirectory of the IDL distribution.

Note

`WRITE_PPM` only writes 8-bit deep PGM/PPM files of the standard type. Images should be ordered so that the first row is the top row.

PPM/PGM format is supported by the `PBMPLUS` toolkit for converting various image formats to and from portable formats, and by the `Netpbm` package.

Syntax

```
WRITE_PPM, Filename, Image [, /ASCII]
```

Arguments

Filename

A scalar string specifying the full pathname of the PPM or PGM file to write.

Image

The 2D (gray scale) or 3D (TrueColor) array to be written to a file.

Keywords

ASCII

Set this keyword to force `WRITE_PPM` to use formatted ASCII input/output to write the image data. The default is to use the far more efficient binary input/output (RAWBITS) format.

Examples

```
image = DIST(100)  
WRITE_PPM, 'file.ppm', image
```

Version History

Introduced: 4.0

See Also

[READ_PPM](#), [QUERY_*](#) Routines

WRITE_SPR

The WRITE_SPR procedure writes a row-indexed sparse array structure to a specified file. Row-indexed sparse arrays are created using the SPRSIN function.

Syntax

```
WRITE_SPR, AS, Filename
```

Arguments

AS

A row-indexed sparse array created by SPRSIN.

Filename

The name of the file that will contain AS.

Keywords

None.

Examples

```
; Create an array:
A = [[3.,0., 1., 0., 0.],$
      [0.,4., 0., 0., 0.],$
      [0.,7., 5., 9., 0.],$
      [0.,0., 0., 0., 2.],$
      [0.,0., 0., 6., 5.]]

; Convert it to sparse storage format:
A = SPRSIN(A)

; Store it in the file sprs.as:
WRITE_SPR, A, 'sprs.as'
```

Version History

Introduced: Pre 4.0

See Also

[FULSTR](#), [LINBCG](#), [SPRSAB](#), [SPRSAX](#), [SPRSIN](#), [READ_SPR](#)

WRITE_SRF

The `WRITE_SRF` procedure writes an image and its color table vectors to a Sun Raster File (SRF).

`WRITE_SRF` only writes 32-, 24-, and 8-bit-deep rasterfiles of type `RT_STANDARD`. Use the UNIX command `rasfilter8to1` to convert these files to 1-bit deep files. See the file `/usr/include/rasterfile.h` for the structure of Sun rasterfiles.

This routine is written in the IDL language. Its source code can be found in the file `write_srf.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WRITE_SRF, Filename [, Image, R, G, B] [, /ORDER] [, /WRITE_32]
```

Arguments

Filename

A scalar string containing the full pathname of the SRF to write.

Image

The array to be written to the SRF. If *Image* has dimensions $(3,n,m)$, a 24-bit SRF is written. If *Image* is omitted, the entire current graphics window is read into an array and written to the SRF file. *Image* should be of byte type, and in top to bottom scan line order.

R, G, B

The Red, Green, and Blue color vectors to be written to the file. If *R*, *G*, *B* values are not provided, the last color table established using `LOADCT` is included. If no color table has been established, `WRITE_SRF` calls `LOADCT` to load the grayscale entry (table 0).

Keywords

ORDER

Set this keyword to write the image from the top down instead of from the bottom up. This setting is only necessary when writing a file from the current IDL graphics window; it is ignored when writing a file from a data array passed as a parameter.

WRITE_32

Set this keyword to write a 32-bit file. If the input image is a TrueColor image, dimensioned (3, n , m), it is normally written as a 24-bit raster file.

Examples

Create a pseudo screen dump from the current window:

```
WRITE_SRF, 'test.srf', TVRD()
```

Version History

Introduced: Original

See Also

[READ_SRF](#), [QUERY_* Routines](#)

WRITE_SYLK

The WRITE_SYLK function writes the contents of an IDL variable to a SYLK (Symbolic Link) format spreadsheet data file.

Note

This routine writes only numeric and string SYLK data. It cannot handle spreadsheet and cell formatting information (cell width, text justification, font type, date, time, monetary notations, etc.). A given SYLK data file cannot be appended with data blocks through subsequent calls.

This routine is written in the IDL language. Its source code can be found in the file `write_sylk.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = WRITE_SYLK( File, Data [, STARTCOL=column] [, STARTROW=row] )
```

Return Value

The function returns TRUE if the write operation was successful.

Arguments

File

A scalar string specifying the full path name of the SYLK file to write.

Data

A scalar, vector, or 2D array to be written to *File*.

Keywords

STARTCOL

Set this keyword to the first column of spreadsheet cells to write. If not specified, the write operation begins with the first column found in the file (column 0).

STARTROW

Set this keyword to the first row of spreadsheet cells to write. If not specified, the write operation begins with the first row of cells found in the file (row 0).

Examples

Suppose you wish to write the contents of a 2 by 2 floating-point array, `data`, to a SYLK data file called “`bar.slk`” such that the matrix would appear with its upper left data at the cell in the 10th row and the 20th column. Use the following command:

```
status = WRITE_SYLK('bar.slk', data, STARTROW = 9, STARTCOL = 19)
```

The IDL variable `status` will contain the value 1 if the operation was successful.

Version History

Introduced: 4.0

See Also

[READ_SYLK](#)

WRITE_TIFF

The WRITE_TIFF procedure can write TIFF files with one or more channels, where each channel can contain 1, 4, 8, 16, or 32-bit integer pixels, or floating-point values.

Syntax

```
WRITE_TIFF, Filename [, Image] [, /APPEND]
  [, BITS_PER_SAMPLE={1 | 4 | 8}] [, RED, GREEN, BLUE=value] [, /CMYK]
  [, COMPRESSION={0 | 1 | 2 | 3}] [, DESCRIPTION=string]
  [, DOCUMENT_NAME=string] [, DOT_RANGE=intarray]
  [, GEOTIFF=structure] [, /LONG | /SHORT | /FLOAT]
  [, ICC_PROFILE=bytearray] [, ORIENTATION=value] [, PHOTOSHOP=array]
  [, PLANARCONFIG={1 | 2}] [, UNITS={1 | 2 | 3}] [, /VERBOSE]
  [, XPOSITION=units] [, XRESOL=pixels/inch] [, YPOSITION=units]
  [, YRESOL=pixels/inch]
```

Arguments

Filename

A scalar string containing the full pathname of the TIFF to write.

Image

The array to be written to the TIFF. If *Image* has dimensions (k,n,m), a k-channel TIFF is written. *Image* should be in top to bottom scan line order. By default, this array is converted to byte format before being written (see the LONG, SHORT and FLOAT keywords below). Note that many TIFF readers can read only one- or three-channel images.

Note

The Image argument is ignored if PLANARCONFIG is set to 2 and the RED, GREEN, and BLUE keywords have been set to 2D arrays.

Note

Grayscale TIFF images are written out with the PhotometricInterpretation tag set to BlackIsZero, implying that values of 0 should correspond to black. If you want values of 0 to correspond to white, you should invert your pixel values before calling WRITE_TIFF.

Order

This argument is obsolete. The Order argument has been replaced by the ORIENTATION keyword. Code that uses the Order argument will continue to work as before, but new code should use the ORIENTATION keyword instead.

Keywords

APPEND

Set this keyword to specify that the image should be added to the existing file, creating a multi-image TIFF file.

BITS_PER_SAMPLE

Set this keyword to either 1, 4, or 8 to create a grayscale image file with the specified number of bits per pixel. For 1-bit (bi-level) images, an output pixel is assigned the value 1 (one) if the corresponding input pixel is nonzero. For 4-bit grayscale images, the input pixel values *must* be in the range 0 through 15, or the image will be garbled. The default is BITS_PER_SAMPLE=8. This keyword is ignored if an RGB image or color palette is present, or if one of the FLOAT, LONG, or SHORT keywords is set.

CMYK

Set this keyword to indicate that *Image* contains a 4-channel image, where the channels represent colors in the CMYK color-separated format. This corresponds to setting PhotometricInterpretation = 5 in the resulting TIFF file, which allows TIFF readers to interpret the image as CMYK. The *Image* argument can either be (4, *Columns*, *Rows*) or (*Columns*, *Rows*, 4) depending upon the PLANARCONFIG keyword.

Note

If *Image* has 4 channels but the CMYK keyword is not set, then the image is written as a standard multi-channel image, with PhotometricInterpretation = 2.

COMPRESSION

Set this keyword to select the type of compression to be used:

- 0 = none (default)
- 1 = LZW
- 2 = PackBits

- 3 = JPEG (TIFF files)

Note

For COMPRESSION=3 (JPEG), all images are assumed to be in top-to-bottom order, and the ORIENTATION keyword should not be specified. If your input image is not in top-to-bottom order, you should use REVERSE(*image*, 2) to flip the order before calling WRITE_TIFF.

DESCRIPTION

Set this keyword to a scalar string containing the value of the ImageDescription tag to be stored in the TIFF file. If this keyword is not provided then the ImageDescription tag is set to "IDL TIFF file".

DOCUMENT_NAME

Set this keyword to a scalar string containing the value of the DocumentName tag to be stored in the TIFF file. If this keyword is not provided then the DocumentName tag is set to the filename.

DOT_RANGE

Set this keyword to a two-element integer array containing the TIFF DotRange tag value. If this keyword is not provided then the DotRange tag is not stored. The DotRange tag is typically only present in CMYK TIFF files. DOT_RANGE[0] gives the image value that corresponds to a 0% dot, while DOT_RANGE[1] gives the image value that corresponds to a 100% dot.

FLOAT

Set this keyword to write the pixel components as floating-point entities (the default is 8-bit).

GEOTIFF

Set this keyword to an anonymous structure containing one field for each of the GeoTIFF tags and keys to be written into the file. The GeoTIFF structure is formed using fields named from the following table.

Anonymous Structure Field Name	IDL Datatype
TAGS:	
"MODELPIXELSCALETAG"	DOUBLE[3]
"MODELTRANSFORMATIONTAG"	DOUBLE[4,4]
"MODELTIPOINTTAG"	DOUBLE[6,*]
KEYS:	
"GTMODELTYPEGEOKEY"	INT
"GTRASTERTYPEGEOKEY"	INT
"GTCITATIONGEOKEY"	STRING
"GEOGRAPHICTYPEGEOKEY"	INT
"GEOGCITATIONGEOKEY"	STRING
"GEOGGEODETICDATUMGEOKEY"	INT
"GEOGPRIMEMERIDIANGEOKEY"	INT
"GEOGLINEARUNITSGEOKEY"	INT
"GEOGLINEARUNITSIZEGEOKEY"	DOUBLE
"GEOGANGULARUNITSGEOKEY"	INT
"GEOGANGULARUNITSIZEGEOKEY"	DOUBLE
"GEOGELLIPSOIDGEOKEY"	INT
"GEOGSEMIMAJORAXISGEOKEY"	DOUBLE
"GEOGSEMIMINORAXISGEOKEY"	DOUBLE
"GEOGINVFLATTENINGGEOKEY"	DOUBLE
"GEOGAZIMUTHUNITSGEOKEY"	INT
"GEOGPRIMEMERIDIANLONGGEOKEY"	DOUBLE
"PROJECTEDCSTYPEGEOKEY"	INT

Table 3-140: GEOTIFF Structures

Anonymous Structure Field Name	IDLDatatype
"PCSCITATIONGEOKEY"	STRING
"PROJECTIONGEOKEY"	INT
"PROJCOORDTRANSGEOKEY"	INT
"PROJLINEARUNITSGEOKEY"	INT
"PROJLINEARUNITSSIZEGEOKEY"	DOUBLE
"PROJSTDPARALLEL1GEOKEY"	DOUBLE
"PROJSTDPARALLEL2GEOKEY"	DOUBLE
"PROJNATORIGINLONGGEOKEY"	DOUBLE
"PROJNATORIGINLATGEOKEY"	DOUBLE
"PROJFALSEEASTINGGEOKEY"	DOUBLE
"PROJFALSENORTHINGGEOKEY"	DOUBLE
"PROJFALSEORIGINLONGGEOKEY"	DOUBLE
"PROJFALSEORIGINLATGEOKEY"	DOUBLE
"PROJFALSEORIGINEASTINGGEOKEY"	DOUBLE
"PROJFALSEORIGINNORTHINGGEOKEY"	DOUBLE
"PROJCENTERLONGGEOKEY"	DOUBLE
"PROJCENTERLATGEOKEY"	DOUBLE
"PROJCENTEREASTINGGEOKEY"	DOUBLE
"PROJCENTERNORTHINGGEOKEY"	DOUBLE
"PROJSCALEATNATORIGINGEOKEY"	DOUBLE
"PROJSCALEATCENTERGEOKEY"	DOUBLE
"PROJAZIMUTHANGLEGEOKEY"	DOUBLE
"PROJSTRAIGHTVERTPOLELONGGEOKEY"	DOUBLE
"VERTICALCSTYPEGEOKEY"	INT

Table 3-140: GEOTIFF Structures (Continued)

Anonymous Structure Field Name	IDLDatatype
"VERTICALCITATIONGEOKEY"	STRING
"VERTICALDATUMGEOKEY"	INT
"VERTICALUNITSGEOKEY"	INT

Table 3-140: GEOTIFF Structures (Continued)

Note

If a GeoTIFF key appears multiple times in a file, only the value for the first instance of the key is returned.

ICC_PROFILE

Set this keyword to a byte array containing the ICC_PROFILE tag, as returned by the READ_TIFF routine.

Note

The ICC_PROFILE tag is an opaque byte array. It is assumed that the tag will be read from an existing file, and written out to a new file without modification.

LONG

Set this keyword to write the pixel components as unsigned 32-bit entities (the default is 8-bit).

ORIENTATION

Set this keyword to an integer value to specify the orientation of the TIFF image. The default is ORIENTATION=1.

Note

For COMPRESSION=3 (JPEG), all images are assumed to be in top-to-bottom order, and this keyword should not be specified.

Possible values are:

Value	Description
0	Column 0 represents the left-hand side, and row 0 represents the bottom (same as 4)
1	Column 0 represents the left-hand side, and row 0 represents the top.
2	Column 0 represents the right-hand side, and row 0 represents the top.
3	Column 0 represents the right-hand side, and row 0 represents the bottom.
4	Column 0 represents the left-hand side, and row 0 represents the bottom (same as 0)
5	Column 0 represents the top, and row 0 represents the left-hand side.
6	Column 0 represents the top, and row 0 represents the right-hand side.
7	Column 0 represents the bottom, and row 0 represents the right-hand side.
8	Column 0 represents the bottom, and row 0 represents the left-hand side.

Table 3-141: ORIENTATION Keyword Values

Warning

Not all TIFF readers honor the value of the ORIENTATION field. IDL writes the value into the file, but some readers are known to ignore this value. In such cases, we recommend that you convert the image to top-to-bottom order with the REVERSE function and then set ORIENTATION to 1.

PHOTOSHOP

Set this keyword to a byte array containing the TIFF PHOTOSHOP tag value, as returned by the READ_TIFF routine.

Note

The PHOTOSHOP tag is an opaque byte array. It is assumed that the tag will be read from an existing file, and written out to a new file without modification.

PLANARCONFIG

This keyword determines the order in which a multi-channel image is stored and written. It has no effect with a single-channel image. Set this keyword to 2 to if the Image parameter is interleaved by “plane”, or band, and its dimensions are (*Columns*, *Rows*, *Channels*). The default value is 1, indicating that multi-channel images are interleaved by color, also called channel, and its dimensions are (*Channels*, *Columns*, *Rows*).

As a special case, this keyword may be set to 2 to write an RGB image that is contained in three separate arrays (color planes), stored in the variables specified by the RED, GREEN, and BLUE keywords. Otherwise, omit this parameter (or set it to 1).

Note

Many TIFF readers can read only one- or three-channel images.

RED, GREEN, BLUE

If you are writing a Palette color image, set these keywords equal to the color table vectors, scaled from 0 to 255.

If you are writing an RGB interleaved image (i.e., if the PLANARCONFIG keyword is set to 2), set these keywords to the names of the variables containing the three image components.

SHORT

Set this keyword to write the pixel components as unsigned 16-bit entities (the default is 8-bit).

UNITS

Set this keyword to one of the following values to specify the units used for the values specified by the XRESOL and YRESOL keywords:

Value	Description
1	No units
2	Inches (the default)
3	Centimeters

Table 3-142: UNITS Keyword Values

VERBOSE

Set this keyword to produce additional diagnostic output during the write.

XPOSITION

Set this keyword to the X offset relative to the left side of the image, in units specified by the UNITS keyword. XPOSITION must be nonnegative. If XPOSITION is not specified then this tag is not written to the file.

Note

The XPOSITION tag is usually ignored by most TIFF readers.

XRESOL

Set this keyword to the horizontal resolution, in pixels per unit, where the unit is specified by the value of the UNITS keyword (inches, by default). The default value of XRESOL is 100. Note that while this value is stored in the TIFF file, it may be interpreted by the TIFF reader in a variety of ways, or ignored.

YPOSITION

Set this keyword to the Y offset relative to the top of the image (with positive Y increasing downwards), in units specified by the UNITS keyword. YPOSITION must be nonnegative. If YPOSITION is not specified then this tag is not written to the file.

Note

The YPOSITION tag is usually ignored by most TIFF readers.

YRESOL

Set this keyword to the vertical resolution, in pixels per unit, where the unit is specified by the value of the UNITS keyword (inches, by default). The default value of YRESOL is 100. Note that while this value is stored in the TIFF file, it may be interpreted by the TIFF reader in a variety of ways, or ignored.

Examples

Example 1

Create a pseudo screen dump from the current window. Note that this works only on a PseudoColor (8-bit) display:

```
WRITE_TIFF, 'test.tiff', TVRD()
```

Example 2

Write a three-channel image from three one-channel (two-dimensional) arrays, contained in the variables Red, Green, and Blue:

```
WRITE_TIFF, 'test.tif', Red, Green, Blue, PLANARCONFIG=2
```

Example 3

Write and read a multi-image TIFF file. The first image is a 16-bit single channel image stored using compression. The second image is an RGB image stored using 32-bits/channel uncompressed.

```
; Write the image data:
data = FIX(DIST(256))
rgbdata = LONARR(3,320,240)
WRITE_TIFF,'multi.tif',data,COMPRESSION=1,/SHORT
WRITE_TIFF,'multi.tif',rgbdata,/LONG,/APPEND
; Read the image data back
ok = QUERY_TIFF('multi.tif',s)
IF (ok) THEN BEGIN
  FOR i=0,s.NUM_IMAGES-1 DO BEGIN
    imp = QUERY_TIFF('multi.tif',t,IMAGE_INDEX=i)
    img = READ_TIFF('multi.tif',IMAGE_INDEX=i)
    HELP,t,/STRUCTURE
    HELP,img
  ENDFOR
ENDIF
```

Version History

Introduced: 5.0

ITIFF support added: 5.6

CMYK, DESCRIPTION, DOCUMENT_NAME, DOT_RANGE, ICC_PROFILE, PHOTOSHOP, XPOSITION and YPOSITION keywords added: 6.1

See Also

[READ_TIFF](#), [QUERY_*](#) Routines

WRITE_WAV

The WRITE_WAV procedure writes the audio stream to the named .WAV file.

Syntax

WRITE_WAV, *Filename*, *Data*, *Rate*

Arguments

Filename

A scalar string containing the full pathname of the .WAV file to write.

Data

The array to write into the new .WAV file. The array can be a one- or two-dimensional array. A two-dimensional array is written as a multi-channel audio stream where the leading dimension of the IDL array is the number of channels. If the input array is in BYTE format, the data is written as 8-bit samples, otherwise, the data is written as signed 16-bit samples.

Rate

The sampling rate for the data array in samples per second.

Keywords

None.

Version History

Introduced: 5.3

WRITE_WAVE

The `WRITE_WAVE` procedure writes a three dimensional IDL array to a `.wave` or `.bwave` file for use with the Wavefront Advanced Data Visualizer. Note that this routine only writes one scalar field for each Wavefront file that it creates.

This routine is written in the IDL language. Its source code can be found in the file `write_wave.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
WRITE_WAVE, File, Array [, /BIN] [, DATANAME=string]  
          [, MESHNAME=string] [, /NOMESHDEF] [, /VECTOR]
```

Arguments

File

A scalar string containing the full path name of the Wavefront file to write.

Array

A 3D array to be written to the file.

Keywords

BIN

Set this keyword to create a binary file. By default, text files are created.

DATANAME

Set this keyword to the name of the data inside of the Wavefront file. If not specified, the name used is “`idldata`”.

MESHNAME

Set this keyword to the name of the mesh used in the Wavefront file. If not specified, the name used is “`idlmesh`”.

NOMESHDEF

Set this keyword to *omit* the mesh definition from the Wavefront file.

VECTOR

Set this keyword to write the variable as a vector. The data is written as an array of 3-space vectors. The array may contain any number of dimensions but must have a leading dimension of 3. If the leading array dimension is not 3, this keyword is ignored.

Version History

Introduced: Pre 4.0

See Also

[READ_WAVE](#)

WRITEU

The WRITEU procedure writes unformatted binary data from an expression into a file. This procedure performs a direct transfer with no processing of any kind being done to the data.

Syntax

```
WRITEU, Unit, Expr1 ..., Exprn [, TRANSFER_COUNT=variable]
```

Arguments

Unit

The IDL file unit to which the output is sent.

Expr_{*j*}

The expressions to be output. For non-string variables, the number of bytes implied by the data type is output. When WRITEU is used with a variable of type string, IDL outputs exactly the number of bytes contained in the existing string.

Keywords

TRANSFER_COUNT

Set this keyword to a named variable in which to return the number of elements transferred by the output operation. Note that the number of elements is not the same as the number of bytes (except in the case where the data type being transferred is bytes). For example, transferring 256 floating-point numbers yields a transfer count of 256, not 1024 (the number of bytes transferred).

This keyword is useful with files opened with the RAWIO keyword to the OPEN routines. Normally, writing more data than an output device will accept causes an error. Files opened with the RAWIO keyword will not generate such an error. Instead, the programmer must keep track of the transfer count to judge the success or failure of a WRITEU operation.

Obsolete Keywords

The following keywords are obsolete:

- REWRITE

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Examples

```
; Create some data to store in a file:
D = BYTSCL(DIST(200))
; Open a new file for writing as IDL file unit number 1:
OPENW, 1, 'newfile'
; Write the data in D to the file:
WRITEU, 1, D
; Close file unit 1:
CLOSE, 1
```

Version History

Introduced: Original

See Also

[OPEN](#), [READU](#), [Chapter 11, “Files and Input/Output”](#) in the *Building IDL Applications* manual, and [“Unformatted Input/Output with Structures”](#) in Chapter 9 of the *Building IDL Applications* manual.

WSET

The WSET procedure selects the current window. Most IDL graphics routines do not explicitly require the IDL window to be specified. Instead, they use the window known as the current window. The window index number of the current window is contained in the read-only system variable !D.WINDOW. WSET only works with devices that have windowing systems.

Syntax

```
WSET [, Window_Index]
```

Arguments

Window_Index

This argument specifies the window index of the window to be made current. If this argument is not specified, a default of 0 is used.

If you set *Window_Index* equal to -1, IDL will try to locate an existing window to make current, ignoring any managed draw widgets that may exist. If there is no window to make current, WSET changes the value of the WINDOW field of the !D system variable to -1, indicating that there are no current windows.

If there are no existing IDL windows, and you call WSET without the *Window_Index* argument or with a *Window_Index* of 0, a new window with the index 0 is opened. Calling WSET with a *Window_Index* for a window that does not exist, except for window 0, results in a “Window is closed and unavailable” error message.

Keywords

None.

Examples

Create IDL windows 1 and 2 by entering:

```
WINDOW, 1 & WINDOW, 2
```

Set the current window to window 1 and display an image by entering:

```
WSET, 1 & TVSCL, DIST(100)
```

Set the current window to window 2 and display an image by entering:

```
WSET, 2 & TVSCL, DIST(100)
```

Version History

Introduced: Original

See Also

[WDELETE](#), [WINDOW](#), [WSHOW](#)

WSHOW

The WSHOW procedure exposes or hides the designated window.

Syntax

```
WSHOW [, Window_Index [, Show]] [, /ICONIC]
```

Arguments

Window_Index

The window index of the window to be hidden or exposed. If this argument is not specified, the current window is assumed. If this index is the window ID of a draw widget, the widget base associated with that drawable is brought to the front of the screen.

Show

Set *Show* to 0 to hide the window. Omit this argument or set it to 1 to expose the window.

Keywords

ICONIC

Set this keyword to iconify the window. Set **ICONIC** to 0 to de-iconify the window.

Under windowing systems, iconification is the task of the window manager, and client applications such as IDL have no direct control over it. The **ICONIC** keyword serves as a hint to the window manager, which is free to iconify the window or ignore the request as it sees fit.

Examples

To bring IDL window number 0 to the front, enter:

```
WSHOW, 0
```

Version History

Introduced: Original

See Also

[WDELETE](#), [WINDOW](#), [WSET](#)

WTN

The WTN function returns a multi-dimensional discrete wavelet transform of the input array *A*. The transform is based on a Daubechies wavelet filter.

WTN is based on the routine `wtn` described in section 13.10 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

```
Result = WTN( A, Coef [, /COLUMN] [, /DOUBLE] [, /INVERSE]
             [, /OVERWRITE] )
```

Return Value

Returns an output array of the same dimensions as *A*, containing the discrete wavelet transform over each dimension.

Arguments

A

The input vector or array. The dimensions of *A* must all be powers of 2.

Coef

An integer that specifies the number of wavelet filter coefficients. The allowed values are 4, 12, or 20. When *Coef* is 4, the `daub4()` function (see *Numerical Recipes*, section 13.10) is used. When *Coef* is 12 or 20, `pwt()` is called, preceded by `pwtset()` (see *Numerical Recipes*, section 13.10).

Keywords

COLUMN

Set this keyword if the input array *A* is in column-major format (composed of column vectors) rather than in row-major format (composed of row vectors).

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

INVERSE

If the INVERSE keyword is set, the inverse transform is computed. By default, WTN performs the forward wavelet transform.

OVERWRITE

Set the OVERWRITE keyword to perform the transform “in place.” The result overwrites the original contents of the array.

Examples

This example demonstrates the use of IDL’s discrete wavelet transform and sparse array storage format to compress and store an 8-bit gray-scale digital image. First, an image selected from the `people.dat` data file is transformed into its wavelet representation and written to a separate data file using the `WRITEU` procedure.

Next, the transformed image is converted, using the `SPRSIN` function, to row-indexed sparse storage format retaining only elements with an absolute magnitude greater than or equal to a specified threshold. The sparse image is written to a data file using the `WRITE_SPR` procedure.

Finally, the transformed image is reconstructed from the storage file and displayed alongside the original.

```

; Begin by choosing the number of wavelet coefficients to use and a
; threshold value:
coeffs = 12 & thres = 10.0

; Open the people.dat data file, read an image using associated
; variables, and close the file:
OPENR, 1, FILEPATH('people.dat', SUBDIR = ['examples', 'data'])
images = assoc(1, bytarr(192, 192))
image_1 = images[0]
close, 1

; Expand the image to the nearest power of two using cubic
; convolution, and transform the image into its wavelet
; representation using the WTN function:
pwr = 256
image_1 = CONGRID(image_1, pwr, pwr, /CUBIC)
wtn_image = WTN(image_1, coeffs)

; Write the image to a file using the WRITEU procedure and check
; the size of the file (in bytes) using the FSTAT function:
OPENW, 1, 'original.dat'
WRITEU, 1, wtn_image

```

```

status = FSTAT(1)
CLOSE, 1
PRINT, 'Size of the file is ', status.size, ' bytes.'

; Now, we convert the wavelet representation of the image to a
; row-indexed sparse storage format using the SPRSIN function,
; write the data to a file using the WRITE_SPR procedure, and check
; the size of the "compressed" file:
sprs_image = SPRSIN(wtn_image, THRES = thres)
WRITE_SPR, sprs_image, 'sparse.dat'
OPENR, 1, 'sparse.dat'
status = FSTAT(1)
CLOSE, 1
PRINT, 'Size of the compressed file is ', status.size, ' bytes.'

; Determine the number of elements (as a percentage of total
; elements) whose absolute magnitude is less than the specified
; threshold. These elements are not retained in the row-indexed
; sparse storage format:
PRINT, 'Percentage of elements under threshold: ', $
      100.*N_ELEMENTS(WHERE(ABS(wtn_image) LT thres, $
      count)) / N_ELEMENTS(image_1)

; Next, read the row-indexed sparse data back from the file
; sparse.dat using the READ_SPR function and reconstruct the
; image from the non-zero data using the FULSTR function:
sprs_image = READ_SPR('sparse.dat')
wtn_image = FULSTR(sprs_image)

; Apply the inverse wavelet transform to the image:
image_2 = WTN(wtn_image, COEFFS, /INVERSE)

; Calculate and print the amount of data used in reconstruction of
; the image:
PRINT, 'The image on the right is reconstructed from:', $
      100.0 - (100.* count/N_ELEMENTS(image_1)), $
      '% of original image data.'

; Finally, display the original and reconstructed images side by
; side:
WINDOW, 1, XSIZE = pwr*2, YSIZE = pwr, $
      TITLE = 'Wavelet Image Compression and File I/O'
TV, image_1, 0, 0
TV, image_2, pwr - 1, 0

```

IDL Output

```

Size of the file is    262144 bytes.
Size of the compressed file is    69600 bytes.

```

```
Percentage of elements under threshold: 87.0331
The image on the right is reconstructed from: 12.9669% of original
image data.
```

The sparse array contains only 13% of the elements contained in the original array. The following figure is created from this example. The image on the left is the original 256 by 256 image. The image on the right was compressed by the above process and was reconstructed from 13% of the original data. The size of the compressed image's data file is 26.6% of the size of the original image's data file. Note that due to limitations in the printing process, differences between the images may not be as evident as they would be on a high-resolution printer or monitor.



Figure 3-67: Original image (left) and image reconstructed from 13% of the data (right).

Version History

Introduced: 4.0

See Also

[FFT](#)

WV_* Routines

For information, see the [Chapter 1, “Introduction to the IDL Wavelet Toolkit”](#) in the *IDL Wavelet Toolkit* manual.

XBM_EDIT

The XBM_EDIT procedure is a utility for creating and editing icons for use with IDL widgets as bitmap labels for widget buttons.

The icons created with XBM_EDIT can be saved in two different file formats. IDL “array definition files” are text files that can be inserted into IDL programs. “Bitmap array files” are data files that can be read into IDL programs. Bitmap array files should be used temporarily until the final icon design is determined and then they can be saved as IDL array definitions for inclusion in the final widget code. This routine does not check the file types of the files being read and assumes that they are of the correct size and type for reading. XBM_EDIT maintains its state in a common block so it is restricted to one working copy at a time.

This routine is written in the IDL language. Its source code can be found in the file `xbm_edit.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XBM_EDIT [, /BLOCK] [, FILENAME=string] [, GROUP=widget_id]
          [, XSIZE=pixels] [, YSIZE=pixels]
```

Arguments

None.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XBM_EDIT block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

FILENAME

Set this keyword to a scalar string that contains the filename to be used for the new icon. If this argument is not specified, the name “idl.bm” is used. The filename can be changed in XBM_EDIT by editing the “Filename” field before selecting a file option.

GROUP

The widget ID of the widget that calls XBM_EDIT. When this ID is specified, the death of the caller results in the death of XBM_EDIT.

XSIZE

The number of pixels across the bitmap is in the horizontal direction. The default value is 16 pixels.

YSIZE

The number of pixels across the bitmap is in the vertical direction. The default value is 16 pixels.

Version History

Introduced: Pre 4.0

See Also

[WIDGET_BUTTON](#)

XDISPLAYFILE

The XDISPLAYFILE procedure is a utility for displaying ASCII text files using a widget interface.

This routine is written in the IDL language. Its source code can be found in the file `xdisplayfile.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XDISPLAYFILE, Filename [, /BLOCK] [, DONE_BUTTON=string]  
  [, /EDITABLE] [, FONT=string] [, GROUP=widget_id] [, HEIGHT=lines]  
  [, /MODAL] [, TEXT=string or string array] [, TITLE=string]  
  [, WIDTH=characters] [, WTEXT=variable]
```

Arguments

Filename

A scalar string that contains the filename of the file to display. *Filename* can include a path to that file.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XDISPLAYFILE block, any earlier calls to XMANAGER must have been called with the [NO_BLOCK](#) keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

DONE_BUTTON

Set this keyword to a string containing the text to use for the Done button label. If omitted, the text “Done with <filename>” is used.

EDITABLE

Set this keyword to allow modifications to the text displayed in XDISPLAYFILE. Setting this keyword also adds a “Save” button in addition to the Done button.

FONT

A string containing the name of the font to use. The font specified is a device font (an X Windows font on Motif systems; a TrueType or PostScript font on Windows systems). See “[About Device Fonts](#)” on page 4424 for details on specifying names for device fonts. If this keyword is omitted, the default font is used.

GROUP

The widget ID of the widget that calls XDISPLAYFILE. If this keyword is specified, the death of the group leader results in the death of XDISPLAYFILE.

HEIGHT

The number of text lines that the widget should display at one time. If this keyword is not specified, 24 lines is the default.

MODAL

Set this keyword to create the XDISPLAYFILE dialog as a modal dialog. Setting the MODAL keyword allows you to call XDISPLAYFILE from another modal dialog.

TEXT

A string or string array to be displayed in the widget instead of the contents of a file. If this keyword is present, the *Filename* input argument is ignored (but is still required). String arrays are displayed one element per line.

TITLE

A string to use as the widget title rather than the file name or “XDisplayFile”.

WIDTH

The width of the widget display in characters. If this keyword is not specified, 80 characters is the default.

WTEXT

Set this keyword to a named variable that will contain the widget ID of the text widget. This allows setting text selections and cursor positions programmatically. For example, the following code opens the XDISPLAYFILE widget and selects the first 10 characters of the file displayed in the text widget:

```
XDISPLAYFILE, 'myfile.txt', /EDITABLE, WTEXT=w  
WIDGET_CONTROL, w, SET_TEXT_SELECT=[0, 10]
```

Version History

Introduced: Pre 4.0

See Also

[PRINT/PRINTF](#), [XYOUTS](#)

XDXF

The XDXF procedure is a utility for displaying and interactively manipulating DXF objects.

Using XDXF

XDXF displays a resizable top-level base with a menu and draw widget used to display and manipulate the orientation of a DXF object.

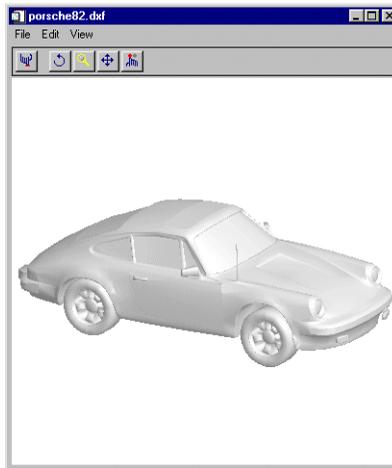


Figure 3-68: The XDXF Utility

XDXF also displays a dialog that contains block and layer information and allows the user to turn on and off the display of individual layers.

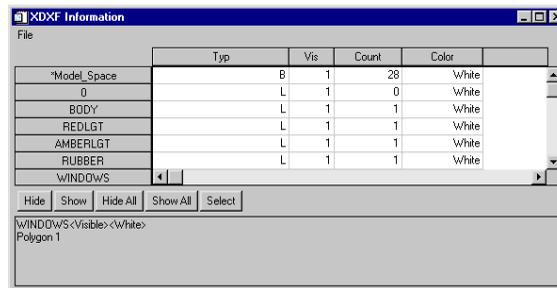


Figure 3-69: The XDXF Information Dialog

The XDXF Toolbar

The XDXF toolbar contains the following buttons:



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the object and drag to rotate.



Pan: Click the left mouse button on the object and drag to pan.



Zoom: Click the left mouse button on the object and drag to zoom in or out.



Select: Click on the object. The name of the selected object is displayed, if the object has a name, otherwise its class is displayed.

The XDXF Information Dialog

The XDXF Information dialog displays information about the blocks and layers contained in the currently displayed object, and allows you to turn on and off the display of each layer.

To show or hide layers in the DXF object, select the layer from the list of layers on the left of the dialog, and click the **Show** or **Hide** button. Alternatively, you can click in the “Vis” field for the desired layer. To show or hide all layers, click the **Show All** or **Hide All** buttons.

If the **View Block Outlines** checkbox is selected, a green box will be displayed around the currently selected block, if any.

Syntax

```
XDXF [, Filename] [, /BLOCK] [, GROUP=widget_id] [, SCALE=value] [, /TEST]
      [keywords to XOBJVIEW]
```

Arguments

Filename

A string specifying the name of the DXF file to display. If this argument is not specified, a file selection dialog is opened.

Keywords

XDXF accepts the keywords to [XOBJVIEW](#). (Note, however, that specifying the XOBJVIEW keyword MODAL is interpreted by XDXF as a call to the BLOCK keyword.) In addition, XDXF supports the following keywords:

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XDXF block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

The widget ID of the widget that calls XDXF. When this ID is specified, the death of the caller results in the death of XDXF.

SCALE

Set this keyword to the zoom factor for the initial view. The default is $1/\text{SQRT}(3)$. This default value provides the largest possible view of the object, while ensuring that no portion of the object will be clipped by the XDXF window, regardless of the object's orientation.

TEST

If this keyword is set, the file `heart.dxf` in the IDL distribution is automatically opened in XDXF.

Examples

Display the file `heart.dxf`, contained in the IDL distribution:

```
XDXF, FILEPATH('heart.dxf', $
    SUBDIR=['examples', 'data'])
```

Version History

Introduced: 5.4

See Also

[IDLffDXF](#)

XFONT

The XFONT function is a utility that creates a modal widget for selecting and viewing an X Windows font.

Calling XFONT resets the current X Windows font.

Note

This routine is only available on UNIX platforms.

This routine is written in the IDL language. Its source code can be found in the file `xfont.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
Result = XFONT( [, GROUP=widget_id] [, /PRESERVE_FONT_INFO] )
```

Return Value

The function returns a string containing the name of the last selected font. If no font is selected, or the “Cancel” button is clicked, a null string is returned.

Arguments

None.

Keywords

GROUP

The widget ID of the widget that calls XFONT. When this ID is specified, the death of the caller results in the death of XFONT.

PRESERVE_FONT_INFO

Set this keyword to make XFONT save the server font directory in common blocks so that subsequent calls to XFONT start-up much faster. If this keyword is not set, the common block is cleaned.

Version History

Introduced: Pre 4.0

See Also

[EFONT](#), [SHOWFONT](#)

XINTERANIMATE

The XINTERANIMATE procedure is a utility for displaying an animated sequence of images using off-screen pixmaps or memory buffers. The speed and direction of the display can be adjusted using the widget interface.

MPEG animation files can be created either programmatically using keywords to open and save a file, or interactively using the widget interface. Note that the MPEG standard does not allow movies with odd numbers of pixels to be created.

Note

MPEG support in IDL requires a special license. For more information, contact your RSI sales representative or technical support.

Note

Only a single copy of XINTERANIMATE can run at a time. If you need to run multiple instances of the animation widget concurrently, use the CW_ANIMATE compound widget.

This routine is written in the IDL language. Its source code can be found in the file `xinteranimate.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Using XINTERANIMATE

Displaying an animated series of images using XINTERANIMATE requires at least three calls to the routine: one to initialize the animation widget, one to load images, and one to display the images. When initialized using the SET keyword, XINTERANIMATE creates an approximately square pixmap or memory buffer, large enough to contain the requested number of frames of the requested size. Images are loaded using the IMAGE and FRAME keywords. Finally, images are displayed by copying them from the pixmap or memory buffer to the visible draw widget.

See [CW_ANIMATE](#) for a description of the widget interface controls used by XINTERANIMATE.

Syntax

XINTERANIMATE [, *Rate*]

Keywords for initialization: [, **SET**=[*sizex*, *sizey*, *nframes*]] [, **/BLOCK**]
 [, **/CYCLE**] [, **GROUP**=*widget_id*] [, **/MODAL**] [, **MPEG_BITRATE**=*value*]
 [, **MPEG_IFRAME_GAP**=*integer value*]
 [, **MPEG_MOTION_VEC_LENGTH**={1 | 2 | 3}]
 [, **/MPEG_OPEN**, **MPEG_FILENAME**=*string*]
 [, **MPEG_QUALITY**=*value*{0 to 100}] [, **/SHOWLOAD**] [, **/TRACK**]
 [, **TITLE**=*string*]

Keywords for loading images: [, **FRAME**=*value*{0 to (*nframes*-1)}
 [, **IMAGE**=*value*]] [, **/ORDER**] [, **WINDOW**=[*window_num* [, *x0*, *y0*, *sx*, *sy*]]]

Keywords for running animations: [, **/CLOSE**] [, **/KEEP_PIXMAPS**]
 [, **/MPEG_CLOSE**] [, **XOFFSET**=*pixels*] [, **YOFFSET**=*pixels*]

Arguments

Rate

A value between 0 and 100 that represents the speed of the animation as a percentage of the maximum display rate. The fastest animation is with a value of 100 and the slowest is with a value of 0. The default animation rate is 100. The animation must be initialized using the SET keyword before calling XINTERANIMATE with a rate value.

Keywords: Initialization

The following keywords are used to initialize the animation display. The SET keyword *must* be provided. Other keywords described in this section are optional; note that they work only when SET is specified.

SET

Set this keyword to a three-element vector [*Sizex*, *Sizey*, *Nframes*] to initialize XINTERANIMATE. *Sizex* and *Sizey* represent the width and height of the images to be displayed, in pixels. *Nframes* is the number of frames in the animation sequence. Note that *Nframes* must be at least 2 frames.

BLOCK

Set this keyword to have *XMANAGER block* when this application is registered. By default, **BLOCK** is set equal to zero, providing access to the command line if active command line processing is available. Note that setting **BLOCK=1** will cause *all* widget applications to block, not just this application. For more information, see the documentation for the **NO_BLOCK** keyword to *XMANAGER*.

Note

Only the outermost call to *XMANAGER* can block. Therefore, to have *XINTERANIMATE* block, any earlier calls to *XMANAGER* must have been called with the **NO_BLOCK** keyword. See the documentation for the **NO_BLOCK** keyword to *XMANAGER* for an example.

CYCLE

Normally, frames are displayed going either forward or backwards. If the **CYCLE** keyword is set, the animation reverses direction after the last frame in either direction is displayed.

GROUP

Set this keyword to the widget ID of the widget that calls *XINTERANIMATE*. When **GROUP** is specified, the death of the calling widget results in the death of *XINTERANIMATE*.

MODAL

Set this keyword to block processing of events from other widgets until the user quits *XINTERANIMATE*. A group leader must be specified (via the **GROUP** keyword) for the **MODAL** keyword to have any effect. By default, *XINTERANIMATE* does not block event processing.

MPEG_BITRATE

Set this keyword to a double-precision value to specify the MPEG movie bit rate. Higher bit rates will create higher quality MPEGs but will increase file size. The following table describes the valid values:

MPEG Version	Range
MPEG 1	0.1 to 104857200.0
MPEG 2	0.1 to 429496729200.0

Table 3-143: BITRATE Value Range

If you do not set this keyword, IDL computes the MPEG_BITRATE value based upon the value you have specified for the MPEG_QUALITY keyword.

Note

Only use the MPEG_BITRATE keyword if changing the MPEG_QUALITY keyword value does not produce the desired results. It is highly recommended to set the MPEG_BITRATE to at least several times the frame rate to avoid unusable MPEG files or file generation errors.

MPEG_FILENAME

Set this keyword equal to a string specifying the name of the MPEG file. If no file name is specified, the default value (`idl.mpg`) is used.

MPEG_IFRAME_GAP

Set this keyword to a positive integer value that specifies the number of frames between I frames to be created in the MPEG file. I frames are full-quality image frames that may have a number of predicted or interpolated frames between them.

If you do not specify this keyword, IDL computes the MPEG_IFRAME_GAP value based upon the value you have specified for the MPEG_QUALITY keyword.

Note

Only use the MPEG_IFRAME_GAP keyword if changing the MPEG_QUALITY keyword value does not produce the desired results.

MPEG_MOTION_VEC_LENGTH

Set this keyword to an integer value specifying the length of the motion vectors to be used to generate predictive frames. Valid values include:

- 1 = Small motion vectors.
- 2 = Medium motion vectors.
- 3 = Large motion vectors.

If you do not set this keyword, IDL computes the MPEG_MOTION_VEC_LENGTH value based upon the value you have specified for the MPEG_QUALITY keyword.

Note

Only use the MPEG_MOTION_VEC_LENGTH keyword if changing the MPEG_QUALITY value does not produce the desired results.

MPEG_OPEN

Set this keyword to open an MPEG file.

MPEG_QUALITY

Set this keyword to an integer value between 0 (low quality) and 100 (high quality) inclusive to specify the quality at which the MPEG stream is to be stored. Higher quality values result in lower rates of time compression and less motion prediction which provide higher quality MPEGs but with substantially larger file size. Lower quality factors may result in longer MPEG generation times. The default is 50.

Note

Since MPEG uses JPEG (lossy) compression, the original picture quality can't be reproduced even when setting QUALITY to its highest setting.

SHOWLOAD

Set this keyword to display each frame and update the frame slider as frames are loaded.

TRACK

Set this keyword to cause the frame slider to track the current frame when the animation is in progress. The default is not to track.

TITLE

Use this keyword to specify a string to be used as the title of the animation widget. If TITLE is not specified, the title is set to “XInterAnimate.”

Keywords: Loading Images

The following keywords are used to load images into the animation display. They have no effect when initializing or running animations.

FRAME

Use this keyword to specify the frame number when loading frames. FRAME must be set to a number in the range 0 to $Nframes-1$.

IMAGE

Use this keyword to specify a single image to be loaded at the animation position specified by the FRAME keyword. (FRAME *must* also be specified.)

ORDER

Set this keyword to display images from the top down instead of the default bottom up.

WINDOW

When this keyword is specified, an image is copied from an existing window to the animation pixmap or memory buffer. (When using some windowing systems, using this keyword is much faster than reading from the display and then calling XINTERANIMATE with a 2-D array.)

The value of this parameter is either an IDL window number (in which case the entire window is copied), or a vector containing the window index and the rectangular bounds of the area to be copied. For example:

```
WINDOW = [Window_Number, X0, Y0, Sx, Sy]
```

Keywords: Running Animations

The following keywords are used when running the animation. They have no effect when initializing the animation or loading images.

CLOSE

Set this keyword to delete the offscreen pixmaps or buffers and the animation widget itself. This also takes place automatically when the user presses the “Done With Animation” button or closes the window with the window manager.

KEEP_PIXMAPS

If this keyword is set, XINTERANIMATE will not destroy the animation pixmaps or buffers when it is killed. Calling XINTERANIMATE again without going through the SET and LOAD steps will play the same animation without the overhead of creating the pixmaps.

MPEG_CLOSE

Set this keyword to close and save the MPEG file. This keyword has no effect if MPEG_OPEN was not used during initialization.

XOFFSET

Use this keyword to specify the horizontal offset, in pixels from the left of the frame, of the image in the destination window.

YOFFSET

Use this keyword to specify the vertical offset, in pixels from the bottom of the frame, of the image in the destination window.

Examples

Enter the following commands to open the file ABNORM.DAT (a series of images of a human heart) and animate the images it contains using XINTERANIMATE.

```
OPENR, unit, FILEPATH('abnorm.dat', SUBDIR=['examples','data']), $
  /GET_LUN
H = BYTARR(64, 64, 16)
READU, unit, H
CLOSE, unit

; Read the images into variable H:
H = REBIN(H, 128, 128, 16)

; Initialize XINTERANIMATE:
XINTERANIMATE, SET=[128, 128, 16], /SHOWLOAD

; Load the images into XINTERANIMATE:
```

```
FOR I=0,15 DO XINTERANIMATE, FRAME = I, IMAGE = H[* ,*,I]  
  
; Play the animation:  
XINTERANIMATE, /KEEP_PIXMAPS
```

Note

Since the `KEEP_PIXMAPS` keyword was supplied, the same animation can be replayed (after the animation widget has been destroyed) with the single command `XINTERANIMATE`.

Version History

Introduced: Pre 4.0

See Also

[CW_ANIMATE](#)

XLOADCT

The XLOADCT procedure is a utility that provides a graphical widget interface to the LOADCT procedure. XLOADCT displays the current colortable and shows a list of available predefined color tables. Clicking on the name of a color table causes that color table to be loaded in true color decomposed visual. Many other options, such as Gamma correction, stretching, and transfer functions can also be applied to the colortable.

This routine is written in the IDL language. Its source code can be found in the file `xloadct.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XLOADCT [, /BLOCK] [, BOTTOM=value] [, FILE=string] [, GROUP=widget_id]
  [, /MODAL] [, NCOLORS=value] [, /SILENT]
  [, UPDATECALLBACK='procedure_name' [, UPDATECBDATA=value]]
  [, /USE_CURRENT]
```

Arguments

None.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XLOADCT block, any earlier calls to XMANAGER must have been called with the [NO_BLOCK](#) keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

BOTTOM

The first color index to use. XLOADCT will use color indices from BOTTOM to BOTTOM+NCOLORS-1. The default is BOTTOM=0.

FILE

Set this keyword to a string representing the name of the file to be used instead of the file `colors1.tbl` in the IDL directory.

GROUP

The widget ID of the widget that calls XLOADCT. When this ID is specified, a death of the caller results in a death of XLOADCT.

MODAL

Set this keyword to block processing of events from other widgets until the user quits XLOADCT. A group leader must be specified (via the GROUP keyword) for the MODAL keyword to have any effect. By default, XLOADCT does not block event processing.

NCOLORS

The number of colors to use. Use color indices from 0 to the smaller of !D.TABLE_SIZE-1 and NCOLORS-1. The default is all available colors (!D.TABLE_SIZE).

SILENT

Normally, no informational message is printed when a color map is loaded. If this keyword is set to zero, the message is printed.

UPDATECALLBACK

Set this keyword to a string containing the name of a user-supplied procedure that will be called when the color table is updated by XLOADCT. The procedure may optionally accept a keyword called DATA, which will be automatically set to the value specified by the optional UPDATECBDATA keyword.

UPDATECBDATA

Set this keyword to a value of any type. It will be passed via the DATA keyword to the user-supplied procedure specified via the UPDATECALLBACK keyword, if any. If

the UPDATECBDATA keyword is not set the value accepted by the DATA keyword to the procedure specified by UPDATECALLBACK will be undefined.

USE_CURRENT

Set this keyword to use the current color tables, regardless of the contents of the COLORS common block.

Version History

Introduced: Pre 4.0

See Also

[LOADCT](#), [XPALETTE](#), [TVLCT](#)

XMANAGER

The XMANAGER procedure provides the main event loop and management for widgets created using IDL. Calling XMANAGER “registers” a widget program with the XMANAGER event handler. XMANAGER takes control of event processing until all widgets have been destroyed.

Beginning with IDL version 5.0, IDL supports an *active command line* that allows the IDL command input line to continue accepting input while properly configured widget applications are running. See “[A Note About Blocking in XMANAGER](#)” on page 2585 for a more detailed explanation of the active command line.

This routine is written in the IDL language. Its source code can be found in the file `xmanager.pro` in the `lib` subdirectory of the IDL distribution.

Warning

Although this routine is written in the IDL language, it may change in the future in its internal implementation. For future upgradability, it is best not to modify or even worry about what this routine does internally.

Syntax

```
XMANAGER [, Name, ID] [, /CATCH] [, CLEANUP=string]  
      [, EVENT_HANDLER=procedure_name] [, GROUP_LEADER=widget_id]  
      [, /JUST_REG] [, /NO_BLOCK]
```

Arguments

Name

A string that contains the name of the routine that creates the widget (i.e., the name of the widget creation routine that is calling XMANAGER).

Note

The *Name* argument is stored in a COMMON block for use by the XREGISTERED routine. The stored name is case-sensitive.

ID

The widget ID of the top-level base that is the root of the widget hierarchy being to be managed.

Keywords

CATCH

Set this keyword to cause XMANAGER to catch any errors, using the **CATCH** procedure, when dispatching widget events. If the **CATCH** keyword is set equal to zero, execution halts and IDL provides traceback information when an error is detected. This keyword is set by default (errors are caught and processing continues).

Do not specify either the *Name* or *ID* argument to XMANAGER when specifying the **CATCH** keyword (they are ignored). **CATCH** turns error catching on and off for *all* applications managed by XMANAGER. When **CATCH** is specified, XMANAGER changes its error-catching behavior and returns immediately, without taking any other action.

Note

Beginning with IDL version 5.0, the default behavior of XMANAGER is to catch errors and continue processing events. In versions of IDL prior to version 5.0, XMANAGER halted when an error was detected. This change in default behavior was necessary in order to allow multiple widget applications (all being managed by XMANAGER) to coexist peacefully. When **CATCH** is set equal to zero, (the old behavior), any error halts XMANAGER, and thus halts event processing for all running widget applications.

Note also that **CATCH** is only effective if XMANAGER is blocking to dispatch errors. If event dispatching for an active IDL command line is in use, the **CATCH** keyword has no effect.

The **CATCH=0** setting (errors are not caught and processing halts in XMANAGER when an error is detected) is intended as a debugging aid. Finished programs should not set **CATCH=0**.

CLEANUP

Set this keyword to a string that contains the name of the routine to be called when the widget program dies. If this keyword is not specified, the routine (if any) specified for the program's top-level base by the **KILL_NOTIFY** keyword to **WIDGET_BASE** or **WIDGET_CONTROL** is used.

The routine specified by **CLEANUP** becomes the **KILL_NOTIFY** routine for the widget application, overriding any cleanup routines that have been set previously via the **KILL_NOTIFY** keyword to **WIDGET_BASE** or **WIDGET_CONTROL**.

Note

Specifying a routine for the widget application's top-level base via the `KILL_NOTIFY` keyword to `WIDGET_CONTROL` *after* the call to `XMANAGER` will override the value of the `CLEANUP` keyword.

The cleanup routine is called with the widget identifier as its only argument.

EVENT_HANDLER

Set this keyword to a string that contains the name of a routine to be called when a widget event occurs in the widget program being registered. If this keyword is not supplied, `XMANAGER` will construct a default name by adding the “_event” suffix to the *Name* argument. See the example below for a more detailed explanation.

GROUP_LEADER

The widget ID of the group leader for the widget being processed. When the leader dies either by the users actions or some other routine, all widgets that have that leader will also die.

For example, a widget that views a help file for a demo widget would have that demo widget as its leader. When the help widget is registered, it sets the keyword `GROUP_LEADER` to the widget ID of the demo widget. If the demo widget were destroyed, the help widget led by it would be killed by the `XMANAGER`.

JUST_REG

Set this keyword to indicate that `XMANAGER` should just register the widget and return immediately. This keyword is useful if you want to register a group of related top-level widgets before beginning event processing and one or more of the registered widgets requests that `XMANAGER` block event processing. (Note that in this case a later call to `XMANAGER` without the `JUST_REG` keyword is necessary to begin blocking.)

(See “[A Note About Blocking in XMANAGER](#)” on page 2585 for further discussion of the active command line.)

Warning

`JUST_REG` is not the same as `NO_BLOCK`. See “[JUST_REG vs. NO_BLOCK](#)” on page 2585 for additional details.

NO_BLOCK

Set this keyword to tell XMANAGER that the registering client does not require XMANAGER to block if active command line event processing is available. If active command line event processing is available *and* every current XMANAGER client specifies NO_BLOCK, then XMANAGER will not block and the user will have access to the command line while widget applications are running.

It is important to understand the result of making nested calls to XMANAGER. XMANAGER can only block event processing for one client at a time. In applications involving multiple calls to XMANAGER (either directly or via calls to other routines that call XMANAGER, such as XLOADCT), blocking occurs only for the outermost call to XMANAGER, unless XMANAGER is told not to block in that call. If an application contains two calls to XMANAGER, the second call cannot block unless the first call sets the NO_BLOCK keyword. If an application contains a call to XMANAGER, followed by a call to XLOADCT, XLOADCT will not block unless the NO_BLOCK keyword was set in the call to XMANAGER (and the BLOCK keyword to XLOADCT is set). Consider the following example:

```

PRO blocking_example_event, event
  ; The following call blocks only if the NO_BLOCK keyword to
  ; XMANAGER is set:
  XLOADCT, /BLOCK
END

PRO blocking_example
  base=WIDGET_BASE(/COLUMN)
  button1=WIDGET_BUTTON(base,VALUE='Run XLOADCT')
  WIDGET_CONTROL,base, /REALIZE
  XMANAGER,'blocking_example', base, /NO_BLOCK
END

```

If the NO_BLOCK keyword to XMANAGER was not set in the above example, XLOADCT would not block, even though the BLOCK keyword was set. Setting the NO_BLOCK keyword to XMANAGER prevents XMANAGER from blocking, thereby allowing the subsequent call to XMANAGER (via XLOADCT) to block.

Warning

NO_BLOCK is not the same as JUST_REG. See [“JUST_REG vs. NO_BLOCK”](#) on page 2585 for additional details.

Obsolete Keywords

The following keywords are obsolete:

- BACKGROUND
- MODAL

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

A Note About Blocking in XMANAGER

By default, IDL widget application blocking is *enabled*. Unless you take the appropriate steps, widget applications will block all other processing from occurring in IDL. Keeping the following issues in mind when writing widget applications will give you the best chance to create applications that coexist with other applications and the IDL command line.

Active Command Line

Beginning with IDL version 5.0, most versions of IDL’s command-processing front-end are able to support an *active command line* while running properly constructed widget applications. What this means is that—provided the widget application is properly configured—the IDL command input line is available for input while a widget application is running and widget events are being processed.

There are currently 3 separate IDL command-processing front-end implementations:

- Microsoft Windows IDLDE
- Motif IDLDE (UNIX)
- UNIX plain tty

Note that widget applications must be well-behaved with respect to blocking widget event processing. Since in most cases XMANAGER is used to handle widget event processing, this means that in order for the command line to remain active, all widget applications must be run with the NO_BLOCK keyword to XMANAGER set. (Note that since NO_BLOCK is *not* the default, it is quite likely that some application will block.) If a single application runs in blocking mode, the command line will be inaccessible until the blocking application exits. When a blocking application exits, the IDL command line will once again become active.

JUST_REG vs. NO_BLOCK

Although their names imply a similar function, the JUST_REG and NO_BLOCK keywords perform very different services. It is important to understand what they do and how they differ.

The JUST_REG keyword tells XMANAGER that it should simply register a client and then return immediately. The result is that the client becomes known to

XMANAGER, and that future calls to XMANAGER will take this client into account. Therefore, JUST_REG only controls how the registering call to XMANAGER should behave. The client can still be registered as requiring XMANAGER to block by setting NO_BLOCK=0. In this case, *future* calls to XMANAGER will block.

The NO_BLOCK keyword tells XMANAGER that the registered client does not require XMANAGER to block if the command-processing front-end is able to support active command line event processing. XMANAGER remembers this attribute of the client until the client exits, even after the call to XMANAGER that registered the client returns. NO_BLOCK is just a “vote” on how XMANAGER should behave—the final decision is made by XMANAGER by considering the NO_BLOCK attributes of *all* of its current clients as well as the ability of the command-processing front-end in use to support the active command line.

Blocking vs. Non-blocking Applications

The issue of blocking in XMANAGER requires some explanation. IDL widget events are not processed until the WIDGET_EVENT function is called to handle them. Otherwise, they are queued by IDL indefinitely. Knowing how and when to call WIDGET_EVENT is the primary service provided by XMANAGER.

There are two ways blocking is typically handled:

1. The first call to XMANAGER processes events by calling WIDGET_EVENT as necessary until no managed widgets remain on the screen. This is referred to as “blocking” because XMANAGER does not return to the caller until it is done, and the IDL command line is not available.
2. XMANAGER does not block, and instead, the part of IDL that reads command input also watches for widget events and calls WIDGET_EVENT as necessary while also reading command input. This is referred to as “non-blocking” or “active command line” mode.

XMANAGER will block unless the following conditions are met:

- All registered widget applications have the NO_BLOCK keyword to XMANAGER set.
- No modal dialogs are displayed. (Modal dialogs always block until dismissed.)

In general, we suggest that new widget applications be written with XMANAGER blocking disabled (that is, with the NO_BLOCK keyword set).

Since a widget application that does block event processing for itself will block event processing for all other widget applications (and the IDL command line) as well, we suggest that older widget applications be upgraded to take advantage of the new, non-

blocking behavior by adding the `NO_BLOCK` keyword to most calls to `XMANAGER`.

Examples

The following code creates a widget named `EXAMPLE` that is just a base widget with a “Done” button and registers it with the `XMANAGER`. Widgets being registered with the `XMANAGER` must provide at least two routines. The first routine creates the widget and registers it with the manager and the second routine processes the events that occur within that widget. An example widget is supplied below that uses only two routines. A number of other “Simple Widget Examples”, can be viewed by entering `WEXMASTER` at the IDL prompt. These simple programs demonstrate many aspects of widget programming.

The following lines of code would be saved in a single file, named `example.pro`:

```

; Begin the event handler routine for the EXAMPLE widget:
PRO example_event, ev

; The uservalue is retrieved from a widget when an event occurs:
WIDGET_CONTROL, ev.id, GET_UVALUE = uv

; If the event occurred in the Done button, kill the widget
; example:
if (uv eq 'DONE') THEN WIDGET_CONTROL, ev.top, /DESTROY

; End of the event handler part:
END

; This is the routine that creates the widget and registers it with
; the XMANAGER:
PRO example

; Create the top-level base for the widget:
base = WIDGET_BASE(TITLE='Example')

; Create the Done button and set its uservalue to "DONE":
done = WIDGET_BUTTON(base, VALUE = 'Done', UVALUE = 'DONE')
; Realize the widget (i.e., display it on screen):
WIDGET_CONTROL, base, /REALIZE

; Register the widget with the XMANAGER, leaving the IDL command
; line active:
XMANAGER, 'example', base, /NO_BLOCK

; End of the widget creation part:
END

```

First the event handler routine is listed. The handler routine has the same name as the main routine with the characters “_event” added. If you would like to use another event handler name, you would need to pass its name to XMANAGER using the `EVENT_HANDLER` keyword.

Notice that the event routine is listed before the main routine. If you include both the event routine and the main routine in a single `.pro` file with the name of the main routine (a common practice), this is necessary to ensure that the event routine is compiled. If the event routine were placed after the main routine in the file, IDL would compile and execute the main routine — and the compiler would exit — before the event routine was compiled. Alternatively, you can save your event routine in its own file with its own name (which in the above example would be `example_event.pro`), and IDL will compile the routine when it is required.

Notice also the `NO_BLOCK` keyword to XMANAGER has been included. This allows IDL to continue processing events and accepting input at the command prompt while the `example` widget application is running.

Version History

Introduced: Pre 4.0

See Also

[XMTOOL](#), [XREGISTERED](#), [Chapter 29, “Creating Widget Applications”](#) in the *Building IDL Applications* manual.

XMNG_TMPL

The `XMNG_TMPL` procedure is a template for widgets that use the `XMANAGER`. Use this template instead of writing your widget applications from “scratch”. This template can be found in the file `xmng_tmpl.pro` in the `lib` subdirectory of the IDL distribution.

The documentation header should be altered to reflect the actual implementation of the `XMNG_TMPL` widget. Use a global search and replace to replace the word `XMNG_TMPL` with the name of the routine you would like to use. All the comments with a “****” in front of them should be read, decided upon and removed from the final copy of your new widget routine.

Syntax

```
XMNG_TMPL [, /BLOCK] [, GROUP=widget_id]
```

Arguments

None.

Keywords

BLOCK

Set this keyword to have `XMANAGER` *block* when this application is registered. By default, `BLOCK` is set equal to zero, providing access to the command line if active command line processing is available. Note that setting `BLOCK=1` will cause *all* widget applications to block, not just this application. For more information, see the documentation for the `NO_BLOCK` keyword to `XMANAGER`.

Note

Only the outermost call to `XMANAGER` can block. Therefore, to have `XMNG_TMPL` block, any earlier calls to `XMANAGER` must have been called with the `NO_BLOCK` keyword. See the documentation for the `NO_BLOCK` keyword to `XMANAGER` for an example.

GROUP

The widget ID of the widget that calls `XMNG_TMPL`. When this ID is specified, the death of the caller results in the death of `XMNG_TMPL`.

Version History

Introduced: Pre 4.0

See Also

[CW_TMPL](#)

XMTOOL

The XMTOOL procedure displays a tool for viewing widgets currently being managed by the XMANAGER. Only one instance of the XMTOOL can run at one time.

This routine is written in the IDL language. Its source code can be found in the file `xmtool.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XMTOOL [, /BLOCK] [, GROUP=widget_id]
```

Arguments

None.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XMTOOL block, any earlier calls to XMANAGER must have been called with the [NO_BLOCK](#) keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

The widget ID of the widget that calls XMTOOL. If the calling widget is destroyed, the XMTOOL is also destroyed.

Version History

Introduced: Pre 4.0

See Also

[XMANAGER](#)

XOBJVIEW

The XOBJVIEW procedure is a utility used to quickly and easily view and manipulate IDL Object Graphics on screen. It displays given objects in an IDL widget with toolbar buttons and menus providing functionality for manipulating, printing, and exporting the resulting graphic. The mouse can be used to rotate, scale, or translate the overall model shown in a view, or to select graphic objects in a view.

This routine is written in the IDL language. Its source code can be found in the file `xobjview.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Using XOBJVIEW

XOBJVIEW displays a resizable top-level base with a menu, toolbar and draw widget, as shown in the following figure:

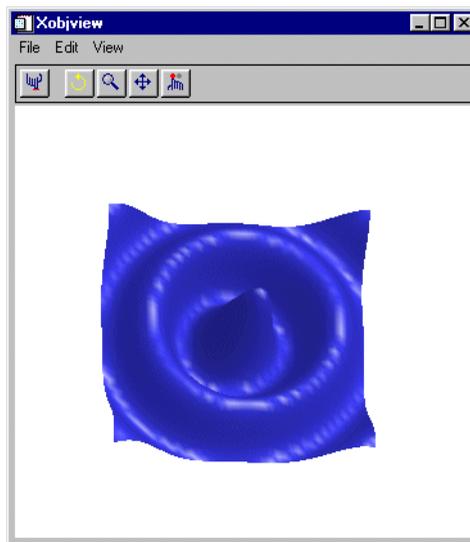


Figure 3-70: The XOBJVIEW widget

The XOBJVIEW Toolbar

The XOBJVIEW toolbar contains the following buttons:



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the object and drag to rotate.



Pan: Click the left mouse button on the object and drag to pan.



Zoom: Click the left mouse button on the object and drag to zoom in or out.



Select: Click on the object. The name of the selected object is displayed, if the object has a name, otherwise its class is displayed.

Syntax

```
XOBJVIEW, Obj [, BACKGROUND=[r, g, b]] [, /BLOCK] [, /DOUBLE_VIEW ]
  [, GROUP=widget_id] [, /JUST_REG] [, /MODAL] [, REFRESH=widget_id]
  [, RENDERER={0 | 1}] [, SCALE=value] [, STATIONARY=objref(s)] [, /TEST]
  [, TITLE=string] [, TLB=variable] [, XOFFSET=value] [, XSIZE=pixels]
  [, YOFFSET=value] [, YSIZE=pixels]
```

Arguments

Obj

A reference to an atomic graphics object, an IDLgrModel, or an array of such references. If *Obj* is an array, the array can contain a mixture of such references. Also, if *Obj* is an array, all object references in the array must be unique (i.e. no two references in the array can refer to the same object).

Obj is not destroyed when XOBJVIEW is quit or killed.

Keywords

BACKGROUND

Set this keyword to a three-element [*r*, *g*, *b*] color vector specifying the background color of the XOBJVIEW window.

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XOBJVIEW block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

DOUBLE_VIEW

Set this keyword to cause XOBJVIEW to set the DOUBLE property on the IDLgrView that it uses to display graphical data.

GROUP

The widget ID of the widget that calls XOBJVIEW. When this ID is specified, the death of the caller results in the death of XOBJVIEW.

JUST_REG

Set this keyword to indicate that the XOBJVIEW utility should just be registered and return immediately. This is useful if you want to register XOBJVIEW before beginning event processing and one or more widgets requests that XMANAGER block event processing. See [“JUST_REG vs. NO_BLOCK”](#) on page 2585 for additional details.

MODAL

Set this keyword to block processing of events from other widgets until the user quits XOBJVIEW. The MODAL keyword does not require a group leader to be specified. If no group leader is specified, and the MODAL keyword is set, XOBJVIEW fabricates an invisible group leader for you.

Note

To be modal, XOBJVIEW does not require that its caller specify a group leader. This is unlike other IDL widget procedures such as XLOADCT, which, to be modal, do require that their caller specify a group leader. These other procedures were implemented this way to encourage the caller to create a modal widget that

will be well-behaved with respect to layering and iconizing. (See [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 2255 for more information.)

To provide a simple means of invoking XOBJVIEW as a modal widget in applications that contain no other widgets, XOBJVIEW can be invoked as MODAL without specifying a group leader, in which case XOBJVIEW fabricates an invisible group leader for you. For applications that contain multiple widgets, however, it is good programming practice to supply an appropriate group leader when invoking XOBJVIEW, /MODAL. As with other IDL widget procedures with names prefixed with “X”, specify the group leader via the GROUP keyword.

REFRESH

Set this keyword to the widget ID of the XOBJVIEW instance to be refreshed. To retrieve the widget ID of an instance of XOBJVIEW, first call XOBJVIEW with the TLB keyword. To refresh that instance of XOBJVIEW, call XOBJVIEW again and set REFRESH to the value retrieved by the TLB keyword in the earlier call to XOBJVIEW. For example, in the initial call to XOBJVIEW, use the TLB keyword as follows:

```
XOBJVIEW, myobj, TLB=tlb
```

If the properties of myobj are changed in your application or at the IDL command line, refresh the view in XOBJVIEW by calling XOBJVIEW again with the REFRESH keyword:

```
XOBJVIEW, REFRESH=tlb
```

For an example application demonstrating the use of the REFRESH keyword, see [“Example 3”](#) on page 2600.

Note

Currently, the REFRESH keyword can only be used to refresh the object itself. All other keywords and arguments to XOBJVIEW are ignored when REFRESH is specified, therefore, properties such as the background color and scale are not affected.

RENDERER

Set this keyword to an integer value indicating which graphics renderer to use when drawing objects in the XOBJVIEW draw window. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL's software implementation

By default, your platform's native OpenGL implementation is used. If your platform does not have a native OpenGL implementation, IDL's software implementation is used regardless of the value of this property. See [“Hardware vs. Software Rendering”](#) in Chapter 34 of the *Using IDL* manual for details. Your choice of renderer may also affect the maximum size of the XOBJVIEW draw window. See [“IDLgrWindow”](#) on page 4135 for details.

SCALE

Set this keyword to the zoom factor for the initial view. The default is 1/SQRT(3). This default value provides the largest possible view of the object, while ensuring that no portion of the object will be clipped by the XOBJVIEW window, regardless of the object's orientation.

STATIONARY

Set this keyword to a reference to an atomic graphics object, an IDLgrModel, or an array of such references. If this keyword is an array, the array can contain a mixture of such references. Also, if this keyword is an array, all object references in the array must be unique (i.e., no two references in the array can refer to the same object). Objects passed to XOBJVIEW via this keyword will not scale, rotate, or translate in response to mouse events. Default stationary objects are two lights. These two lights are replaced if one or more lights are supplied via this keyword. Objects specified via this keyword are not destroyed by XOBJVIEW when XOBJVIEW is quit or killed.

For example, to change the default lights used by XOBJVIEW, you could specify your own lights using the STATIONARY keyword as follows:

```
mylight1 = OBJ_NEW('IDLgrLight', TYPE=0, $
    COLOR=[255,0,0]) ; Ambient red
mylight2 = OBJ_NEW('IDLgrLight', TYPE=2, $
    COLOR=[255,0,0], LOCATION=[2,2,5]) ; Directional red

mymodel = OBJ_NEW('IDLgrModel')
mymodel->Add, mylight1
mymodel->Add, mylight2

XOBJVIEW, /TEST, STATIONARY=mymodel
```

TEST

If set, the *Obj* argument is not required (and is ignored if provided). A blue sinusoidal surface is displayed. This allows you to test code that uses XOBJVIEW without having to create an object to display.

TITLE

Set this keyword to the string that appears in the XOBJVIEW title bar.

TLB

Set this keyword to a named variable that upon return will contain the widget ID of the top level base.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row-major or column-major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

XSIZE

Set this keyword to the width of the drawable area in pixels. The default is 400.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the upper left corner of the parent widget.

Specifying an offset relative to a row-major or column-major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

YSIZE

Set this keyword to the height of the drawable area in pixels. The default is 400.

Examples

Example 1

This example displays a simple IDLgrSurface object using XOBJVIEW:

```
oSurf = OBJ_NEW('IDLgrSURFACE', DIST(20))
XOBJVIEW, oSurf
```

Example 2

This example displays an IDLgrModel object consisting of two separate objects:

```
; Create contour object:
oCont = OBJ_NEW('IDLgrContour', DIST(20), N_LEVELS=10)

; Create surface object:
oSurf = OBJ_NEW('IDLgrSurface', $
    DIST(20), INDGEN(20)+20, INDGEN(20)+20)

; Create model object:
oModel = OBJ_NEW('IDLgrModel')

; Add contour and surface objects to model:
oModel->Add, oCont
oModel->Add, oSurf

; View model:
XOBJVIEW, oModel
```

This code results in the following view in the XOBJVIEW widget:

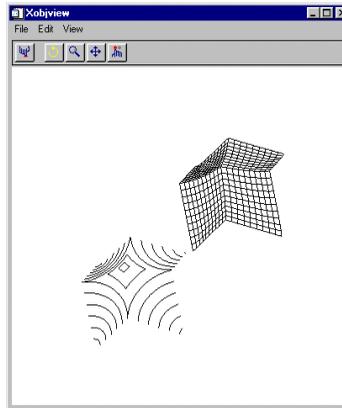


Figure 3-71: Using XOBJVIEW to view a model consisting of two objects

Note that when you click the Select button, and then click on an object, the class of that object appears next to the Select button. If the selected object has a non-null NAME property associated with it, that string value will be displayed, otherwise the name of the selected object's class will be displayed.

If you want the class of the model to appear when you click over any object in the model, you can set the SELECT_TARGET property of the model as follows:

```
oModel->SetProperty, /SELECT_TARGET
```

Also note that it is not necessary to create a model to view more than one object using XOBJVIEW. We could view the oCont and oSurf objects created in the above example by placing them in an array as follows:

```
XOBJVIEW, [oCont, oSurf]
```

Example 3

This example demonstrates how the REFRESH keyword can be used to refresh the object displayed in an instance of XOBJVIEW.

```
PRO xobjview_refresh_event, event
    WIDGET_CONTROL, event.id, GET_UVALUE=aval
    WIDGET_CONTROL, event.top, GET_UVALUE=state
```

```

CASE uval OF
  'red': BEGIN
    WIDGET_CONTROL, event.id, GET_VALUE=val
    state.myobj->GetProperty, COLOR=c
    state.myobj->SetProperty, COLOR=[val,c[1],c[2]]
    XOBJVIEW, REFRESH=state.tlb
  END
  'green': BEGIN
    WIDGET_CONTROL, event.id, GET_VALUE=val
    state.myobj->GetProperty, COLOR=c
    state.myobj->SetProperty, COLOR=[c[0],val,c[2]]
    XOBJVIEW, REFRESH=state.tlb
  END
  'blue': BEGIN
    WIDGET_CONTROL, event.id, GET_VALUE=val
    state.myobj->GetProperty, COLOR=c
    state.myobj->SetProperty, COLOR=[c[0],c[1],val]
    XOBJVIEW, REFRESH=state.tlb
  END
ENDCASE
END

PRO xobjview_refresh_cleanup, wID
  WIDGET_CONTROL, wID, GET_UVALUE=state
  OBJ_DESTROY, state.myobj
END

PRO xobjview_refresh
  base = WIDGET_BASE(/COLUMN, TITLE='Adjust Object Color', $
    XOFFSET=420, XSIZE=200)

  myobj = OBJ_NEW('IDLgrSurface', $
    BESELJ(SHIFT(DIST(40), 20, 20) / 2,0) * 20, $
    COLOR=[255, 60, 60], STYLE=2, SHADING=1)
  XOBJVIEW, myobj, TLB=tlb, GROUP=base, BACKGROUND=[0,0,0]

  red = WIDGET_SLIDER(base, /DRAG, MIN=0, MAX=255, TITLE='Red', $
    UVALUE='red', VALUE=255)
  green = WIDGET_SLIDER(base, /DRAG, MIN=0, MAX=255, $
    TITLE='Green', UVALUE='green', VALUE=60)
  blue = WIDGET_SLIDER(base, /DRAG, MIN=0, MAX=255, $
    TITLE='Blue', UVALUE='blue', VALUE=60)

  WIDGET_CONTROL, base, /REALIZE

  state = {myobj:myobj, tlb:tlb}
  WIDGET_CONTROL, base, SET_UVALUE=state

```

```
        XMANAGER, 'xobjview_refresh', base, /NO_BLOCK, $
        CLEANUP='xobjview_refresh_cleanup'
END
```

Version History

Introduced: 5.3

See Also

[XOBJVIEW_ROTATE](#), [XOBJVIEW_WRITE_IMAGE](#)

XOBJVIEW_ROTATE

The XOBJVIEW_ROTATE procedure is used to programmatically rotate the object currently displayed in XOBJVIEW. XOBJVIEW must be called prior to calling XOBJVIEW_ROTATE. This procedure can be used to create animations of object displays.

This routine is written in the IDL language. Its source code can be found in the file `xobjview_rotate.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XOBJVIEW_ROTATE, Axis, Angle [, /PREMULTIPLY]
```

Arguments

Axis

A 3-element vector of the form $[x, y, z]$ describing the axis about which the model is to be rotated.

Angle

The amount of rotation, measured in degrees.

Keywords

PREMULTIPLY

Set this keyword to cause the rotation matrix specified by *Axis* and *Angle* to be pre-multiplied to the model's transformation matrix. By default, the rotation matrix is post-multiplied.

Examples

The following example creates an animation of the test object (a surface) currently displayed in XOBJVIEW. It does this by rotating the surface through 360 degrees in increments of 10 degrees using XOBJVIEW_ROTATE, and writing the display image to a BMP file for each increment using XOBJVIEW_WRITE_IMAGE.

```
PRO RotateAndWriteObject

XOBJVIEW, /TEST
FOR i = 0, 359 DO BEGIN
    XOBJVIEW_ROTATE, [0, 1, 0], 1, /PREMULTIPLY;
    XOBJVIEW_WRITE_IMAGE, 'img' + $
    STRCOMPRESS(i, /REMOVE_ALL) + '.bmp', 'bmp'
ENDFOR

END
```

Version History

Introduced: 5.5

See Also

[XOBJVIEW](#), [XOBJVIEW_WRITE_IMAGE](#)

XOBJVIEW_WRITE_IMAGE

The XOBJVIEW_WRITE_IMAGE procedure is used to write the object currently displayed in XOBJVIEW to an image file with the specified name and file format. XOBJVIEW must be called prior to calling XOBJVIEW_WRITE_IMAGE.

This routine is written in the IDL language. Its source code can be found in the file `xobjview_write_image.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XOBJVIEW_WRITE_IMAGE, Filename, Format [, DIMENSIONS=[x, y] ]
```

Arguments

Filename

A scalar string containing the name of the file to write.

Format

A scalar string containing the name of the file format to write. See [WRITE_IMAGE](#) for a list of supported formats.

Keywords

DIMENSIONS

Set this keyword to a 2-element vector of the form [*x*, *y*] specifying the size of the output image, in pixels. If this keyword is not specified, the image will be written using the dimensions of the current XOBJVIEW draw widget.

Examples

See [XOBJVIEW_ROTATE](#).

Version History

Introduced: 5.5

See Also

[XOBJVIEW](#), [XOBJVIEW_ROTATE](#)

XPALETTE

The XPALETTE procedure is a utility that displays a widget interface that allows interactive creation and modification of colortables using the RGB, CMY, HSV, or HLS color systems. Single colors can be defined or multiple color indices between two endpoints can be interpolated.

This routine is written in the IDL language. Its source code can be found in the file `xpalette.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Using the XPALETTE Interface

Calling XPALETTE causes a graphical interface to appear. The elements of this interface are described below.

Plots on Left Side of Interface

Three plots show the current red, green, and blue vectors.

Status Region

The center of the XPALETTE widget is a status region containing:

- The total number of colors.
- The current color index. XPALETTE allows changing one color at a time. This color is known as the “current color” and is indicated in the color spectrum display with a special marker.
- The current mark index. The mark is used to remember a color index. Click the “Set Mark Button” to make the current color index the mark index.
- A sample of the current color. The special marker used in the color spectrum display prevents the user from seeing the color of the current index, but it is visible here.

Control Panel

A panel of 8 buttons control common XPALETTE functions:

- **Done:** Click this button to exit XPALETTE. The new color tables are saved in the `COLORS` common block and loaded to the display.
- **Predefined:** Click this button to start XLOADCT, allowing selection of one of the predefined color tables. Note that when you change the color map via XLOADCT, XPALETTE is not always able to keep its display accurate. This problem can be overcome by pressing the XPALETTE “Redraw” button after changing the colortable via XLOADCT.

- **Help:** Click this button to display help information.
- **Redraw:** Click this button to redraw the display using the current state of the color map.
- **Set Mark:** Click this button to set the value of the mark index to the current color index.
- **Switch Mark:** Click this button to exchange the mark and the current index.
- **Copy Current:** Click this button to make every color lying between the current index and the mark index (inclusive) the same color as the current color.
- **Interpolate:** Click this button to smoothly interpolate colors between the current index and the mark index.

Color System Control

This section of the interface allows you to select the color system used to modify individual colors. The “Select Color System” pulldown menu lets you select from four different systems—RGB, CMY, HSV, and HLS. Depending upon the current system, 3 sliders below the pulldown menu allow you to alter the current color.

Right Side Color Spectrum Display

A display on the right side of the XPALETTE interface shows the current color map as a series of squares. Color index 0 is at the upper left. The color index increases monotonically by rows going left to right and top to bottom. The current color index is indicated by a special marker symbol. There are 4 ways to change the current color:

- Click on any square in the color map display.
- Use the “By Index” slider to move to the desired color index.
- Use the “Row” Slider to move the marker vertically.
- Use the “Column” Slider to move the marker horizontally.

A Note about the Colors Used in the Interface

XPALETTE uses two colors from the current color table as drawing foreground and background colors. These are used for the RGB plots on the left, and the current index marker on the right. This means that if the user set these two colors to the same value, the XPALETTE display could become unreadable (like writing on black paper with black ink). XPALETTE minimizes this possibility by noting changes to the color map and always using the brightest available color for the foreground color and the darkest for the background. Thus, the only way to make XPALETTE’s display

unreadable is to set the entire color map to a single color, which is highly unlikely. The only side effect of this policy is that you may notice XPALETTE redrawing the entire display after you've modified the current color. This simply means that the change has made XPALETTE pick new drawing colors.

Syntax

```
XPALETTE [, /BLOCK] [, GROUP=widget_id]  
        [, UPDATECALLBACK='procedure_name' [, UPDATECADATA=value]]
```

Arguments

None.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XPALETTE block, any earlier calls to XMANAGER must have been called with the [NO_BLOCK](#) keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

The widget ID of the widget that calls XPALETTE. When this ID is specified, a death of the caller results in a death of XPALETTE.

UPDATECALLBACK

Set this keyword to a string containing the name of a user-supplied procedure that will be called when the color table is updated by XLOADCT. The procedure may optionally accept a keyword called DATA, which will be automatically set to the value specified by the optional UPDATECADATA keyword.

UPDATECBDATA

Set this keyword to a value of any type. It will be passed via the DATA keyword to the user-supplied procedure specified via the UPDATECALLBACK keyword, if any. If the UPDATECBDATA keyword is not set the value accepted by the DATA keyword to the procedure specified by UPDATECALLBACK will be undefined.

Version History

Introduced: Pre 4.0

See Also

[LOADCT](#), [MODIFYCT](#), [XLOADCT](#), [TVLCT](#)

XPCOLOR

The XPCOLOR procedure is a utility that allows you to adjust the value of the current plotting color (foreground) using sliders, and store the desired color in the global system variable, !P.COLOR.

When XPCOLOR is called from the IDL input command line, the **Set Plot Color** dialog box appears. The dialog has two buttons (**Done** and **Help**) a single color swatch window, three sliders, and a pulldown menu with the four color systems: red, green, blue (RGB); cyan, magenta, yellow (CMY); hue, saturation, value (HSV); and hue, lightness, and saturation (HLS).

When you have chosen the color system and adjusted the sliders to your liking, click **Done** to store the color selected in the !P.COLOR system variable. Any plots generated in IDL afterwards use the color selected as the plotting (foreground) color until !P.COLOR is changed again.

Note

For a more flexible color editor, use the XPALETTE User Library routine.

This routine is written in the IDL language. Its source code can be found in the file `xpcolor.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XPCOLOR [, GROUP=widget_id ]
```

Arguments

None.

Keywords

GROUP

Set this keyword to the group leader widget ID as passed to XMANAGER.

Version History

Introduced: 4.0

XPLOT3D

The XPLOT3D procedure is a utility for creating and interactively manipulating 3-D plots.

This routine is written in the IDL language. Its source code can be found in the file `xplot3d.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Using XPLOT3D

XPLOT3D displays a resizable top-level base with a menu, toolbar and draw widget, as shown in the following figure:

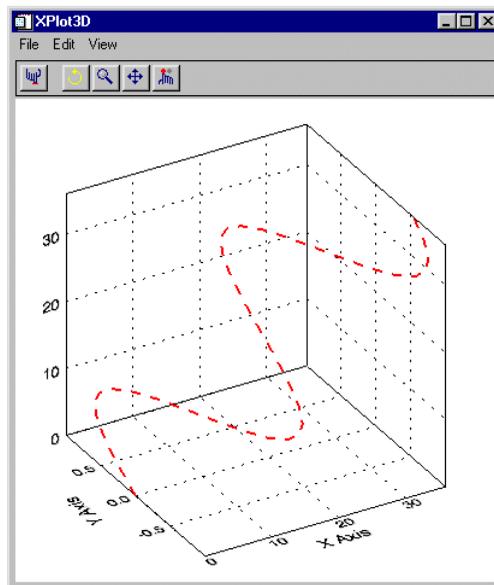


Figure 3-72: The XPLOT3D Utility

The XPLOT3D Toolbar

The XPLOT3D toolbar contains the following buttons:



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the plot and drag to rotate.



Zoom: Click the left mouse button on the plot and drag to zoom in or out.



Pan: Click the left mouse button on the plot and drag to pan.



Select: Click on a curve to display the curve name (if defined with the NAME keyword) on the XPLOT3D toolbar. If no name was defined for the curve, “IDLGRPOLYLINE” is displayed.

Projecting Data onto Plot “Walls”

To turn on or off the projection of data onto the walls of the box enclosing the 3-D plot, select **All On**, **All Off**, **XY**, **YZ**, or **XZ** from the **View** → **2D Projection** menu.

Changing the Axis Type

The **View** → **Axes** menu allows you to select one of the following types of axes:

- Simple Axes — displays the X, Y, and Z axes as lines.
- Box Axes — displays the X, Y, and Z axes as planes.
- No Axes — turns off the display of axes.

Syntax

```
XPLOT3D, X, Y, Z [, /BLOCK] [, COLOR=[r,g,b]] [, /DOUBLE_VIEW]
  [, GROUP=widget_id] [, LINESYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, /MODAL]
  [, NAME=string] [, /OVERPLOT] [, SYMBOL=objref(s)] [, /TEST]
  [, THICK=points{1.0 to 10.0}] [, TITLE=string] [, XRANGE=[min, max]]
  [, XTITLE=string] [, YRANGE=[min, max]] [, YTITLE=string]
  [, ZRANGE=[min, max]] [, ZTITLE=string]
```

Arguments

X

A vector of X data values.

Y

A vector of Y data values.

Z

A vector of Z data values.

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XPLOT3D block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

COLOR

Set this keyword to an $[r, g, b]$ triplet specifying the color of the curve.

DOUBLE_VIEW

Set this keyword to cause XPLOT3D to set the DOUBLE property on the IDLgrView that it uses to display the plot.

GROUP

Set this keyword to the widget ID of the widget that calls XPLOT3D. When this keyword is specified, the death of the caller results in the death of XPLOT3D.

LINestyle

Set this keyword to a value indicating the line style that should be used to draw the curve. The value can be either an integer value specifying a pre-defined line style, or a 2-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINestyle keyword to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot

- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1s or 0s in the *bitmask* should be repeated. (That is, if three consecutive 0s appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq \text{repeat} \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. The *bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

MODAL

Set this keyword to block processing of events from other widgets until the user quits XPLOT3D. The MODAL keyword does not require a group leader to be specified. If no group leader is specified, and the MODAL keyword is set, XPLOT3D fabricates an invisible group leader for you.

Note

To be modal, XPLOT3D does not require that its caller specify a group leader. This is unlike other IDL widget procedures such as XLOADCT, which, to be modal, do require that their caller specify a group leader. These other procedures were implemented this way to encourage the caller to create a modal widget that will be well-behaved with respect to layering and iconizing. (See [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 2255 for more information.)

To provide a simple means of invoking XPLOT3D as a modal widget in applications that contain no other widgets, XPLOT3D can be invoked as MODAL without specifying a group leader, in which case XPLOT3D fabricates an invisible group leader for you. For applications that contain multiple widgets, however, it is good programming practice to supply an appropriate group leader when invoking XPLOT3D, /MODAL. As with other IDL widget procedures with names prefixed with “X”, specify the group leader via the GROUP keyword.

NAME

Set this keyword to a string specifying the name for the data curve being plotted. The name is displayed on the XPLOT3D toolbar when the curve is selected with the

mouse. (To select the curve with the mouse, XPLOT3D must be in select mode. You can put XPLOT3D in select mode by clicking on the rightmost button on the XPLOT3D toolbar.)

OVERPLOT

Set this keyword to draw the curve in the most recently created view. The TITLE, [XYZ]TITLE, [XYZ]RANGE, and MODAL keywords are ignored if this keyword is set.

SYMBOL

Set this keyword to a vector containing one or more instances of the IDLgrSymbol object class to indicate the plotting symbols to be used at each vertex of the polyline. If there are more vertices than elements in SYMBOL, the elements of the SYMBOL vector are cyclically repeated. By default, no symbols are drawn. To remove symbols from a polyline, set SYMBOL to a scalar.

TEST

If set, the X, Y, and Z arguments are not required (and are ignored if provided). A sinusoidal curve is displayed instead. This allows you to test code that uses XPLOT3D without having to specify plot data.

THICK

Set this keyword to a value between 1.0 and 10.0, specifying the line thickness to be used to draw the polyline, in points. The default is 1.0 points.

TITLE

Set this keyword to a string to appear in the XPLOT3D title bar.

XRANGE

Set this keyword to a 2-element array of the form $[min, max]$ specifying the X-axis range.

XTITLE

Set this keyword to a string specifying the title for the X axis of the plot.

YRANGE

Set this keyword to a 2-element array of the form $[min, max]$ specifying the Y-axis range.

YTITLE

Set this keyword to a string specifying the title for the Y axis of the plot.

ZRANGE

Set this keyword to a 2-element array of the form $[min, max]$ specifying the Z-axis range.

ZTITLE

Set this keyword to a string specifying the title for the Z axis of the plot.

Examples

The following example displays two curves in XPLOT3D, using a custom plotting symbol for one of the curves:

```

;Define plot data:
X = INDGEN(20)
Y1 = SIN(X/3.)
Y2 = COS(X/3.)
Z = X

;Display curve 1 in XPLOT3D:
XPLOT3D, X, Y1, Z, NAME='Curve1', THICK=2

;Define custom plotting symbols:
oOrb = OBJ_NEW('orb', COLOR=[0, 0, 255])
oOrb->Scale, .75, .1, .5
oSymbol = OBJ_NEW('IDLgrSymbol', oOrb)

;Overplot curve 2 in XPLOT3D:
XPLOT3D, X, Y2, Z, COLOR=[0,255,0], NAME='Curve2', $
    SYMBOL=oSymbol, THICK=2, /OVERPLOT

```

This code results in the following:

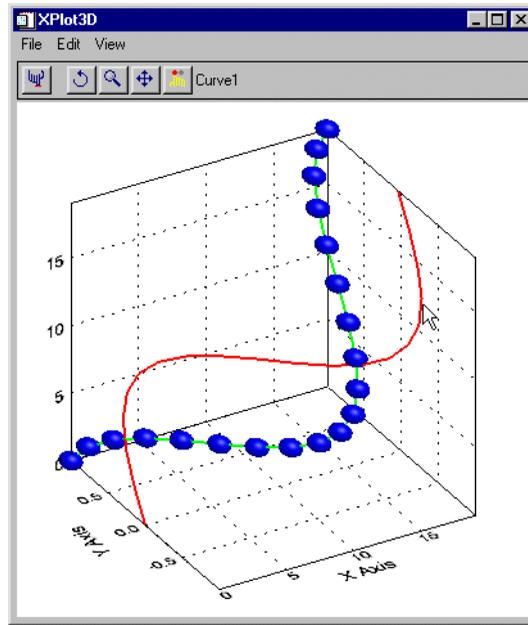


Figure 3-73: Two curves displayed in XPLOT3D

Version History

Introduced: 5.4

See Also

[I PLOT](#)

XREGISTERED

The XREGISTERED function returns the number of instances of the widget named as its argument that are currently registered with the XMANAGER. The registered widget is brought to the front of the desktop unless the NOSHOW keyword is set.

If the specified widget is not currently registered with XMANAGER, XREGISTERED returns zero.

This routine is written in the IDL language. Its source code can be found in the file `xregistered.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = XREGISTERED(*Name* [, /NOSHOW])

Arguments

Name

A string containing the name of the widget in question.

Note

XREGISTERED checks for *Name* in a COMMON block created by XMANAGER. The stored name is case-sensitive.

Keywords

NOSHOW

If the widget in question is registered, it is brought to the front of all the other windows by default. Set this keyword to keep the widget from being brought to the front.

Examples

Suppose that you have a widget program that registers itself with the XMANAGER with the command:

```
XMANAGER, 'mywidget', base
```

You could limit this widget to one instantiation by adding the following line as the first line (after the procedure definition statement) of the widget creation routine:

```
IF ( XREGISTERED('mywidget') NE 0 ) THEN RETURN
```

Version History

Introduced: Pre 4.0

See Also

[XMANAGER](#)

XROI

The XROI procedure is a utility for interactively defining regions of interest (ROIs), and obtaining geometry and statistical data about these ROIs.

This routine is written in the IDL language. Its source code can be found in the file `xroi.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Note

See [“Using XROI”](#) on page 2627, for detailed information describing importing images, modifying image and ROI colors, retrieving ROI information and growing regions.

Syntax

```
XROI [, ImageData] [, R] [, G] [, B] [, /BLOCK]
      [, /FLOATING] [, GROUP=widget_ID] [, /MODAL] [, REGIONS_IN=value]
      [, REGIONS_OUT=value] [, REJECTED=variable] [, RENDERER={0 | 1}]
      [, ROI_COLOR=[r, g, b] or variable] [, ROI_GEOMETRY=variable]
      [, ROI_SELECT_COLOR=[r, g, b] or variable] [, STATISTICS=variable]
      [, TITLE=string] [, TOOLS=string/string array { valid values are
      'Translate-Scale', 'Rectangle', 'Ellipse', 'Freehand Draw', 'Polygon Draw', and
      'Selection' }] [, X_SCROLL_SIZE=value] [, Y_SCROLL_SIZE=value]
```

Arguments

ImageData

ImageData is both an input and output argument. It is an array representing an 8-bit or 24-bit image to be displayed. *ImageData* can be any of the following:

- [*m, n*] — 8-bit image
- [3, *m, n*] — 24-bit image
- [*m, 3, n*] — 24-bit image
- [*m, n, 3*] — 24-bit image

If *ImageData* is not supplied, the user will be prompted for a file via `DIALOG_PICKFILE`. On output, *ImageData* will be set to the current image data. (The current image data can be different than the input image data if the user imported an image via the **File** → **Import Image** menu item.)

R, G, B

R, *G*, and *B* are arrays of bytes representing red, green, or blue color table values, respectively. *R*, *G*, and *B* are both input and output arguments. On input, these values are applied to the image if the image is 8-bit. To get the red, green, or blue color table values for the image on output from XROI, specify a named variable for the appropriate argument. (If the image is 24-bit, this argument will output a 256-element byte array containing the values given at input, or BINDGEN(256) if the argument was undefined on input.)

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XROI block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the NO_BLOCK keyword to XMANAGER for an example.

FLOATING

Set this keyword, along with the GROUP keyword, to create a floating top-level base widget. If the windowing system provides Z-order control, floating base widgets appear above the base specified as their group leader. If the windowing system does not provide Z-order control, the FLOATING keyword has no effect.

Note

Floating widgets must have a group leader. Setting this keyword without also setting the GROUP keyword causes an error.

GROUP

Set this keyword to the widget ID of the widget that calls XROI. When this keyword is specified, the death of the caller results in the death of XROI.

MODAL

Set this keyword to block other IDL widgets from receiving events while XROI is active.

REGIONS_IN

Set this keyword to an array of IDLgrROI references. This allows you to open XROI with previously defined regions of interest (see [Example 3](#)). This is also useful when using a loop to open multiple images in XROI. By using the same named variable for both the REGIONS_IN and REGIONS_OUT keywords, you can reuse the same ROIs in multiple images (see [Example 2](#)). This keyword also accepts -1, or OBJ_NEW() (Null object) to indicate that there are no ROIs to read in. This allows you to assign the result of a previous REGIONS_OUT to REGIONS_IN without worrying about the case where the previous REGIONS_OUT is undefined.

REGIONS_OUT

Set this keyword to a named variable that will contain an array of IDLgrROI references. This keyword is assigned the null object reference if there are no ROIs defined. By using the same named variable for both the REGIONS_IN and REGIONS_OUT keywords, you can reuse the same ROIs in multiple images (see [Example 2](#)).

Note

This keyword must be used in conjunction with the BLOCK keyword.

REJECTED

Set this keyword to a named variable that will contain those REGIONS_IN that are not in REGIONS_OUT. The objects defined in the variable specified for REJECTED can be destroyed with a call to OBJ_DESTROY, allowing you to perform cleanup on objects that are not required (see [Example 2](#)). This keyword is assigned the null object reference if no REGIONS_IN are rejected by the user.

Note

This keyword must be used in conjunction with the BLOCK keyword.

RENDERER

Set this keyword to an integer value to indicate which graphics renderer to use when drawing objects within the window. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL's software implementation (the default)

ROI_COLOR

This keyword is both an input and an output parameter. Set this keyword to a 3-element byte array, $[r, g, b]$, indicating the color of ROI outlines when they are not selected. This color will be used by XROI unless and until the color is changed by the user via the “Unselected Outline Color” portion of the “ROI Outline Colors” dialog (which is accessed by selecting **Edit** → **ROI Outline Colors**). If this keyword is assigned a named variable, that variable will be set to the current $[r, g, b]$ value at the time that XROI returns.

ROI_GEOMETRY

Set this keyword to a named variable that will contain an array of anonymous structures, one for each ROI that is valid when this routine returns. The structures will contain the following fields:

Field	Description
area	The area of the region of interest, in square pixels.
centroid	The coordinates (x, y, z) of the centroid of the region of interest, in pixels.
perimeter	The perimeter of the region of interest, in pixels.

Table 3-144: Fields of the structure returned by ROI_GEOMETRY

If there are no valid regions of interest when this routine returns, ROI_GEOMETRY will be undefined.

Note

If there are no REGIONS_IN, XROI must either be modal or must block control flow in order for ROI_GEOMETRY to be defined upon exit from XROI. Otherwise, XROI will return before an ROI can be defined, and ROI_GEOMETRY will therefore be undefined.

Note

This keyword must be used in conjunction with the BLOCK keyword.

ROI_SELECT_COLOR

This keyword is both an input and an output parameter. Set this keyword to a 3-element byte array, $[r, g, b]$, indicating the color of ROI outlines when they are selected. This color will be used by XROI unless and until the color is changed by the user via the “Selected Outline Color” portion of the “ROI Outline Colors” dialog (which is accessed by selecting **Edit** → **ROI Outline Colors**). If this keyword is assigned a named variable, that variable will be set to the current $[r, g, b]$ value at the time that XROI returns.

STATISTICS

Set this keyword to a named variable to receive an array of anonymous structures, one for each ROI that is valid when this routine returns. The structures will contain the following fields:

Field	Description
count	Number of pixels in region.
minimum	Minimum pixel value.
maximum	Maximum pixel value.
mean	Mean pixel value.
stddev	Standard deviation of pixel values.

Table 3-145: Fields of the structure returned by STATISTICS

If *ImageData* is 24-bit, or if there are no valid regions of interest when the routine exits, STATISTICS will be undefined.

Note

If there are no REGIONS_IN, XROI must either be modal or must block control flow in order for STATISTICS to be defined upon exit from XROI. Otherwise, XROI will return before an ROI can be defined, and STATISTICS will therefore be undefined.

TITLE

Set this keyword to a string to appear in the XROI title bar.

TOOLS

Set this keyword a string or vector of strings from the following list to indicate which ROI manipulation tools should be supported when XROI is run:

- 'Translate-Scale' — Translation and scaling of ROIs. Mouse down inside the bounding box selects a region, mouse motion translates (repositions) the region. Mouse down on a scale handle of the bounding box enables scaling (stretching, enlarging and shrinking) of the region according to mouse motion. Mouse up finishes the translation or scaling.
- 'Rectangle' — Rectangular ROI drawing. Mouse down positions one corner of the rectangle, mouse motions creates the rectangle, positioning the rectangle's opposite corner, mouse up finishes the rectangular region.
- 'Ellipse' — Elliptical ROI drawing. Mouse down positions the center of the ellipse, mouse motion positions the corner of the ellipse's imaginary bounding box, mouse up finishes the elliptical region.
- 'Freehand Draw' — Freehand ROI drawing. Mouse down begins a region, mouse motion adds vertices to the region (following the path of the mouse), mouse up finishes the region.
- 'Polygon Draw' — Polygon ROI drawing. Mouse down begins a region, subsequent mouse clicks add vertices, double-click finishes the region.
- 'Selection' — ROI selection. Mouse down/up selects the nearest region. The nearest vertex in that region is identified with a crosshair symbol.

If more than one string is specified, a series of bitmap buttons will appear at the top of the XROI widget in the order specified (to the right of the fixed set of bitmap buttons used for saving regions, displaying region information, copying to clipboard, and flipping the image). If only one string is specified, no additional bitmap buttons will appear, and the manipulation mode is implied by the given string. If this keyword is not specified, bitmap buttons for all three manipulation tools are included on the XROI toolbar.

X_SCROLL_SIZE

Set this keyword to the width of the scroll window. If this keyword is larger than the image width then it will be set to the image width. The default is to use the image width or the screen width, whichever is smaller.

Y_SCROLL_SIZE

Set this keyword to the height of the scroll window. If this keyword is larger than the image height then it will be set to the image height. The default is to use the image height or the screen height, whichever is smaller.

Using XROI

XROI displays a top-level base with a menu, toolbar and draw widget. After defining an ROI, the **ROI Information** window appears, as shown in the following figure:

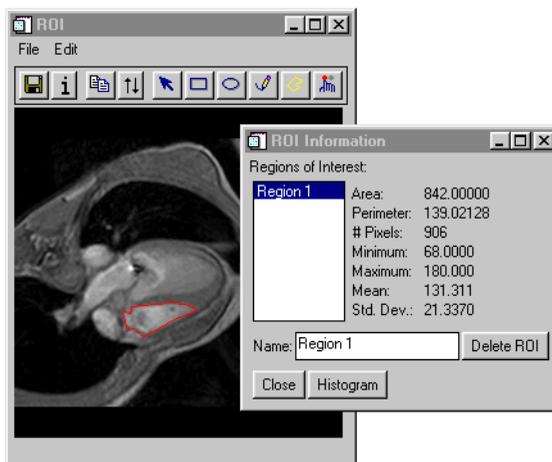


Figure 3-74: The XROI Utility

As you move the mouse over an image, the x and y pixel locations are shown in the status line on the bottom of the XROI window. For 8-bit images, the data value (z) is also shown. If an ROI is defined, the status line also indicates the mouse position relative to the ROI using the text “Inside”, “Outside”, “On Edge,” or “On Vertex.”

The XROI Toolbar

The XROI toolbar contains the following buttons:



Save:

Opens a file selection dialog for saving the currently defined ROIs to a save file.

-  **Info:** Opens the **ROI Information** window.
-  **Copy:** Copies the contents of the display area to the clipboard.
-  **Flip:** Flips image vertically. Note that only the image is flipped; any ROIs that have been defined do not move.

Depending on the value of the TOOLS keyword, the XROI toolbar may also contain the following buttons:

-  **Translate/Scale:** Click this button to translate or scale ROIs. Mouse down inside the bounding box selects a region, mouse motion translates (repositions) the region. Mouse down on a scale handle of the bounding box enables scaling (stretching, enlarging and shrinking) of the region according to mouse motion. Mouse up finishes the translation or scaling.
-  **Draw Rectangle:** Click this button to draw rectangular ROIs. Mouse down positions one corner of the rectangle, mouse motions creates the rectangle, positioning the rectangle's opposite corner, mouse up finishes the rectangular region.
-  **Draw Ellipse:** Click this button to draw elliptical ROIs. Mouse down positions the center of the ellipse, mouse motion positions the corner of the ellipse's imaginary bounding box, mouse up finishes the elliptical region.
-  **Draw Freehand:** Click this button to draw freehand ROIs. Mouse down begins a region, mouse motion adds vertices to the region (following the path of the mouse), mouse up finishes the region.
-  **Draw Polygon:** Click this button to draw polygon ROIs. Mouse down begins a region, subsequent mouse clicks add vertices, double-click finishes the region.
-  **Select:** Click this button to select an ROI region. Clicking the image causes a cross hairs symbol to be drawn at the nearest vertex of the selected ROI.

Importing an Image into XROI

To import an image into XROI, select **File** → **Import Image**. This opens a `DIALOG_READ_IMAGE` dialog, which can be used to preview and select an image.

Changing the Image Color Table

To change the color table properties for the current image, select **Edit** → **Image Color Table**. This opens the `CW_PALETTE_EDITOR` dialog, which is a compound widget used to edit color palettes. See [CW_PALETTE_EDITOR](#) for more information. This menu item is grayed out if the image does not have a color palette.

Changing the ROI Outline Colors

To change the outline colors for selected and unselected ROIs, select **Edit** → **ROI Outline Colors**. This opens the **ROI Outline Colors** dialog, which consists of two `CW_RGBSLIDER` widgets for interactively adjusting the ROI outline colors. The left widget is used to define the color for the selected ROI, and the right widget is used to define the color of unselected ROIs. You can select the RGB, CMY, HSV, or HLS color system from the **Color System** drop-down list.

Viewing ROI Information

To view geometry and statistical data about the currently selected ROI, click the **Info** button or select **Edit** → **ROI Information**. This opens the **ROI Information** dialog, which displays area, perimeter, number of pixels, minimum and maximum pixel values, mean, and standard deviation. Values for statistical information (minimum, maximum, mean, and standard deviation) appear as “N/A” for 24-bit images.

Viewing a Histogram Plot for an ROI

To view a histogram for an ROI, use either the shortcut menu or the ROI Information dialog.

To view an ROI's histogram plot using the shortcut menu:

1. Position the cursor on the line defining the boundary of an ROI in the drawing window and click the right mouse button. This selects the region and brings up its shortcut menu.
2. Select the **Plot Histogram** menu option from the shortcut menu.

To view an ROI's histogram plot using the ROI Information dialog:

1. Open the ROI Information dialog by clicking the **Info** button or selecting **Edit** → **ROI Information**.

2. Select a region from the list and click the **Histogram** button on the ROI Information dialog.

Either of the previous methods opens an iTool showing the ROI's histogram that can be used to interactively control the plot properties.

Note

XROI's histogram plot feature now supports RGB images.

Growing an ROI

Once a region has been created, it may be used as a source ROI for region growing. Region growing is a process of generating one or more new ROIs based upon the image pixel values that fall within the source ROI and the values of the neighboring pixels. New pixels are added to the new grown region if those image pixel values fall within a specified threshold.

Note

This option is an interactive implementation of the `REGION_GROW` function.

To create a new, grown region, do the following:

1. Within the draw area, click the right mouse button on the ROI that is to be grown. This will select the region and bring up its shortcut menu.
2. Select **Grow Region** → **By threshold** or select **Grow Region** → **By std. dev. multiple** from the shortcut menu to control how the region is grown.

The **By threshold** option grows the region to include all neighboring pixels that fall within a specified threshold range. By default, the range is defined by the minimum and maximum pixel values occurring within the original region. To specify a different threshold range, see [Using the Region Grow Properties Dialog](#) in the following section.

The **By std. dev. multiple** option grows a region to include all neighboring pixels that fall within the range of:

$$\text{Mean} \pm \text{StdDevMultiplier} * \text{StdDev}$$

where `Mean` is the mean value of the pixel values within the source ROI, `StdDevMultiplier` is a multiplier that is set using the Region Grow Properties dialog (described below), and `StdDev` is the sample standard deviation of the pixel values within the original region.

Using the Region Grow Properties Dialog

The Region Grow Properties dialog allows you to view and edit the properties associated with a region growing process. To bring up the Region Grow Properties dialog, do one of the following:

- Click the right mouse button on an ROI in the drawing window and select **Grow Region** → **Properties...** shortcut menu option.
- Select **Edit** → **Region Grow Properties...** from the XROI menu bar.

This brings up the Region Grow Properties dialog, shown in the following figure.

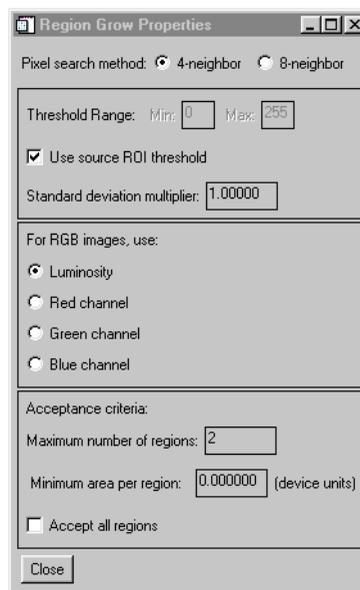


Figure 3-75: XROI's Region Grow Properties Dialog

The Region Grow Properties dialog offers the following options:

Option	Description
Pixel search method:	<p>Describes which pixels are searched when growing the original ROI. The option are:</p> <ul style="list-style-type: none"> • 4-neighbor — Searches only the four neighboring pixels that share a common edge with the current pixel. This is the default. • 8-neighbor — Searches all eight neighboring pixels, including those that are located diagonally relative to the original pixel and share a common corner.
Threshold range:	<p>Represents the minimum and maximum image pixel values that are to be included in the grown region when using the Grow Region → By threshold option (described in “Growing an ROI” on page 2630). By default, the range of pixel values used are those occurring in the ROI to be grown.</p> <p>To change the threshold values, uncheck Use source ROI threshold and enter the minimum and maximum threshold values in the Min: and Max: fields provided.</p>
Standard deviation multiplier:	<p>Represents the factor by which the sample standard deviation of the original ROI’s pixel values is multiplied. This factor only applies when the Grow Region → By std. dev. multiple option (described in “Growing an ROI” on page 2630) is used.</p> <p>Change the multiplier value by typing the value into the Standard deviation multiplier field provided.</p>

Table 3-146: Options of the Region Grow Properties Dialog

Option	Description
For RGB image, use:	<p>Determines the basis of region growing for an RGB (rather than indexed) image. The image data values used when growing a RGB region can be one of the following:</p> <ul style="list-style-type: none"> • Luminosity — Uses the luminosity values associated with an RGB image. This is the default method. Luminosity is computed as: $\text{Luminosity} = (0.3 * \text{Red}) + (0.59 * \text{Green}) + (0.11 * \text{Blue})$ • Red Channel, Green Channel or Blue Channel — Uses the ROI's red, green or blue channel as a basis for region growing. Click the channel's associated button to specify the channel to be used. <p>Note - For indexed images, the image data itself is always used for region growing.</p>
Acceptance criteria:	<p>Determines which contours of the grown region are accepted as new regions, (which will also be displayed in the draw area and in the ROI Information dialog list of regions). The region growing process can result in a large number of contours, some of which may be considered insignificant. By default, no more than two regions (those with the greatest geometrical area) are accepted. Modify the acceptance criteria by altering the following values:</p> <ul style="list-style-type: none"> • Maximum number of regions: — Specifies the upper limit of the number of regions to create when growing an ROI. • Minimum area per region: — Specifies that only contours having a geometric area (computed in device coordinates) of at least the value stated are accepted and displayed. • Accept all regions: — Select this option to accept all generated contours, regardless of count or area.

Table 3-146: Options of the Region Grow Properties Dialog (Continued)

Deleting an ROI

An ROI can be deleted using either the shortcut menu or using the ROI Information dialog.

To delete an ROI using the shortcut menu:

1. Click the right mouse button on the line defining the boundary of the ROI in the drawing area that you wish to delete. This selects the region and bring up the shortcut menu.
2. Select the **Delete** menu option from the shortcut menu.

To delete an ROI using the ROI Information dialog:

1. Click the **Info** button or select **Edit** → **ROI Information**. This opens the **ROI Information** dialog.
2. In the **ROI Information** dialog, select the ROI you wish to delete from the list of ROIs. You can also select an ROI by clicking the **Select** button on the XROI toolbar, then clicking on an ROI on the image.
3. Click the **Delete ROI** button.

Examples

Example 1

This example opens a single image in XROI:

```
image = READ_PNG(FILEPATH('mineral.png', $
    SUBDIR=['examples', 'data']))
XROI, image
```

Example 2

This example reads 3 images from the file `mr_abdomen.dcm`, and calls XROI for each image. A single list of regions is maintained, saving the user from having to redefine regions on each image:

```
;Read 3 images from mr_abdomen.dcm and open each one in XROI:
FOR i=0,2 DO BEGIN
    image = READ_DICOM(FILEPATH('mr_abdomen.dcm', $
        SUBDIR=['examples', 'data']), IMAGE_INDEX=i)
    XROI, image, r, g, b, REGIONS_IN = regions, $
        REGIONS_OUT = regions, $
        ROI_SELECT_COLOR = roi_select_color, $
        ROI_COLOR = roi_color, REJECTED = rejected, /BLOCK
```

```
        OBJ_DESTROY, rejected
    ENDFOR
```

```
OBJ_DESTROY, regions
```

Perform the following steps:

1. Draw an ROI on the first image, then close that XROI window. Note that the next image contains the ROI defined in the first image. This is accomplished by setting `REGIONS_IN` and `REGIONS_OUT` to the same named variable in the FOR loop of the above code.
2. Draw another ROI on the second image.
3. Click the **Select** button and select the first ROI. Then click the **Info** button to open the **ROI Information** window, and click the **Delete ROI** button.
4. Close the second XROI window. Note that the third image contains the ROI defined in the second image, but not the ROI deleted on the second image. This example sets the `REJECTED` keyword to a named variable, and calls `OBJ_DESTROY` on that variable. Use of the `REJECTED` keyword is not necessary to prevent deleted ROIs from appearing on subsequent images, but allows you perform cleanup on objects that are no longer required.

Example 3

XROI's **File** → **Save ROIs** option allows you to save selected regions of interest. This example shows how to restore such a save file. Suppose you have a file named `mineralRoi.sav` that contains regions of interest selected in the `mineral.png` image file. You would need to complete the following steps to restore the file:

1. First, restore the file, `mineralRoi.sav`. Provide a value for the `RESTORE` procedure's `RESTORED_OBJECTS` keyword. Using the scenario stated above, you could enter the following:

```
RESTORE, 'mineralRoi.sav', RESTORED_OBJECTS = myRoi
```

2. Pass the restored object data containing your regions of interest into XROI by specifying `myRoi` as the value for `REGIONS_IN` as follows:

```
XROI, READ_PNG(FILEPATH('mineral.png', SUBDIRECTORY = $
  ['examples', 'data'])), REGIONS_IN = myRoi
```

This opens the previously selected regions of interest in the XROI utility.

Version History

Introduced: 5.4

X_SCROLL_SIZE and Y_SCROLL_SIZE keywords added: 5.6

See Also

[IIMAGE](#)

XSQ_TEST

The XSQ_TEST function computes the Chi-square goodness-of-fit test between observed frequencies and the expected frequencies of a theoretical distribution.

Expected frequencies of magnitude less than 5 are combined with adjacent elements resulting in a reduction of cells used to formulate the chi-squared test statistic. If the observed frequencies differ significantly from the expected frequencies, the Chi-square test statistic will be large and the fit is poor. This situation requires the rejection of the hypothesis that the given observed frequencies are an accurate approximation to the expected frequency distribution.

This routine is written in the IDL language. Its source code can be found in the file `xsq_test.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = XSQ_TEST( Obfreq, Exfreq [, EXCELL=variable] [, OBCELL=variable]
                  [, RESIDUAL=variable] )
```

Return Value

The result is a two-element vector containing the Chi-square test statistic X2 and the one-tailed probability of obtaining a value of X2 or greater.

Arguments

Obfreq

An n -element integer, single-, or double-precision floating-point vector containing observed frequencies.

Exfreq

An n -element integer, single-, or double-precision floating-point vector containing expected frequencies.

Keywords

EXCELL

Set this keyword to a named variable that will contain a vector of expected frequencies used to formulate the Chi-square test statistic. If each of the expected frequencies contained in *Exfreq*, has a magnitude of 5 or greater, then this vector is identical to *Exfreq*. If *Exfreq* contains elements of magnitude less than 5, adjacent expected frequencies are combined. The identical combinations are performed on the corresponding elements of *Obfreq*.

OBCELL

Set this keyword to a named variable that will contain a vector of observed frequencies used to formulate the Chi-square test statistic. The elements of this vector are often referred to as the “cells” of the observed frequencies. The length of this vector is determined by the length of EXCELL described below.

RESIDUAL

Set this keyword to a named variable that will contain a vector of signed differences between corresponding cells of observed frequencies and expected frequencies.

```
RESIDUAL[i] = OBCELL[i] - EXCELL[i].
```

The length of this vector is determined by the length of EXCELL described above.

Examples

```
; Define the vectors of observed and expected frequencies:
obfreq = [2, 1, 4, 15, 10, 5, 3]
exfreq = [0.5, 2.1, 5.9, 10.3, 10.7, 7.0, 3.5]

; Test the hypothesis that the given observed frequencies are an
; accurate approximation to the expected frequency distribution:
result = XSQ_TEST(obfreq, exfreq)
PRINT, result
```

IDL Output

```
3.05040      0.383920
```

Since the vector of expected frequencies contains elements of magnitude less than 5, adjacent expected frequencies are combined resulting in fewer cells. The identical combinations are performed on the corresponding elements of observed frequencies.

The computed value of 0.383920 indicates that there is no reason to reject the proposed hypothesis at the 0.05 significance level.

Version History

Introduced: 4.0

See Also

[CTL_TEST](#)

XSURFACE

The XSURFACE procedure is a utility that provides a graphical interface to the SURFACE and SHADE_SURF commands. Different controls are provided to change the viewing angle and other plot parameters. The command used to generate the resulting surface plot is shown in a text window. Note that this procedure does not accept SURFACE or SHADE_SURF keywords.

This routine is written in the IDL language. Its source code can be found in the file `xsurface.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XSURFACE, Data [, /BLOCK] [, GROUP=widget_id]
```

Arguments

Data

The two-dimensional array to display as a wire-mesh or shaded surface.

Keywords

BLOCK

Set this keyword to have XMANAGER *block* when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting BLOCK=1 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XSURFACE block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

Set this keyword to the widget ID of the widget that calls XSURFACE. When GROUP is specified, the death of the calling widget results in the death of XSURFACE.

Examples

```
; Make a 2-D array:  
z = DIST(30)  
  
; Call XSURFACE. The XSURFACE widget appears:  
XSURFACE, z
```

Version History

Introduced: Pre 4.0

See Also

[ISURFACE](#), [SHADE_SURF](#), [SURFACE](#)

XVAREEDIT

The XVAREEDIT procedure is a utility that provides a widget-based editor for any IDL variable. Use the input fields to change desired values of the variable or array. Click “Accept” to write the new values into the variable. Click “Cancel” to exit XVAREEDIT without saving changes.

This routine is written in the IDL language. Its source code can be found in the file `xvareedit.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XVAREEDIT, Var [, NAME='variable_name'{ignored if variable is a structure}]  
    [, GROUP=widget_id] [, X_SCROLL_SIZE=columns]  
    [, Y_SCROLL_SIZE=rows]
```

Arguments

Var

The variable to be edited. On output, this variable contains the edited value if the user selects the “Accept” button, or the original value if the user selects the “Cancel” button.

Keywords

NAME

The NAME of the variable. This keyword is overwritten with the structure name if the variable is a structure.

GROUP

The widget ID of the widget that calls XVAREEDIT. When this ID is specified, a death of the caller results in a death of XVAREEDIT.

X_SCROLL_SIZE

Set this keyword to the column width of the scrolling viewport. The default is 4.

Y_SCROLL_SIZE

Set this keyword to the row width of the scrolling viewport. The default is 4.

Version History

Introduced: Pre 4.0

XVOLUME

The `XVOLUME` procedure is a utility for viewing and interactively manipulating volumes and isosurfaces.

This routine is written in the IDL language. Its source code can be found in the file `xvolume.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Tip

The `XVOLUME_ROTATE` and `XVOLUME_WRITE_IMAGE` procedures, which can be called only after a call to `XVOLUME`, can be used to easily create animations of volumes and isosurfaces displayed in `XVOLUME`. See `XVOLUME_ROTATE` for an example.

Using XVOLUME

`XVOLUME` displays a resizable top-level base with a toolbar, a menu, a graphical interface for controlling volume and isosurface properties, and a draw widget for displaying and manipulating the volume, as shown in the following figure:

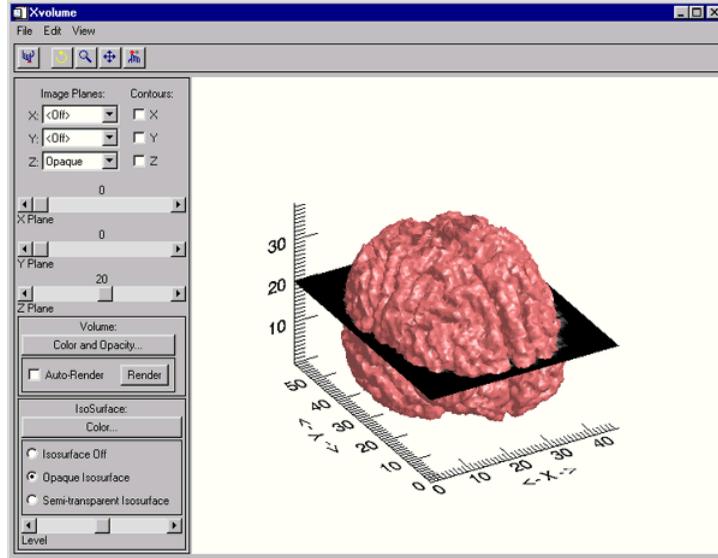


Figure 3-76: The `XVOLUME` Utility

The XVOLUME Toolbar

The XVOLUME toolbar contains the following buttons.

Note

If you have the **Auto-Render** option selected, the Rotate, Zoom, and Pan features may be more difficult to use. For the best performance while manipulating the orientation of a volume using these features, uncheck the **Auto-Render** option.



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the volume and drag to rotate.



Zoom: Click the left mouse button on the volume and drag to zoom in or out.



Pan: Click the left mouse button on the volume and drag to pan.



Select: Click in the draw widget to identify the selected item. A name identifying the selected item is displayed next to the Select button.

The XVOLUME Interface

The XVOLUME interface provides the following elements for controlling the display of image planes and contours, volumes, and isosurfaces:

Image Planes and Contours

Image planes and contours allow you to visualize the values associated with the volume or isosurface at a specified X, Y, or Z plane.

- **Image Planes** — Select one of the following options from the drop-down list for each dimension to control the display of image planes:
 - **Off:** Turns off the image plane display.
 - **Opaque:** Displays an opaque image plane at the location specified by the corresponding plane slider.
 - **Transparent:** Displays a transparent image plane at the location specified by the corresponding plane slider. The transparency value of the plane is taken from the volume at the current location of the image plane.

- **Contours** — Check this option to display contours on the specified plane at the location specified by corresponding the plane slider.
- **Plane Sliders** — Move these sliders to change the position of the plane in each dimension.

Volume

- **Color and Opacity:** Click this button to change the color and/or opacity of the current volume. This opens a `CW_PALETTE_EDITOR` dialog, which is a compound widget used to edit color palettes. See [CW_PALETTE_EDITOR](#) for more information.
- **Auto-Render:** Select this option to have rendering executed automatically after each change you make to the volume. If **Auto-Render** is unchecked, you must manually click the **Render** button to see changes you have made to the volume. If **Auto-Render** is checked, the **Render** button will be grayed out.
- **Render:** Click on this button to execute rendering computations and display the current volume. If **Auto-Render** is checked, this button will be grayed out.

Isosurface

An isosurface is a 3-D surface on which the data values are constant along the entire surface. Use the following elements to control the appearance of the isosurface:

- **Color:** Click this button to change the color system and/or values for the current isosurface. This opens a `CW_RGBSLIDER` dialog, which is a compound widget that provides a drop-down list for selecting the RGB, CMY, HSV, or HLS color system, and three sliders for adjusting the values associated with each color system.
- **Isosurface Off:** Select this option to turn off the isosurface display.
- **Opaque Isosurface:** Select this option to display an opaque isosurface.
- **Semi-transparent Isosurface:** Select this option to display a semi-transparent isosurface.
- **Level:** Use this slider to adjust the threshold value of the isosurface.

Syntax

```
XVOLUME, Vol, [, /BLOCK] [, GROUP=widget_id] [, /INTERPOLATE]
[, /MODAL] [, RENDERER={0 | 1}] [, /REPLACE] [, SCALE=value] [, /TEST]
[, XSIZE=pixels] [, YSIZE=pixels]
```

Arguments

Vol

A 3-dimensional array of the form $[x, y, z]$ that specifies a data volume.

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XVOLUME block, any earlier calls to XMANAGER must have been called with the [NO_BLOCK](#) keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

Set this keyword to the widget ID of the widget that calls XVOLUME. When this keyword is specified, the death of the caller results in the death of XVOLUME.

INTERPOLATE

Set this keyword to indicate that trilinear interpolation is to be used when rendering the volume and the image planes. Setting this keyword improves the quality of images produced, at the cost of more computing time, especially when the volume has low resolution with respect to the size of the viewing plane. Nearest neighbor sampling is used by default.

MODAL

Set this keyword to block processing of events from other widgets until the user quits XVOLUME. The MODAL keyword does not require a group leader to be specified. If no group leader is specified, and the MODAL keyword is set, XVOLUME fabricates an invisible group leader for you.

Note

To be modal, XVOLUME does not require that its caller specify a group leader. This is unlike other IDL widget procedures such as XLOADCT, which, to be modal, do require that their caller specify a group leader. These other procedures were implemented this way to encourage the caller to create a modal widget that will be well-behaved with respect to layering and iconizing. (See [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 2255 for more information.)

To provide a simple means of invoking XVOLUME as a modal widget in applications that contain no other widgets, XVOLUME can be invoked as MODAL without specifying a group leader, in which case XVOLUME fabricates an invisible group leader for you. For applications that contain multiple widgets, however, it is good programming practice to supply an appropriate group leader when invoking XVOLUME, /MODAL. As with other IDL widget procedures with names prefixed with “X”, specify the group leader via the GROUP keyword.

RENDERER

Set this keyword to an integer value indicating which graphics renderer to use when drawing objects within the window. Valid values are:

- 0 = Platform native OpenGL (the default)
- 1 = IDL’s software implementation

REPLACE

If this keyword is set, and there is a current instance of XVOLUME running, the volume displayed in XVOLUME is replaced with the volume specified by *Vol*. For example, display volume1 using the command

```
XVOLUME, volume1
```

To replace volume1 with volume2, you would use the command

```
XVOLUME, volume2, /REPLACE
```

SCALE

Set this keyword to the zoom factor for the initial view. The default is 1/SQRT(3). This default value provides the largest possible view of the volume, while ensuring that no portion of the volume will be clipped by the XVOLUME window, regardless of the volume’s orientation.

When using the `SCALE` keyword for `XVOLUME`, you can scale with a 3-element array of values [x, y, z].

TEST

If set, the `Vol` argument is not required (and is ignored if provided). A volume of random numbers is displayed instead. This allows you to test code that uses `XVOLUME` without having to specify volume data.

XSIZE

The width of the drawable area in pixels.

YSIZE

The height of the drawable area in pixels.

Examples

Create a volume and display using `XVOLUME`:

```
; Create a volume:
vol = BYTSCL(RANDOMU((SEED=0),5,5,5))
vol = CONGRID(vol, 30,30,30)

; Display volume:
XVOLUME, vol
```

Version History

Introduced: 5.4

See Also

[IVOLUME](#), [XVOLUME_ROTATE](#), [XVOLUME_WRITE_IMAGE](#), [IDLgrVolume](#), [INTERPOLATE](#), [ISOSURFACE](#), [SHADE_VOLUME](#), [SLICER3](#), “Volume Objects” in Chapter 32 of the *Using IDL* manual.

XVOLUME_ROTATE

The `XVOLUME_ROTATE` procedure is used to programmatically rotate the volume currently displayed in `XVOLUME`. `XVOLUME` must be called prior to calling `XVOLUME_ROTATE`. This procedure can be used to create animations of volumes and isosurfaces.

This routine is written in the IDL language. Its source code can be found in the file `xvolume_rotate.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XVOLUME_ROTATE, Axis, Angle [, /PREMULTIPLY]
```

Arguments

Axis

A 3-element vector of the form $[x, y, z]$ describing the axis about which the model is to be rotated.

Angle

The amount of rotation, measured in degrees.

Keywords

PREMULTIPLY

Set this keyword to cause the rotation matrix specified by *Axis* and *Angle* to be pre-multiplied to the model's transformation matrix. By default, the rotation matrix is post-multiplied.

Examples

The following example creates an animation of the volume currently displayed in `XVOLUME`. It does this by rotating the volume through 360 degrees in increments of 10 degrees using `XVOLUME_ROTATE`, and writing the volume to a BMP file for each increment using `XVOLUME_WRITE_IMAGE`. It then loops through the images and uses `TV` to display each image.

First, display a volume as follows:

```

; Create a volume:
vol = BYTSCL(RANDOMU((SEED=0),5,5,5))
vol = CONGRID(vol, 30,30,30)

; Display volume:
XVOLUME, vol

```

Now, use the `XVOLUME` interface to modify the orientation and appearance of the volume or isosurface as desired. Once you have the volume or isosurface displayed the way you want it, run the following program:

```

PRO spin_volume

inc = 10. ; degrees.
; Create images
FOR i=0,(360./inc)-2 DO BEGIN
  XVOLUME_WRITE_IMAGE, $
    'spin' + STRCOMPRESS(i, /REMOVE_ALL) + '.bmp', 'bmp'
  XVOLUME_ROTATE, [0,0,1], inc, /PREMULTIPLY
ENDFOR
XVOLUME_ROTATE, [0,0,1], inc, /PREMULTIPLY

; Read images
img = READ_BMP('spin0.bmp')
siz = SIZE(img, /DIM)
arr = BYTARR(3, siz[1], siz[2], 360./inc-1)
FOR i=0,360./inc-2 DO BEGIN
  img = READ_BMP( $
    'spin' + STRCOMPRESS(i, /REMOVE_ALL) + '.bmp', /RGB)
  arr[0,0,0, i] = img
  PRINT, i
ENDFOR

; Display animation
FOR i=0,2 DO BEGIN ; num rotations
  FOR j=0,(360./inc)-2 DO BEGIN
    TV, arr[*,*,*,j], /TRUE
  ENDFOR
ENDFOR

TV, arr[*,*,*,0], /TRUE

END

```

Version History

Introduced: 5.4

See Also

[XVOLUME](#), [XVOLUME_WRITE_IMAGE](#)

XVOLUME_WRITE_IMAGE

The `XVOLUME_WRITE_IMAGE` procedure is used to write the volume currently displayed in `XVOLUME` to an image file with the specified name and file format. `XVOLUME` must be called prior to calling `XVOLUME_WRITE_IMAGE`.

This routine is written in the IDL language. Its source code can be found in the file `xvolume_write_image.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XVOLUME_WRITE_IMAGE, Filename, Format [, DIMENSIONS=[x, y] ]
```

Arguments

Filename

A scalar string containing the name of the file to write.

Format

A scalar string containing the name of the file format to write. See [QUERY_IMAGE](#) for a list of supported formats.

Keywords

DIMENSIONS

Set this keyword to a 2-element vector of the form [*x*, *y*] specifying the size of the output image, in pixels. If this keyword is not specified, the image will be written using the dimensions of the current `XVOLUME` draw widget.

Examples

See [XVOLUME_ROTATE](#).

Version History

Introduced: 5.4

See Also

[XVOLUME](#), [XVOLUME_ROTATE](#)

XYOUTS

The XYOUTS procedure draws text on the currently-selected graphics device starting at the designated coordinate.

Arguments *X*, *Y*, and *String* can be any combination of scalars or arrays. If the arguments are arrays, multiple strings are output.

If the optional *X* and *Y* arguments are omitted, the text is positioned at the end of the most recently output text string.

Important keywords that control the appearance and positioning of the text include: ALIGNMENT, the justification of the text; CHARSIZE, the size of the text; FONT, chooses between vector drawn and hardware fonts; COLOR, the color of the text; and ORIENTATION, the angle between the baseline of the text and the horizontal. With hardware fonts, most of the text attributes, (e.g., size and orientation), are predetermined and not changeable.

Note

Specify the Z coordinate with the Z keyword when positioning text in three dimensions.

Syntax

```
XYOUTS, [X, Y] String [, ALIGNMENT=value{0.0 to 1.0}] [, CHARSIZE=value]
  [, CHARTHICK=value] [, TEXT_AXES={0 | 1 | 2 | 3 | 4 | 5}]
  [, WIDTH=variable]
```

```
Graphics Keywords: [, CLIP=[X0, Y0, X1, Y1]] [, COLOR=value][, /DATA | ,
  /DEVICE | , /NORMAL] [, FONT=integer]
  [, ORIENTATION=ccw_degrees_from_horiz] [, /NOCLIP] [, /T3D] [, Z=value]
```

Arguments

X, Y

The horizontal and vertical coordinates used to position the string(s). *X* and *Y* are normally interpreted in data coordinates. The DEVICE and NORMAL keywords can be used to specify the coordinate units.

X and *Y* can be arrays of positions if *String* is an array.

String

The string(s) to be output. This argument can be a scalar string or an array of strings. If this argument is not a string, it is converted prior to use using the default formatting rules. If *String* is an array, *X*, *Y*, and the **COLOR** keyword can also be arrays so that each string can have a separate location and color.

Keywords

ALIGNMENT

Specifies the alignment of the text baseline. An alignment of 0.0 (the default) aligns the left edge of the text baseline with the given (x, y) coordinate. An alignment of 1.0 right-justifies the text, while 0.5 results in text centered over the point (x, y) .

CHARSIZE

The overall character size for the annotation. A CHARSIZE of 1.0 is normal. Setting CHARSIZE = -1 suppresses output of the text string. This keyword has no effect when used with the hardware drawn fonts; for exceptions, see [“Scaled Hardware Fonts”](#) on page 2657.

CHARTHICK

The line thickness of the vector drawn font characters. This keyword has no effect when used with the hardware drawn fonts; for exceptions, see [“Scaled Hardware Fonts”](#) on page 2657. The default value is 1.0.

TEXT_AXES

This keyword specifies the plane of vector drawn text when three-dimensional plotting is enabled. By default, text is drawn in the plane of the XY axes. The horizontal text direction is in the X plane, and the vertical text direction is in the Y plane. Values for this keyword can range from 0 to 5, with the following effects: 0 for XY, 1 for XZ, 2 for YZ, 3 for YX, 4 for ZX, and 5 for ZY. The notation ZY means that the horizontal direction of the text lies in the Z plane, and the vertical direction of the text is drawn in the Y plane.

WIDTH

Set this keyword to a named variable in which to return the width of the text string, in normalized coordinate units.

Graphics Keywords Accepted

See [Appendix B, “Graphics Keywords”](#) for the description of graphics and plotting keywords not listed above. [CLIP](#), [COLOR](#), [DATA](#), [DEVICE](#), [FONT](#), [NOCLIP](#), [NORMAL](#), [ORIENTATION](#), [T3D](#), [Z](#).

Examples

Print the string “This is text” at device coordinate position (100,100):

```
XYOUTS, 100, 100, 'This is text', /DEVICE
```

Print an array of strings with each element of the array printed at a different location. Use larger text than in the previous example:

```
XYOUTS, [0, 200, 250], [200, 50, 100], $
      ['This', 'is', 'text'], CHARSIZE = 3, /DEVICE
```

Determine the text size for a window device before opening an on-screen window:

```
WINDOW, /FREE, /PIXMAP, XSIZE=myWinXSize, YSIZE=myWinYSize
XYOUTS, 'Check this out', WIDTH=w
WDELETE
```

myWinXSize and *myWinYSize* are chosen to match your onscreen window. Since we can not know the characteristics of a given device (such as character size) until a window has been opened, the PIXMAP keyword to WINDOW allows you to compute appropriate dimensions for text with an invisible window before displaying a window on your screen.

Scaled Hardware Fonts

One example of hardware fonts which can be scaled are PostScript fonts. If you are using PostScript fonts, the keywords CHARTHICK and CHARSIZE will have an effect on a call to XYOUTS. Of the devices we provide that support hardware fonts, only the PostScript device uses scalable PostScript fonts for its “hardware” font system. All other devices use a bitmapped font technology.

Scaling is related to whether or not a device supports Hershey formatting commands when hardware fonts are used. Formatting requires the ability to scale the text on a per-character basis (i.e. for subscripting). To see if a given device supports Hershey formatting when hardware fonts are used, look at bit 12 of !D.FLAGS. You can also use this indicator to determine whether or not the hardware fonts will be scaled.

Version History

Introduced: Original

See Also

[ANNOTATE](#), [PRINT/PRINTF](#)

ZOOM

The ZOOM procedure displays part of an image from the current window enlarged in a new (“zoom”) window. The cursor is used to mark the center of the zoom area, and different zoom factors can be specified interactively.

Note

ZOOM only works with color systems.

This routine is written in the IDL language. Its source code can be found in the file `zoom.pro` in the `lib` subdirectory of the IDL distribution.

Using ZOOM

After calling ZOOM, place the mouse cursor over an image in an IDL graphics window. Click the left mouse button to display a magnified version of the image in a new window. The zoomed image is centered around the pixel selected in the original window. Click the middle mouse button to display a menu of zoom factors. Click the right mouse button to exit the procedure.

Using ZOOM with Draw Widgets

Note that the ZOOM procedure is only for use with IDL graphics windows. It should not be used with draw widgets. To obtain a zooming effect in a draw widget, use the `CW_ZOOM` function.

Syntax

```
ZOOM [, /CONTINUOUS] [, FACT=integer] [, /INTERP] [, /KEEP]
      [, /NEW_WINDOW] [, XSIZE=value] [, YSIZE=value]
      [, ZOOM_WINDOW=variable]
```

Arguments

None.

Keywords

CONTINUOUS

Set this keyword to make the zoom window track the mouse without requiring the user to press the left mouse button. This feature only works well on fast computers.

FACT

Use this keyword to specify the zoom factor, which must be an integer. The default zoom factor is 4.

INTERP

Set this keyword to use bilinear interpolation. The default is to use pixel replication.

KEEP

Set this keyword to keep the zoom window after exiting the procedure.

NEW_WINDOW

Normally, if ZOOM is called with KEEP and then called again, it will use the same window to display the new zoomed image. Set the NEW_WINDOW keyword to force ZOOM to create a new window for this purpose.

XSIZE

Use this keyword to specify the X size of the zoom window. The default is 512.

YSIZE

Use this keyword to specify the Y size of the zoom window. The default is 512.

ZOOM_WINDOW

Set this keyword to a named variable that will contain the index of the zoom window. KEEP must also be set. If KEEP is not set, ZOOM_WINDOW will contain the integer -1.

Version History

Introduced: Original

See Also

[CW_ZOOM](#), [ZOOM_24](#)

ZOOM_24

The ZOOM_24 procedure displays part of a 24-bit color image from the current window expanded in a new (“zoom”) window, and provides information about cursor location and color values in an auxiliary (“data”) window. The cursor is used to mark the center of the zoom area, and different zoom factors can be specified interactively.

Note

ZOOM only works on 24-bit color systems.

This routine is written in the IDL language. Its source code can be found in the file `zoom_24.pro` in the `lib` subdirectory of the IDL distribution.

Using ZOOM_24

After calling ZOOM_24, windows titled “Zoomed Image” (the zoom window) and “Pixel Values” (the data window) appear on the screen. Place the mouse cursor over a 24-bit color image in an IDL graphics window and click the left mouse button to display a magnified version of the image in the zoom window. The zoomed image is centered around the pixel selected in the original window. Move the mouse cursor in the zoom window to determine the coordinates (in the original image) and color values of individual pixels.

With the cursor located in the zoom window, click the right mouse button to return to selection mode, which allows you to either choose a new zoom center, change the zoom factor, or exit the procedure. Move the cursor to the original image and click the middle mouse button to display a menu of zoom factors, or click the right mouse button to exit the procedure.

Using ZOOM_24 with Draw Widgets

Note that the ZOOM_24 procedure is only for use with IDL graphics windows. It should not be used with draw widgets. To obtain a zooming effect in a draw widget, use the CW_ZOOM function.

Syntax

```
ZOOM_24 [, FACT=integer] [, /RIGHT] [, XSIZE=value] [, YSIZE=value]
```

Arguments

None.

Keywords

FACT

Use this keyword to specify the zoom factor, which must be an integer. The default zoom factor is 4.

RIGHT

Set this keyword to position the zoom and data windows to the right of the original window.

XSIZE

Use this keyword to specify the X size of the zoom window. The default is 512.

YSIZE

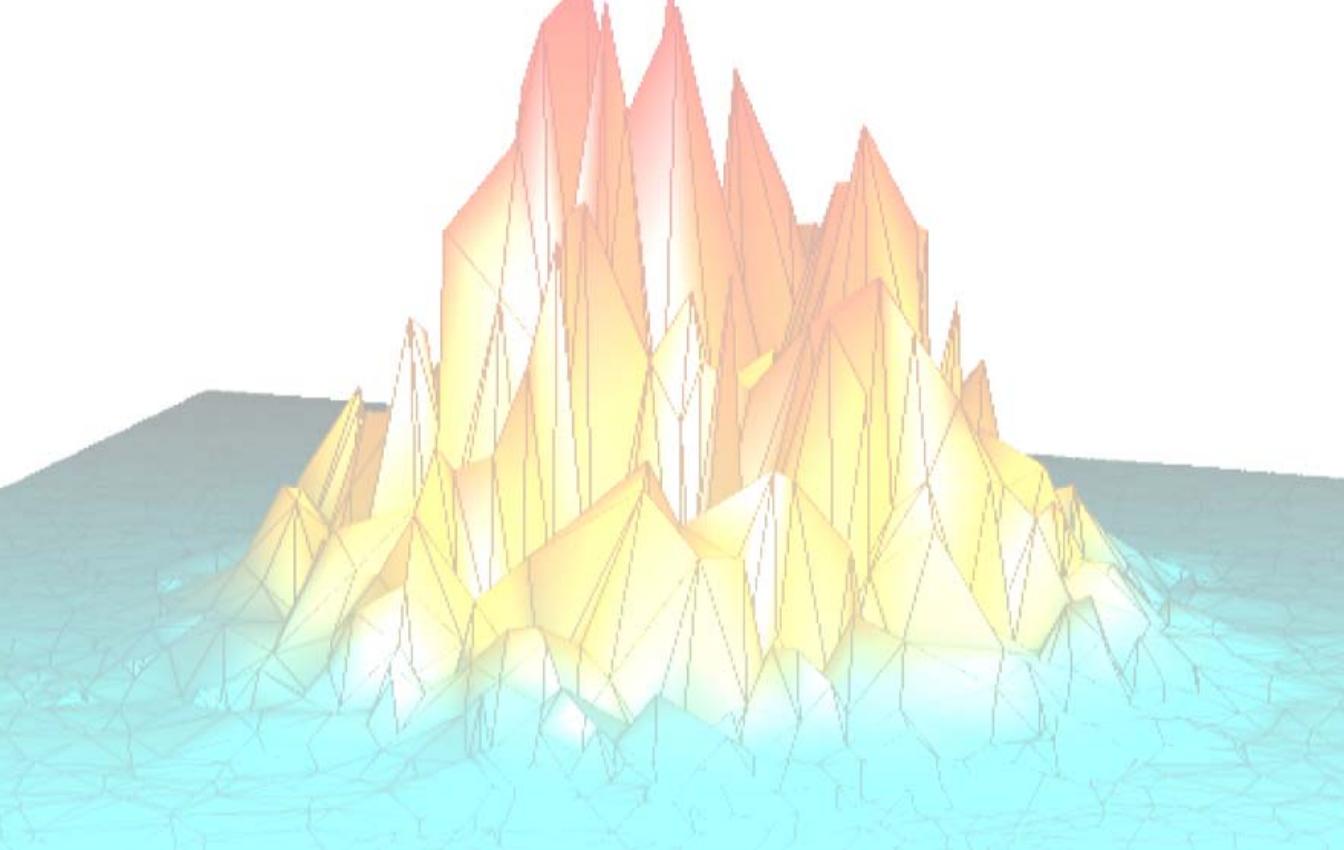
Use this keyword to specify the Y size of the zoom window. The default is 512.

Version History

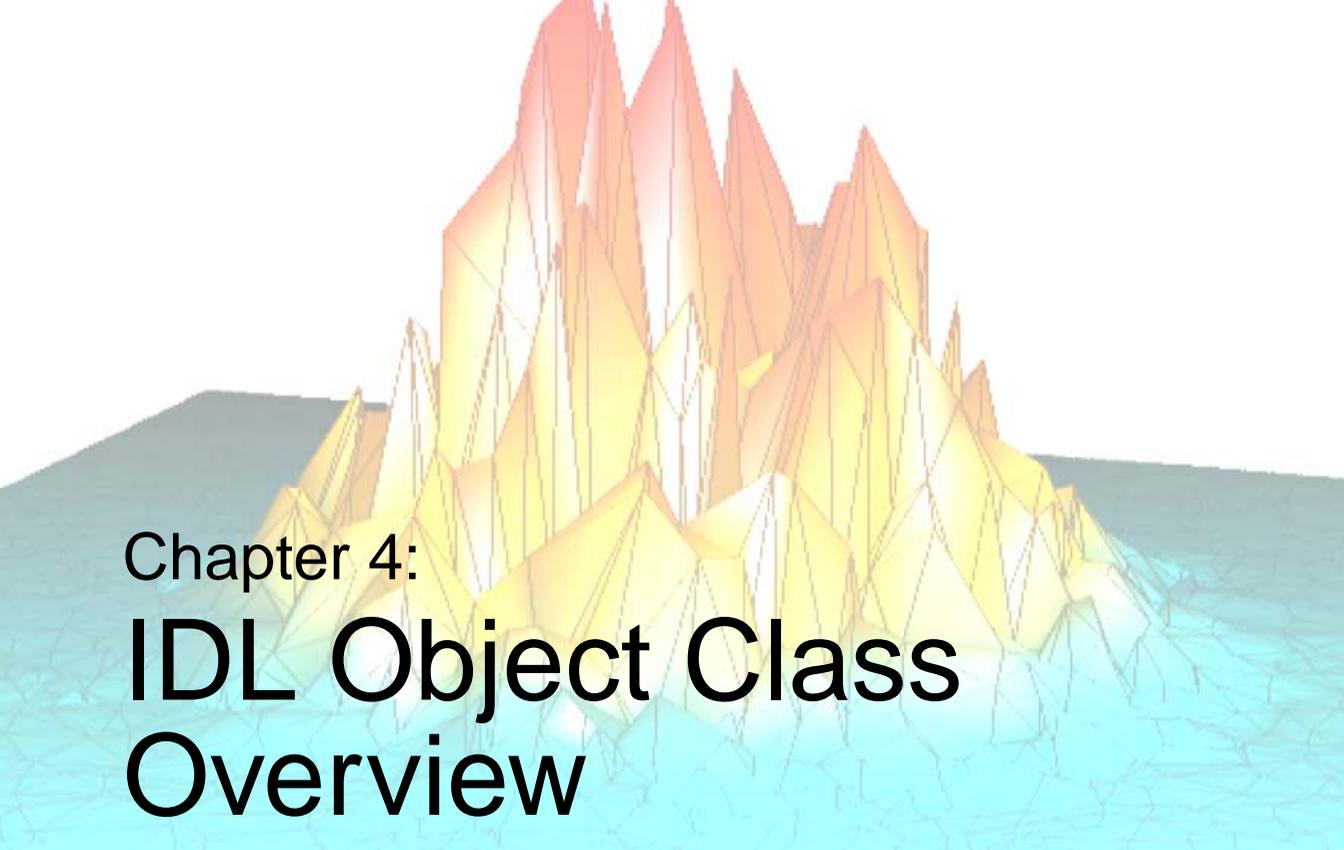
Introduced: Pre 4.0

See Also

[CW_ZOOM](#), [ZOOM](#)



Part II: Object Class and Method Reference



Chapter 4: IDL Object Class Overview

This section describes IDL's object class library.

Using the Class Reference	2666	File Format Object Classes	2727
Object Properties	2669	Graphics Object Classes	3539
Registered Properties	2673	iTools Object Classes	3082
Undocumented Object Classes	2677	Miscellaneous Object Classes	4181
Analysis Object Classes	2679		

Using the Class Reference

IDL's object class library is documented in this section. The page or pages describing each class include references to any superclasses of the class, to the properties of the class, and to the methods associated with the class. Class methods are documented alphabetically following the description of the class itself.

A description of each method follows its name. Beneath the general description of the method are a number of sections that describe the Syntax for the method, its arguments (if any), its keywords (if any). These sections are described below.

Syntax

The Syntax section shows the proper syntax for calling the method.

Procedure Methods

IDL procedure methods have the syntax:

Obj->Procedure_Name, Argument [, Optional_Arguments]

where *Obj* is a valid object reference, *Procedure_Name* is the name of the procedure method, *Argument* is a required parameter, and *Optional_Argument* is an optional parameter to the procedure method. The square brackets around optional arguments are not used in the actual call to the procedure, they are simply used to denote the optional nature of the arguments within this document.

Function Methods

IDL function methods have the syntax:

Result = Obj->Function_Name(Argument [, Optional_Arguments])

where *Obj* is a valid object reference, *Result* is the returned value of the function method, *Function_Name* is the name of the function method, *Argument* is a required parameter, and *Optional_Argument* is an optional parameter. The square brackets around optional arguments are not used in the actual call to the function, they are simply used to denote the optional nature of the arguments within this document.

Note

All arguments and keywords to functions should be supplied *within* the parentheses that follow the function's name.

Arguments

The *Arguments* section describes each valid argument to the method.

Note

These arguments are positional parameters that must be supplied in the order indicated by the method's syntax.

Named Variables

Often, arguments that contain values upon return from the function or procedure method (“output arguments”) are described as accepting “named variables.” A named variable is simply a valid IDL variable name. This variable *does not* need to be defined before being used as an output argument. Note, however that when an argument calls for a named variable, only a named variable can be used—sending an expression causes an error.

Keywords

The *Keywords* section describes each valid keyword argument to the method.

Note

Keyword arguments are formal parameters that can be supplied in any order.

Keyword arguments are supplied to IDL methods by including the keyword name followed by an equal sign (“=”) and the value to which the keyword should be set. Note that keywords can be abbreviated to their shortest unique length. For example, the XSTYLE keyword can be abbreviated to XST.

Note

In the case of Init, GetProperty and SetProperty methods, keywords often correspond to object *properties*. See [“Object Properties”](#) on page 2669 for additional discussion.

Setting Keywords

When the documentation for a keyword says something similar to, “Set this keyword to enable logarithmic plotting,” the keyword is simply a switch that turns an option on and off. Usually, setting such keywords equal to 1 causes the option to be turned on. Explicitly setting the keyword to zero (or not including the keyword) turns the option off.

There is a “shortcut” that can be used to set a keyword equal to 1 without the usual syntax (i.e., `KEYWORD=1`). To “set” a keyword, simply preface it with a slash character (“/”). For example, to create a surface object with a skirt around it, set the `SKIRT` keyword to the `SURFACE` routine as follows:

```
mySurface = OBJ_NEW('IDLgrSurface', DIST(10), /SKIRT)
```

Creating Objects from the Class Library

To create an object from the IDL object class library, use the `OBJ_NEW` function. See “[OBJ_NEW](#)” on page 1491. The `Init` method for each class describes the arguments and keywords available when you are creating a new object.

For example, to create a new object from the `IDLgrAxis` class, use the following call to `OBJ_NEW` along with the arguments and keywords accepted by the `IDLgrAxis::Init` method:

```
myAxis = OBJ_NEW(IDLgrAxis, DIRECTION = 1, RANGE = [0.0, 40.0])
```

Object Properties

Some IDL objects have *properties* associated with them — things like color, line style, size, and so on. Properties are set or changed by supplying property-value pairs in a call to the object class' `Init` or `SetProperty` method:

```
Obj->OBJ_NEW('ObjectClass', PROPERTY = value, ... )
```

or

```
Obj->SetProperty, PROPERTY = value, ...
```

where *PROPERTY* is the name of a property and *value* is the associated property value.

Property values are retrieved by supplying property-value pairs in a call to the object class' `GetProperty` method:

```
Obj->GetProperty, PROPERTY = variable, ...
```

where *PROPERTY* is the name of a property and *variable* is the name of an IDL variable that will hold the associated property value.

Note

Property-value pairs behave in exactly the same way as Keyword-value pairs. This means that you can set the value of a boolean property to 1 by preceding the name of the property with a “/” character. The following are equivalent:

```
Obj->SetProperty, PROPERTY = 1
```

```
Obj->SetProperty, /PROPERTY
```

If you are familiar with IDL Direct Graphics, you will note that many of the properties of IDL objects correspond to keywords to the Direct Graphics routines. Unlike IDL Direct Graphics, the IDL Object Graphics system allows you to change the value of an object's properties without re-creating the entire object. Objects must be redrawn, however, with a call to the destination object's `Draw` method, for the changes to become visible.

Properties and the Property Sheet Interface

In addition to being able to set and change object property values programmatically, IDL provides a way for users to change property values via a graphical user interface. The [WIDGET_PROPERTY SHEET](#) function creates a user interface that allows users to select and change property values using the mouse and keyboard.

For an object property to be displayed in a property sheet, the property must be registered. See “[Registered Properties](#)” on page 2673 for additional discussion.

Setting Properties at Initialization

Often, you will set an object’s properties when creating the object for the first time, which is done by specifying any keywords to the object’s Init method directly in the call of OBJ_NEW that creates the object. For example, suppose you are creating a plot and wish to use a red line to draw the plot line. You could specify the COLOR keyword to the IDLgrPlot::Init method directly in the call to OBJ_NEW:

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata, COLOR = [255, 0, 0])
```

In most cases, an object’s Init method cannot be called directly. Arguments to OBJ_NEW are passed directly to the Init method when the object is created.

For some graphics objects, you can specify a keyword that has the same meaning as an argument. In Object Graphics, the value of the keyword overrides the value set by the argument. For example,

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata, DATA_X = newXData)
```

The Plot object uses the data in newXData for the plot’s X data.

Setting Properties of Existing Objects

After you have created an object, you can also set its properties using the object’s SetProperty method. For example, the following two statements duplicate the single call to OBJ_NEW shown above:

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata)
myPlot->SetProperty, COLOR = [255, 0, 0]
```

Note

Not all keywords available when the object is being initialized are necessarily available via the SetProperty method. Keywords available when using an object’s SetProperty method are noted with the word “Set” in the table included after the text description of the property.

Retrieving Property Settings

You can retrieve the value of a particular property using an object’s GetProperty method. The GetProperty method accepts a list of keyword-variable pairs and returns

the value of the specified properties in the variables specified. For example, to return the value of the COLOR property of the plot object in our example, use the statement:

```
myPlot->GetProperty, COLOR = plotcolor
```

The value of the COLOR property is returned in the IDL variable `plotcolor`.

You can retrieve the values of all of the properties associated with a graphics object by using the ALL keyword to the object's GetProperty method. The following statement:

```
myPlot->GetProperty, ALL = allprops
```

returns an anonymous structure in the variable `allprops`; the structure contains the values of all of the retrievable properties of the object.

Note

Not all keywords available when the object is being initialized are necessarily available via the GetProperty method. Keywords available when using an object's GetProperty method are noted with the word "Get" in the table included after the text description of the property.

About Object Property Descriptions

In the documentation for the IDL object class library, the description of each class is followed by a section describing the properties of the class. Each property description is followed by a table that looks like this:

Property Type	Boolean		
Name String	Hide		
Get: Yes	Set: No	Init: Yes	Registered: Yes

where

- **Property Type** describes the property type associated with the property. If the property is *registered*, the property type will be one of the types described in ["Registered Property Data Types"](#) on page 2674. If the property is not registered, this field will describe the generic IDL data type of the property value.
- **Name String** is the default value of the Name property attribute. If the property is registered, this is the value that appears in the left-hand column when the property is displayed in a property sheet widget. If the property is not registered, this field will contain the words *not displayed*.

- **Get, Set,** and **Init** describe whether the property can be specified as a keyword to the `GetProperty`, `SetProperty`, and `Init` methods, respectively.
- **Registered** describes whether the property is registered for display in a property sheet widget. See [“Registered Properties”](#) on page 2673 for details on registered properties.

Registered Properties

In order for an object property to be displayed in the WIDGET_PROPERTY SHEET interface, which makes it possible for a user to interactively change the value of the property using the mouse and keyboard, the following two conditions must be met:

- The property must belong to an object that subclasses from the [IDLitComponent](#) class. IDLitComponent provides the infrastructure necessary to display property sheets.
- The property must be *registered*. Properties that are not registered will never be displayed in a property sheet. (Note that registered properties can also be *hidden*, which prevents them from being displayed in a property sheet.)

Note

This section provides a brief overview of property registration. See [Chapter 4, “Property Management”](#) in the *iTool Developer’s Guide* manual for a more in-depth discussion.

Registering a Property

To register a property, use the RegisterProperty method of the IDLitComponent class:

```
Obj->IDLitComponent::RegisterProperty, PropertyIdentifier, $  
    [, TypeCode] [, ATTRIBUTE = value]
```

where *PropertyIdentifier* is a string that uniquely identifies the property, *TypeCode* is an integer between 0 and 9 specifying the property data type, and *ATTRIBUTE* is a property attribute.

Property identifier strings must obey certain rules; see [“Property Identifiers”](#) in Chapter 4 of the *iTool Developer’s Guide* manual for details. Property type codes are discussed in [“Registered Property Data Types”](#) on page 2674. Property attributes are discussed in [“Property Attributes”](#) in Chapter 4 of the *iTool Developer’s Guide* manual.

Registering All Available Properties

Some of the classes in the IDL object class library include a mechanism to register all available properties in a single operation. If an object class has a REGISTER_PROPERTIES property, then setting that property to 1 when creating the object automatically registers all properties that can be registered. If a property contains a “Yes” in the Registered box of the property description table, it will be

registered automatically if the REGISTER_PROPERTIES property is set when the object is created. (See “[About Object Property Descriptions](#)” on page 2671 for a description of the property description table.)

Registered Property Data Types

Registered properties must be of one of the data types listed in [Table 4-1](#).

Note

Properties of objects that are *not* registered (that is, properties that can not appear in a property sheet) can be of any IDL data type.

Type Code	Type	Description
0	USERDEF	User Defined properties can contain values of any IDL type, but must also include a string value that will be displayed in the property sheet. See “ Property Data Types ” in Chapter 4 of the <i>iTool Developer’s Guide</i> manual for additional discussion of User Defined property types.
1	BOOLEAN	Boolean properties contain either the integer 0 or the integer 1.
2	INTEGER	Integer properties contain an integer value. If a property of integer data type has a VALID_RANGE attribute that includes an increment value, the property is displayed in a property sheet using a slider. If no increment value is supplied, the property sheet allows the user to edit values manually.
3	FLOAT	Float properties contain a double-precision floating-point value. If a property of float data type has a VALID_RANGE attribute that includes an increment value, the property is displayed in a property sheet using a slider. If no increment value is supplied, the property sheet allows the user to edit values manually.
4	STRING	String properties contain a scalar string value

Table 4-1: *iTools* property data types.

Type Code	Type	Description
5	COLOR	Color properties contain an RGB color triple
6	LINESTYLE	<p>Linestyle properties contain an integer value between 0 and 6, corresponding to the following IDL line styles:</p> <ul style="list-style-type: none"> • 0 = Solid • 1 = Dotted • 2 = Dashed • 3 = Dash Dot • 4 = Dash Dot Dot • 5 = Long Dashes • 6 = No Line <p>See Appendix B, “Property Controls” in the <i>iTool User’s Guide</i> manual for a visual example of the available line styles.</p>
7	SYMBOL	<p>Symbol properties contain an integer value between 0 and 8, corresponding to the following IDL symbol types:</p> <ul style="list-style-type: none"> • 0 = No symbol • 1 = Plus sign • 2 = Asterisk • 3 = Period (Dot) • 4 = Diamond • 5 = Triangle • 6 = Square • 7 = X • 8 = Arrow Head <p>See Appendix B, “Property Controls” in the <i>iTool User’s Guide</i> manual for a visual example of the available symbols.</p>

Table 4-1: *iTools* property data types.

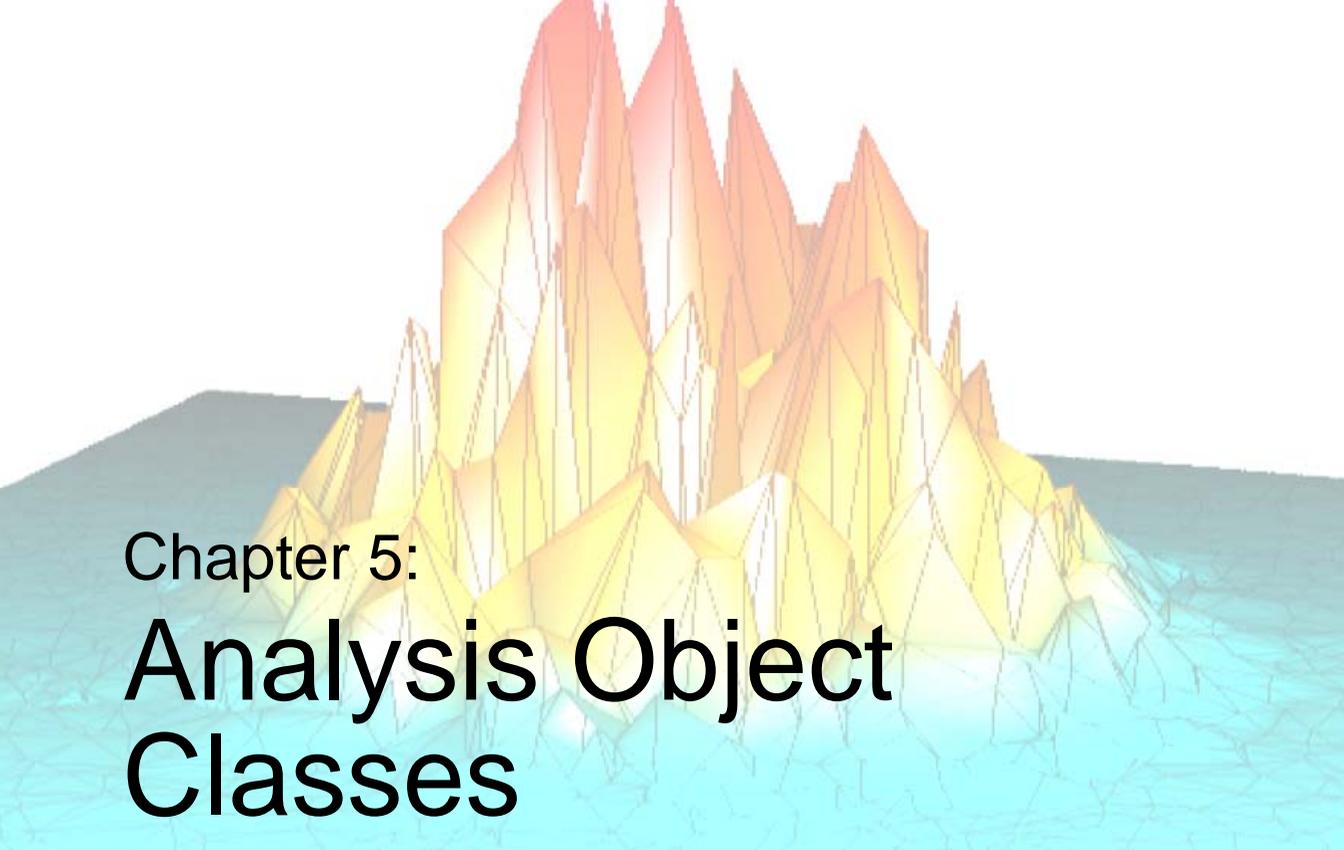
Type Code	Type	Description
8	THICKNESS	Thickness properties contain an integer value between 1 and 10, corresponding to the thickness (in pixels) of the line.
9	ENUMLIST	Enumerated List properties contain an array of string values defined when the property is registered. The GetProperty method returns the zero-based index of the selected item.

Table 4-1: iTools property data types.

Undocumented Object Classes

Some of IDL's object classes are subclassed from more generic IDL objects. You may see references to the generic IDL objects when using IDL's HELP procedure to get information on an object, or when you use the OBJ_ISA or OBJ_CLASS functions. You may also notice that the generic objects are not documented in this section. This is not an oversight.

We have chosen not to document the workings of the more generic objects from which the IDL graphics objects are subclassed because we reserve the right to make changes to their operation. We strongly recommend that you do not use the undocumented object classes directly, or subclass your own object classes from them. Research Systems, Inc. does not guarantee that user-written code that uses undocumented features will continue to function in future releases of IDL.



Chapter 5: Analysis Object Classes

This chapter describes IDL's built-in analysis class library.

IDLanROI	2680	IDLanROIGroup	2708
--------------------------------	------	-------------------------------------	------

IDLanROI

The IDLanROI object class represents a region of interest.

Note

The IDLan* naming convention is used for objects in the analysis domain.

Regions of interest are described as a set of vertices that may be connected to generate a path or a polygon, or may be treated as separate points. This object may be used as a source for analytical computations on regions. (For additional information about display of ROIs in Object Graphics, refer to the [IDLgrROI](#) object class.)

Superclasses

None

Creation

See [IDLanROI::Init](#).

Properties

Objects of this class have the following properties. See “[IDLanROI Properties](#)” on page 2682 for details on individual properties.

- [ALL](#)
- [BLOCK_SIZE](#)
- [DATA](#)
- [DOUBLE](#)
- [INTERIOR](#)
- [N_VERTS](#)
- [ROI_XRANGE](#)
- [ROI_YRANGE](#)
- [ROI_YRANGE](#)
- [ROI_ZRANGE](#)
- [TYPE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

The IDLanROI class has the following methods.

- [IDLanROI::AppendData](#)
- [IDLanROI::Cleanup](#)
- [IDLanROI::ComputeGeometry](#)
- [IDLanROI::ComputeMask](#)
- [IDLanROI::ContainsPoints](#)
- [IDLanROI::GetProperty](#)
- [IDLanROI::Init](#)
- [IDLanROI::RemoveData](#)
- [IDLanROI::ReplaceData](#)
- [IDLanROI::Rotate](#)
- [IDLanROI::Scale](#)
- [IDLanROI::SetProperty](#)
- [IDLanROI::Translate](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.3

IDLanROI Properties

IDLanROI objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLanROI::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLanROI::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLanROI::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the state of this object. State information about the object includes things like block size, type, etc., but not vertex data.

The fields in this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

BLOCK_SIZE

The number of vertices to allocate per block as needed for the region. When additional vertices are required, an additional block is allocated. The default is 100.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DATA

A $2 \times n$ or a $3 \times n$ array that defines the 2-D or 3-D vertex data, respectively. DATA is equivalent to the optional arguments, X, Y, and Z. This property is stored as double precision floating point values if the argument variable is of type DOUBLE or if the

DOUBLE property is non-zero. Otherwise it is stored as single precision floating point.

Property Type	Array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DOUBLE

A non-zero value that indicates whether or not data should be stored in this object in double precision floating point. Set this keyword to zero to indicate that the data should be stored in single precision floating point, which is the default. The DOUBLE property controls the precision used for storing the data in the AppendData, Init, and ReplaceData methods via the X, Y, and Z arguments and in SetProperty method via the DATA keyword. IDL converts any data already stored in the object to the requested precision, if necessary. Note that this keyword does not need to be set if any of the X, Y, or Z arguments or the DATA parameters are of type DOUBLE. However, setting this keyword may be desirable if the data consists of large integers that cannot be accurately represented in single precision floating point. This property is also automatically set to one if any of the X, Y or Z arguments or the DATA parameter is stored using a variable of type DOUBLE.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

INTERIOR

A Boolean variable that marks this region as an interior region (i.e., a region treated as a hole). By default, the region is treated as an exterior region.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_VERTS

An integer that contains the number of vertices currently being used by the region.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ROI_XRANGE

A two-element double-precision floating-point vector of the form $[xmin, xmax]$ that specifies the range of X data coordinates covered by the region.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ROI_YRANGE

A two-element double-precision floating-point vector of the form $[ymin, ymax]$ specifying the range of Y data coordinates covered by the region.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ROI_ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ specifying the range of Z data coordinates covered by the region.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

TYPE

An integer that indicates the type of the region. The TYPE keyword determines how computational operations, such as mask generation, are performed. Valid values include:

- 0 = points
- 1 = path
- 2 = closed polygon (the default)

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

IDLanROI::AppendData

The IDLanROI::AppendData procedure method appends vertices to the region.

Syntax

```
Obj->[IDLanROI:]AppendData, X [, Y] [, Z] [, XRANGE=variable]  
    [, YRANGE=variable] [, ZRANGE=variable]
```

Arguments

X

A vector providing the X components of the vertices to be appended. If the Y and Z arguments are not specified, X must be a two-dimensional array with the leading dimensions either 2 or 3 ([2,*] or [3,*]), in which case, X[0,*] represents the X values, X[1,*] represents the Y values, and X[2,*] represents the Z values. If the DOUBLE property is non-zero, the data is converted to double precision and is appended to the existing double precision data. Otherwise it is converted to single precision floating point and appended to the existing single precision data.

Y

A vector providing the Y components of the vertices to be appended. If the DOUBLE property is non-zero, the data is converted to double precision and is appended to the existing double precision data. Otherwise it is converted to single precision floating point and appended to the existing single precision data.

Z

A vector providing the Z components of the vertices to be appended. If the DOUBLE property is non-zero, the data is converted to double precision and is appended to the existing double precision data. Otherwise it is converted to single precision floating point and appended to the existing single precision data.

Keywords

XRANGE

Set this keyword to a named variable that upon return contains a two-element vector, $[xmin, xmax]$, representing the X range of the modification to the region. The reported range accounts for the last vertex in the region before the append occurred, as well as all vertices appended. This data is returned in double-precision floating-point.

YRANGE

Set this keyword to a named variable that upon return contains a two-element vector, $[ymin, ymax]$, representing the Y range of the modification to the region. The reported range accounts for the last vertex in the region before the append occurred, as well as all vertices appended. This data is returned in double-precision floating-point.

ZRANGE

Set this keyword to a named variable that upon return contains a two-element vector, $[zmin, zmax]$, representing the Z range of the modification to the region. The reported range accounts for the last vertex in the region before the append occurred, as well as all vertices appended. This data is returned in double-precision floating-point.

Version History

Introduced: 5.3

IDLanROI::Cleanup

The IDLanROI::Cleanup procedure method performs all cleanup for a region of interest object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLanROI::]Cleanup (In a subclass' Cleanup method only)

Arguments

None.

Keywords

None.

Version History

Introduced: 5.3

IDLanROI::ComputeGeometry

The IDLanROI::ComputeGeometry function method computes the geometrical values for area, perimeter, and/or centroid of the region.

Syntax

```
Result = Obj->[IDLanROI::]ComputeGeometry( [, AREA=variable]
      [, CENTROID=variable] [, PERIMETER=variable]
      [, SPATIAL_OFFSET=vector] [, SPATIAL_SCALE=vector] )
```

Return Value

Result

This function method returns a 1 for success, or a 0 for failure. Each computed value is returned in the *variable* name assigned to each keyword.

Arguments

None.

Keywords

AREA

Set this keyword to a named variable that upon return contains a double-precision floating-point value representing the area of the region. Interior regions (holes) return a negative area.

Note that the computed area represents the geometric area described by the region's vertex data. A pixel-based area can be computed as follows:

1. Compute a mask for the region.

```
mask = oROI->ComputeMask( )
```

2. Call IMAGE_STATISTICS to count number of samples within the mask.

```
IMAGE_STATISTICS, myImage, MASK = mask, COUNT = nSamples
```

3. Compute pixel area.

```
pixelArea = nSamples * pixelXSize * pixelYSize
```

CENTROID

Set this keyword to a named variable that upon return contains a double-precision floating-point vector $[x,y,z]$ representing the centroid for the region. If the TYPE of the region is 0 (points), the centroid is computed as the average of each of the vertices in the region. If the TYPE of the region is 1 (path), the centroid is computed as the weighted average of each of the midpoints of the lines in the region. Weights are proportional to the length of the lines. If the TYPE of the region is 2 (polygon), the centroid is computed as a weighted average of the centroids of the polygons making up the ROI (interior centroids use negative weights). Weights are proportional to the polygon area.

PERIMETER

Set this keyword to a named variable that upon return contains a double-precision floating-point value representing the perimeter of the region.

SPATIAL_OFFSET

Set this keyword to a two or three-element vector, $[tx, ty]$ or $[tx, ty, tz]$, representing the spatial calibration offset factors to be applied for the geometry calculations. The value of SPATIAL_SCALE is applied before the spatial offset values are applied. The default is $[0.0, 0.0, 0.0]$. IDL converts and maintains this value in double-precision floating-point.

SPATIAL_SCALE

Set this keyword to a two or three-element vector, $[sx, sy]$ or $[sx, sy, sz]$, representing the spatial calibration scaling factors to be applied for the geometry calculations. The spatial calibration scale is applied first, then the value of SPATIAL_OFFSET is applied. The default is $[1.0, 1.0, 1.0]$. IDL converts and maintains this value in double-precision floating-point.

Version History

Introduced: 5.3

IDLanROI::ComputeMask

The IDLanROI::ComputeMask function method prepares a two-dimensional mask for the region.

Syntax

```
Result = Obj->[IDLanROI::]ComputeMask( [, INITIALIZE={ -1 | 0 | 1 }]  
  [, DIMENSIONS=[xdim, ydim]] | [, MASK_IN=array] [, LOCATION=[x, y  
  [, z]]] [, MASK_RULE={ 0 | 1 | 2 }] [, PIXEL_CENTER=[x, y]]  
  [, PLANE_NORMAL=[x, y, z]] [, PLANE_XAXIS=[x,y,z]] [, /RUN_LENGTH] )
```

Return Value

Result

The return value is a two-dimensional array of bytes whose values range from 0 to 255. The mask is computed by applying the following formula to the current mask for each mask point contained within the ROI:

$$M_{out} = \text{MAX}(\text{MIN}(255, (M_{roi} * Ext) + M_{in}), 0)$$

where M_{roi} is 255 and Ext is 1 for points within an exterior region and -1 for points within an interior region.

If the TYPE of the region is 0 (points), a single mask pixel is set for each region vertex that falls within the bounds of the mask.

If the TYPE of the region is 1 (path), one-pixel-wide line segments are set within the mask.

If the TYPE of the region is 2 (closed polygon), a mask pixel is set if that pixel is on the plane of a region, and the pixel falls within the region (according to the MASK_RULE).

Arguments

None.

Keywords

DIMENSIONS

Set this keyword to a two-element vector, [*xdim*, *ydim*], specifying the requested dimensions of the returned mask. If MASK_IN is provided, the value of this keyword is ignored and the dimensions of that mask are used. Otherwise, the default dimensions are [100, 100].

INITIALIZE

Set this keyword to indicate how the mask should be initialized. Valid values include:

- -1 = The mask is not initialized. This option is useful when updating an already existing mask. This is the default if the MASK_IN keyword is set.
- 0 = The mask is initialized so that each pixel is set to 0. This is the default if the MASK_IN keyword is not set.
- 1 = The mask is initialized so that each pixel is set to 255.

LOCATION

Set this keyword to a vector of the form [*X*, *Y*[, *Z*]] specifying the location of the origin of the mask. The default is [0, 0, 0]. IDL converts and maintains this value in double-precision floating-point.

MASK_IN

Set this keyword to a two-dimensional array representing a mask that is already allocated and to be updated for this region. If the variable specified by this keyword is of type BYTE and is also specified as the function result, then this method updates the mask data in-place without copying. If this keyword is not provided, a mask is allocated by default to match the dimensions specified via the DIMENSIONS keyword.

MASK_RULE

Set this keyword to an integer specifying the rule used to determine whether a given pixel should be set within the mask. Valid values include:

- 0 = Boundary only. All pixels falling on a region's boundary are set.
- 1 = Interior only. All pixels falling within the region's boundary, but not on the boundary, are set.
- 2 = Boundary + Interior. All pixels falling on or within a region's boundary are set.

PIXEL_CENTER

Set this keyword to a 2-element vector, $[x, y]$, to indicate where the lower-left mask pixel is to be centered relative to a Cartesian grid. The default value is $[0.0, 0.0]$, indicating that the lower-left pixel is centered at $[0.0, 0.0]$.

PLANE_NORMAL

Set this keyword to a three-element vector, $[x, y, z]$, specifying the normal vector for the plane on which the mask is to be computed. The default is $[0, 0, 1]$.

PLANE_XAXIS

Set this keyword to a three-element vector, $[x, y, z]$, specifying the direction vector along which each row of mask pixels is to be computed (starting at LOCATION). The default is $[1, 0, 0]$.

RUN_LENGTH

Set this keyword to a non-zero value to return a run-length encoded representation of the mask, stored in a one-dimensional unsigned long array. When run-length encoded, each element with an even subscript contains the length of the run, and the following element contains the starting index of the run.

Version History

Introduced: 5.3

RUN_LENGTH keyword: 5.6

IDLanROI::ContainsPoints

The IDLanROI::ContainsPoints function method determines whether the given data coordinates are contained within the closed polygon region.

Syntax

Result = *Obj*->[IDLanROI::]ContainsPoints(*X* [, *Y* [, *Z*]])

Return Value

The return value is a vector of values, one per provided point, indicating whether that point is contained. Valid values within this return vector include:

- 0 = Exterior. The point lies strictly out of bounds of the ROI.
- 1 = Interior. The point lies strictly inside the bounds of the ROI.
- 2 = On edge. The point lies on an edge of the ROI boundary.
- 3 = On vertex. The point matches a vertex of the ROI.

A point is considered to be exterior if:

- the point falls within the boundary of an interior region (hole).
- the point does not lie in the plane of the region.
- the region TYPE property is set to 0 (points) or 1 (path).

Arguments

X

A vector providing the *X* components of the points to be tested. If the *Y* and *Z* arguments are not specified, *X* must be a two-dimensional array with the leading dimension either 2 or 3 ([2,*] or [3,*]), in which case, *X*[0,*] represents the *X* values, *X*[1,*] represents the *Y* values, and *X*[2,*] represents the *Z* values.

Y

A vector providing the *Y* components of the points to be tested.

Z

A scalar or vector providing the *Z* component(s) of the points to be tested. If not provided, the *Z* components default to 0.0.

Keywords

None.

Version History

Introduced: 5.3

IDLanROI::GetProperty

The IDLanROI::GetProperty procedure method retrieves the value of a property or group of properties for the region.

Syntax

```
Obj->[IDLanROI:]GetProperty [, PROPERTY=variable]
```

Arguments

None.

Keywords

Any property listed under “[IDLanROI Properties](#)” on page 2682 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 5.3

IDLanROI::Init

The IDLanROI::Init function method initializes a region of interest object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW( 'IDLanROI' [, X [, Y [, Z ]]] [, PROPERTY=value] )
```

or

```
Result = Obj->[IDLanROI::]Init( [X [, Y [, Z ]]] [, PROPERTY=value] )  
(Only in a subclass' Init method.)
```

Note

Keywords can be used in either form. They are omitted in the second form for brevity.

Arguments

X

A vector providing the X components of the vertices for the region. If the Y and Z arguments are not specified, X must be a two-dimensional array with the leading dimension either 2 or 3 ([2,*] or [3,*]), in which case, X[0,*] represents the X values, X[1,*] represents the Y values, and X[2,*] represents the Z values. The value for this argument is double-precision floating-point if the DOUBLE keyword is set or the inputted value is of type DOUBLE. Otherwise, it is converted to single-precision floating-point.

Y

A vector providing the *Y* components of the vertices. The value for this argument is double-precision floating-point if the `DOUBLE` keyword is set or the inputted value is of type `DOUBLE`. Otherwise, it is converted to single-precision floating-point.

Z

A scalar or vector providing the *Z* component(s) of the vertices. If not provided, *Z* values default to 0.0. The value for this argument is double-precision floating-point if the `DOUBLE` keyword is set or the inputted value is of type `DOUBLE`. Otherwise, it is converted to single-precision floating-point.

Keywords

Any property listed under [“IDLanROI Properties”](#) on page 2682 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.3

IDLanROI::RemoveData

The IDLanROI::RemoveData procedure method removes vertices from the region.

Syntax

```
Obj->[IDLanROI:]RemoveData[, COUNT=vertices] [, START=index]  
[, XRANGE=variable] [, YRANGE=variable] [, ZRANGE=variable]
```

Arguments

None.

Keywords

COUNT

Set this keyword to the number of vertices to remove. The default is one vertex.

START

Set this keyword to an index (into the region's current vertex list) where the removal is to begin. By default, the final vertex is removed.

XRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*xmin*, *xmax*], that represents the X range of the modification to the region. The reported range accounts for the vertex just before the removal (if any), the vertex just after the removal (if any), and the removed vertices. This data is returned in double-precision floating-point.

YRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [*ymin*, *ymax*], that represents the Y range of the modification to the region. The reported range accounts for the vertex just before the removal (if any), the vertex just after the removal (if any), and the removed vertices. This data is returned in double-precision floating-point.

ZRANGE

Set this keyword to a named variable that upon return contains a two-element vector, $[zmin, zmax]$, that represents the Z range of the modification to the region. The reported range accounts for the vertex just before the removal (if any), the vertex just after the removal (if any), and the removed vertices. This data is returned in double-precision floating-point.

Version History

Introduced: 5.3

IDLanROI::ReplaceData

The IDLanROI::ReplaceData procedure method replaces vertices in the region with alternate values. The number of replacement values need not match the number of values being replaced.

Syntax

```
Obj->[IDLanROI::]ReplaceData, X[, Y[, Z]] [, START=index] [, FINISH=index]  
    [, XRANGE=variable] [, YRANGE=variable] [, ZRANGE=variable]
```

Arguments

X

A vector providing the X components of the new replacement vertices. If the Y and Z arguments are not specified, X must be a two-dimensional array with the leading dimensions either 2 or 3 ([2, *] or [3, *]), in which case, X[0, *] represents the X values, X[1, *] represents the Y values, and X[2, *] represents the Z values. If the DOUBLE property is non-zero, the data is converted to double precision and is appended to the existing double precision data. Otherwise it is converted to single precision floating point and appended to the existing single precision data.

Y

A vector providing the Y components of the new replacement vertices. If the DOUBLE property is non-zero, the data is converted to double precision and is appended to the existing double precision data. Otherwise it is converted to single precision floating point and appended to the existing single precision data.

Z

A vector providing the Z components of the new replacement vertices. If the DOUBLE property is non-zero, the data is converted to double precision and is appended to the existing double precision data. Otherwise it is converted to single precision floating point and appended to the existing single precision data.

Keywords

FINISH

Set this keyword to the index of the region's current subregion vertex list where the replacement ends. If the START keyword value is ≥ 0 , the default FINISH is given by

$$\text{FINISH} = ((\text{START} + \text{N_NEW} - 1) \text{ MOD } \text{N_OLD})$$

where N_NEW is the number of replacement vertices provided via the [X, Y, Z] arguments and N_OLD is the number of vertices (prior to replacement) in the current subregion.

If the START keyword is not set or is negative, the default FINISH is given by

$$\text{FINISH} = \text{N_OLD} - 1$$

FINISH may be less than START in which case the vertices, including and following START and the vertices preceding and including FINISH, are replaced with the new values.

START

Set this keyword to an index of the region's current subregion vertex list where the replacement begins. If the FINISH keyword value is ≥ 0 , the default START is given by

$$\text{START} = ((\text{FINISH} - \text{N_NEW} + 1) \text{ MOD } \text{N_OLD})$$

where N_NEW is the number of replacement vertices provided via the [X, Y, Z] arguments and N_OLD is the number of vertices (prior to replacement) in the current subregion.

If the FINISH keyword is not set (or negative), the default START is clamped to 0 and is given by

$$\text{N_OLD} - \text{N_NEW}$$

XRANGE

Set this keyword to a named variable that upon return contains a two-element vector, [xmin, xmax], representing the X range of the modification to the region. The reported range accounts for the replaced vertices, the vertex just before the replacement (if any), the vertex just after the replacement (if any), and the new replacement vertices. This data is returned in double-precision floating-point.

YRANGE

Set this keyword to a named variable that upon return contains a two-element vector, $[ymin, ymax]$, representing the Y range of the modification to the region. The reported range accounts for the replaced vertices, the vertex just before the replacement (if any), the vertex just after the replacement (if any), and the new replacement vertices. This data is returned in double-precision floating-point.

ZRANGE

Set this keyword to a named variable that upon return contains a two-element vector, $[zmin, zmax]$, representing the Z range of the modification to the region. The reported range accounts for the replaced vertices, the vertex just before the replacement (if any), the vertex just after the replacement (if any), and the new replacement vertices. This data is returned in double-precision floating-point.

Version History

Introduced: 5.3

IDLanROI::Rotate

The IDLanROI::Rotate procedure method modifies the vertices for the region by applying a rotation.

Syntax

Obj->[IDLanROI::]Rotate, *Axis*, *Angle* [, CENTER=[*x*, *y*, *z*]]

Arguments

Axis

A three-element vector of the form [*x*, *y*, *z*] describing the axis about which the region is to be rotated.

Angle

The angle, measured in degrees, by which the rotation is to occur.

Keywords

CENTER

Set this keyword to a two or three-element vector of the form [*x*, *y*], or [*x*, *y*, *z*] specifying the center of rotation. The default is [0, 0, 0]. IDL converts and applies this data in double-precision floating-point.

Version History

Introduced: 5.3

IDLanROI::Scale

The IDLanROI::Scale procedure method modifies the vertices for the region by applying a scale.

Syntax

Obj->[IDLanROI::]Scale, *Sx*[, *Sy*[, *Sz*]]

Arguments

Sx

The X scale factor. If the *Sy* and *Sz* arguments are not specified, *Sx* must be a two or three-element vector, in which case *Sx*[0] represents the scale in X, *Sx*[1] represents the scale in Y, *Sx*[2] represents the scale in Z. IDL converts and applies this data in double-precision floating-point.

Sy

The Y scale factor. IDL converts and applies this data in double-precision floating-point.

Sz

The Z scale factor. IDL converts and applies this data in double-precision floating-point.

Keywords

None.

Version History

Introduced: 5.3

IDLanROI:: SetProperty

The IDLanROI::SetProperty procedure method sets the value of a property or group of properties for the region.

Syntax

Obj->[IDLanROI::]SetProperty

Arguments

None.

Keywords

Any property listed under “[IDLanROI Properties](#)” on page 2682 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.3

IDLanROI::Translate

The IDLanROI::Translate procedure method modifies the vertices for the region by applying a translation.

Syntax

Obj->[IDLanROI::]Translate, *Tx*[, *Ty*[, *Tz*]]

Arguments

Tx

The X translation factor. If the *Ty* and *Tz* arguments are not specified, *Tx* must be a two or three-element vector, in which case *Tx*[0] represents translation in X, *Tx*[1] represents translation in Y, *Tx*[2] represents translation in Z. IDL converts and applies this data in double-precision floating-point.

Ty

The Y translation factor. IDL converts and applies this data in double-precision floating-point.

Tz

The Z translation factor. IDL converts and applies this data in double-precision floating-point.

Keywords

None.

Version History

Introduced: 5.3

IDLanROIGroup

The IDLanROIGroup object class is an analytical representation of a group of regions of interest.

Superclasses

This class is a subclass of [IDL_Container](#).

Creation

See [IDLanROIGroup::Init](#).

Properties

Objects of this class have the following properties. See “[IDLanROIGroup Properties](#)” on page 2710 for details on individual properties.

- [ALL](#)
- [ROIGROUP_XRANGE](#)
- [ROIGROUP_YRANGE](#)
- [ROIGROUP_ZRANGE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

The IDLanROIGroup class has the following methods:

- [IDLanROIGroup::Add](#)
- [IDLanROIGroup::Cleanup](#)
- [IDLanROIGroup::ComputeMask](#)
- [IDLanROIGroup::ComputeMesh](#)
- [IDLanROIGroup::ContainsPoints](#)
- [IDLanROIGroup::GetProperty](#)
- [IDLanROIGroup::Init](#)
- [IDLanROIGroup::Rotate](#)

- [IDLanROIGroup::Scale](#)
- [IDLanROIGroup::Translate](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.3

IDLanROIGroup Properties

IDLanROIGroup objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via “[IDLanROIGroup::GetProperty](#)” on page 2721. Properties with the word “Yes” in the “Init” column of the property table can be set via “[IDLanROIGroup::Init](#)” on page 2722.

Note

For a discussion of the property description tables shown below, see “[About Object Property Descriptions](#)” on page 2671.

ALL

An anonymous structure with the values of all of the properties associated with the state of this object.

The fields in this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ROIGROUP_XRANGE

A two-element double-precision floating-point vector of the form [*xmin*, *xmax*] specifying the range of X data coordinates covered by the region.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ROIGROUP_YRANGE

A two-element double-precision floating-point vector of the form $[ymin, ymax]$ specifying the range of Y data coordinates covered by the region.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ROIGROUP_ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ specifying the range of Z data coordinates covered by the region.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLanROIGroup::Add

The IDLanROIGroup::Add procedure method adds a region to the region group. Only objects of the IDLanROI class may be added to the group. The regions in the group must all be of the same type: all points, all paths, or all polygons.

Syntax

Obj->[IDLanROIGroup::]Add, *ROI*

Arguments

ROI

A reference to an instance of the IDLanROI object class representing the region of interest to be added to the group.

Keywords

Accepts all keywords accepted by the [IDL_Container::Add](#) method.

Version History

Introduced: 5.3

IDLanROIGroup::Cleanup

The IDLanROIGroup::Cleanup procedure method performs all cleanup for a region of interest group object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLanROIGroup::]Cleanup (In a subclass' Cleanup method only.)

Arguments

None.

Keywords

None.

Version History

Introduced: 5.3

IDLanROIGroup::ComputeMask

The IDLanROIGroup::ComputeMask function method prepares a two-dimensional mask for this group of regions.

Syntax

```
Result = Obj->[IDLanROIGroup:]ComputeMask( [, INITIALIZE={ -1 | 0 | 1 } ]
  [, DIMENSIONS=[xdim, ydim]] | [, MASK_IN=array] [, LOCATION=[x, y
  [, z]]] [, MASK_RULE={ 0 | 1 | 2 } ] [, PLANE_NORMAL=[x, y, z]]
  [, PLANE_XAXIS=[x, y, z]] [, /RUN_LENGTH ] )
```

Return Value

Result

The return value is a two-dimensional array of bytes whose values range from 0 to 255. The mask is computed by applying the following formula to the current mask for each mask point contained within the ROI:

$$M_{out} = \text{MAX}(\text{MIN}(0, (M_{roi} * Ext) + M_{in}), 255)$$

where M_{roi} is 255 and Ext is 1 for points within an exterior region and -1 for points within an interior region.

If the TYPE of the contained regions is 0 (points), a single mask pixel is set for each region vertex that falls within the bounds of the mask.

If the TYPE of the contained regions is 1 (path), each pixel along the paths of the regions is set if it falls within the mask.

If the TYPE of the region is 2 (closed polygon), a mask pixel is set if that pixel is on the plane of a contained region, and the pixel falls within that region (according to the MASK_RULE).

Arguments

None.

Keywords

DIMENSIONS

Set this keyword to a two-element vector, $[xdim, ydim]$, specifying the requested dimensions of the returned mask. If `MASK_IN` is provided, the value of this keyword is ignored, and the dimensions of that mask are used. Otherwise, the default dimensions are $[100, 100]$.

INITIALIZE

Set this keyword to indicate how the mask should be initialized. Valid values include:

- `-1` = The mask is not initialized; the default if the `MASK_IN` keyword is set. This option is useful when updating an already existing mask.
- `0` = The mask is initialized with each pixel set to 0; the default if the `MASK_IN` keyword is not set.
- `1` = The mask is initialized with each pixel set to 255.

LOCATION

Set this keyword to a vector of the form $[X, Y, Z]$ specifying the location of the origin of the mask. The default is $[0, 0, 0]$.

MASK_IN

Set this keyword to a two-dimensional array representing a mask that is already allocated and to be updated for this region. If the variable specified by this keyword is of type `BYTE` and is also specified as the function result, then this method updates the mask data in-place without copying. If this keyword is not provided, a mask is allocated by default to match the dimensions specified via the `DIMENSIONS` keyword.

MASK_RULE

Set this keyword to an integer specifying the rule used to determine whether a given pixel should be set within the mask. Valid values include:

- `0` = Boundary Only. All pixels falling on a region's boundary are set.
- `1` = Interior Only. All pixels falling within the region's boundary, but not on the boundary, are set.
- `2` = Boundary + Interior. All pixels falling on or within a region's boundary are set.

PLANE_NORMAL

Set this keyword to a three-element vector, $[x, y, z]$, specifying the normal vector for the plane on which the mask is to be computed. The default is $[0, 0, 1]$.

PLANE_AXIS

Set this keyword to a three-element vector, $[x, y, z]$, specifying the direction vector along which each row of mask pixels is to be computed (starting at LOCATION). The default is $[1, 0, 0]$.

RUN_LENGTH

Set this keyword to a non-zero value to return a run-length encoded representation of the mask, stored in a one-dimensional unsigned long array. When run-length encoded, each element with an even subscript contains the length of the run, and the following element contains the starting index of the run.

Version History

Introduced: 5.3

RUN_LENGTH keyword: 5.6

IDLAnROIGroup::ComputeMesh

The IDLAnROIGroup::ComputeMesh function method triangulates a surface mesh with optional capping from the stack of regions contained within this group.

Note

The contained regions may be concave. However, this method will fail under the following conditions:

- The region group contains fewer than two regions.
 - The TYPE property of the contained regions is 0 (points) or 1 (path).
 - Any of the contained regions are not simple (i.e., a region is self-intersecting).
 - The region group contains interior regions (holes).
 - More than one region lies on the same plane (i.e., the region group contains branches).
-

Each region pair is normalized by perimeter and the triangulation is computed by walking the contours in parallel, keeping the normalized progress along each contour in sync. The returned triangulation minimizes the mesh surface area. Each vertex may appear only once in the output, and the resulting polygon mesh is solid with outward facing normals computed via the right-hand rule. If capping is requested, it is computed using the [IDLgrTessellator](#) on the top and bottom regions, and/or the regions on either side of an inter-slice gap.

Syntax

```
Result = Obj->[IDLAnROIGroup::]ComputeMesh( Vertices, Conn  
[, CAPPED={ 0 | 1 | 2}] [, SURFACE_AREA=variable] )
```

Return Value

Result

The return value of this function method is the number of triangles generated if the surface mesh triangulation is successful, or zero if unsuccessful.

Arguments

Vertices

An output [3, n] array of vertices. If all regions in the group are defined with single precision vertices (DOUBLE property is zero), then IDL returns a single precision floating point array. Otherwise, if any of the regions in the group are defined with double precision vertices (DOUBLE property is non-zero), then IDL returns a double precision floating point array.

Conn

An output polygon mesh connectivity array.

Keywords

CAPPED

Set this keyword to a value to indicate whether flat caps are to be computed at the top-most or bottom-most regions (as selected by a counter-clockwise rule), or at the regions on either side of an inter-slice gap. The value of this keyword is a bit-wise OR of the values shown below. For example, to cap the top-most and bottom-most regions only, set the CAPPED keyword to 3. The default is 0 (no caps).

- 0 = no caps
- 1 = cap the top-most region
- 2 = cap the bottom-most region

SURFACE_AREA

Set this keyword to a named variable that upon return contains the overall surface area of the computed triangulation. This value was minimized in the computation of the triangulation. IDL returns this value in a double-precision floating-point variable.

Version History

Introduced: 5.3

IDLanROIGroup::ContainsPoints

The IDLanROIGroup::ContainsPoints function method determines whether the given points (in data coordinates) are contained within this region group.

The regions within this group must have a TYPE of 2 (closed polygon) and must fall on parallel planes for successful containment testing to occur.

For each point to be tested:

- If the point lies directly on one of the region planes, it is tested for containment within each of the regions that fall on that plane.
- If the point lies between two of the region planes, it is projected onto the nearest region plane, and tested for containment within each of the regions on that plane.
- If the point lies above or below the stack of parallel region planes, the point will be considered to be exterior to the region group.

On a given plane, a point will be considered to be exterior if either of the following conditions are true:

- The point does not fall within any of the regions on that plane.
- The point falls within as many or more holes than non-hole regions on that plane.

Syntax

Result = *Obj*->[IDLanROIGroup::]ContainsPoints(*X*[, *Y*[, *Z*]])

Return Value

The return value is a vector of values, one per provided point, indicating whether that point is contained. Valid values within this return vector include:

- 0 = Exterior. The point lies strictly outside the bounds of the ROI.
- 1 = Interior. The point lies strictly inside the bounds of the ROI.
- 2 = On Edge. The point lies on an edge of the ROI boundary.
- 3 = On Vertex. The point matches a vertex of the ROI.

Arguments

X

A vector providing the X components of the points to be tested. If the Y and Z arguments are not specified, X must be a two-dimensional array with the leading dimension either 2 or 3 ($[2,*]$ or $[3,*]$), in which case, $X[0,*]$ represents the X values, $X[1,*]$ represents the Y values, and $X[2,*]$ represents the Z values.

Y

A vector providing the Y components of the points to be tested.

Z

A scalar or vector providing the Z components of the points to be tested. If not provided, the Z components default to 0.0.

Keywords

None.

Version History

Introduced: 5.3

IDLanROIGroup::GetProperty

The IDLanROIGroup::GetProperty procedure method retrieves the value of a property or group of properties for the region group.

Syntax

Obj->[IDLanROIGroup::]GetProperty [, *PROPERTY=value*]

Arguments

None.

Keywords

Any property listed under “[IDLanROIGroup Properties](#)” on page 2710 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 5.3

IDLanROIGroup::Init

The IDLanROIGroup::Init function method initializes a region of interest group object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLanROIGroup')

or

Result = *Obj*->[IDLanROIGroup::]Init() (*Only in a subclass' Init method.*)

Arguments

None.

Keywords

Any property listed under “[IDLanROIGroup Properties](#)” on page 2710 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.3

IDLanROIGroup::Rotate

The IDLanROIGroup::Rotate procedure method modifies the vertices for all regions within the group by applying a rotation.

Syntax

Obj->[IDLanROIGroup::]Rotate, *Axis*, *Angle*[, CENTER=[*x*, *y* [, *z*]]]

Arguments

Axis

A three-element vector of the form [*x*, *y*, *z*] describing the axis about which the region group is to be rotated.

Angle

The angle, measured in degrees, by which to rotate the ROI group.

Keywords

CENTER

Set this keyword to a two or three-element vector of the form [*x*, *y*], or [*x*, *y*, *z*] specifying the center of rotation. The default is [0, 0, 0]. IDL converts and applies this data in double-precision floating-point.

Version History

Introduced: 5.3

IDLanROIGroup::Scale

The IDLanROIGroup::Scale procedure method modifies the vertices for the region by applying a scale.

Syntax

```
Obj->[IDLanROIGroup::]Scale, Sx[, Sy[, Sz]]
```

Arguments

Sx

The X scale factor. If the Sy and Sz arguments are not specified, Sx must be a two or three-element vector, in which case Sx[0] represents the scale in X, Sx[1] represents the scale in Y, Sx[2] represents the scale in Z. IDL converts and applies this data in double-precision floating-point.

Sy

The Y scale factor. IDL converts and applies this data in double-precision floating-point.

Sz

The Z scale factor. IDL converts and applies this data in double-precision floating-point.

Keywords

None.

Version History

Introduced: 5.3

IDLanROIGroup::Translate

The IDLanROIGroup::Translate procedure method modifies the vertices of all regions within the group by applying a translation.

Syntax

Obj->[IDLanROIGroup::]Translate, *Tx*[, *Ty*[, *Tz*]]

Arguments

Tx

The *X* translation factor. If the *Ty* and *Tz* arguments are not specified, *Tx* must be a two or three-element vector, in which case *Tx*[0] represents translation in *X*, *Tx*[1] represents translation in *Y*, *Tx*[2] represents translation in *Z*. IDL converts and applies this data in double-precision floating-point.

Ty

The *Y* translation factor. IDL converts and applies this data in double-precision floating-point.

Tz

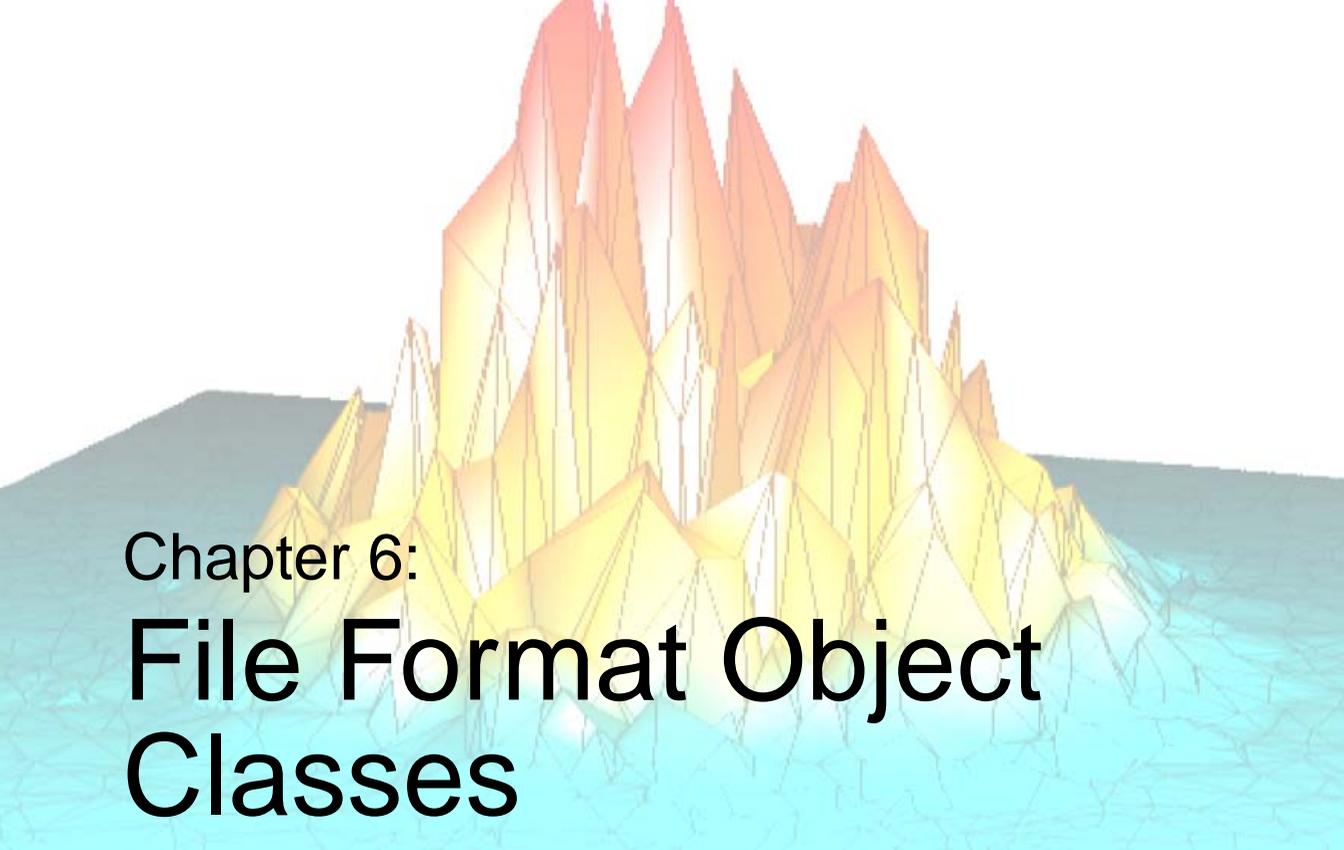
The *Z* translation factor. IDL converts and applies this data in double-precision floating-point.

Keywords

None.

Version History

Introduced: 5.3



Chapter 6: File Format Object Classes

This chapter describes IDL's built-in file format class library.

IDLffDICOM	2728	IDLffMrSID	2835
IDLffDXF	2761	IDLffShape	2849
IDLffJPEG2000	2790	IDLffXMLDOM Classes	2887
IDLffLangCat	2821	IDLffXMLSAX	3039

IDLffDICOM

An IDLffDICOM object contains the data for one or more images embedded in a DICOM Part 10 file. The API to the IDLffDICOM object provides accessor methods to the basic data elements of a DICOM file, namely the group/element tag, value representation, length, and data values. Additional methods deal with the file header preamble, data dictionary description for individual elements, and embedded sequences of elements. Most methods take a DICOM group/element tag as a parameter. An alternative parameter to the DICOM tag in some methods is the reference. A reference value is a LONG integer that is unique to each element in the DICOM object. This value can be used to directly access a specific element and to differentiate between elements in the DICOM file that have the same group/element tag. Valid reference values are always positive.

See [“IDL DICOM v3.0 Conformance Summary”](#) on page 2730 for information regarding IDL DICOM file reading support.

Note

The IDLffDicomEx object greatly expands on the capabilities of the IDLffDICOM object and requires an additional license key to access the functionality. The IDLffDicomEx object includes methods for reading and writing pixel data to a DICOM file, and specifying the data compression. You can also add, modify, or remove public and private DICOM attribute tags, public and private sequences, and sets of repeating tags within sequences. See [Medical Imaging in IDL](#) for details.

Superclasses

This class has no superclasses.

Creation

See [“IDLffDICOM::Init”](#) on page 2757.

Properties

Objects of this class have the following properties. See [“IDLffDICOM Properties”](#) on page 2734 for details on individual properties.

- [VERBOSE](#)

Methods

This class has the following methods:

- [IDLffDICOM::Cleanup](#)
- [IDLffDICOM::DumpElements](#)
- [IDLffDICOM::GetChildren](#)
- [IDLffDICOM::GetDescription](#)
- [IDLffDICOM::GetElement](#)
- [IDLffDICOM::GetGroup](#)
- [IDLffDICOM::GetLength](#)
- [IDLffDICOM::GetParent](#)
- [IDLffDICOM::GetPreamble](#)
- [IDLffDICOM::GetReference](#)
- [IDLffDICOM::GetValue](#)
- [IDLffDICOM::GetVR](#)
- [IDLffDICOM::Init](#)
- [IDLffDICOM::Read](#)
- [IDLffDICOM::Reset](#)

Version History

Introduced: 5.2

IDL DICOM v3.0 Conformance Summary

Introduction

This section is an abbreviated DICOM conformance statement for IDL, and specifies the compliance of RSI IDL DICOM file reading support to the DICOM v3.0 standard. As described in the DICOM Standard PS 3.2 (Conformance), the purpose of this document is to outline the level of conformance to the DICOM standard and to enumerate the supported DICOM Service Classes, Information Objects, and Communications Protocols supported by this implementation.

IDL does not contain or support any of the DICOM services such as Storage, Query/Retrieve, Print, Verification, etc., so there will be no conformance claims relating to these services and no mention of any Application Entities for these services. Communications Protocol profiles will also be absent from this document for the same reasons. The remainder of this document will describe how IDL handles the various Information Objects it is capable of reading.

Reading of DICOM Part 10 files

IDL supports reading files that conform to the DICOM Standard PS 3.10 DICOM File Format. This format provides a means to encapsulate in a file the Data Set representing a SOP (Service Object Pair) Instance related to a DICOM IOD (Information Object Definition). Files written to disk in this DICOM File Format will be referred to as DICOM Part 10 files for the remainder of this document. Note that IDL does NOT support the writing of files in this DICOM File Format, only reading.

Encapsulated Transfer Syntaxes Supported

IDL supports reading DICOM Part 10 files whose contents have been written using the following Transfer Syntaxes. The Transfer Syntax UID is in the file's DICOM Tag field (0002,0010).

UID Value	UID Name
1.2.840.10008.1.2	Implicit VR Little Endian: Default Transfer Syntax for DICOM
1.2.840.10008.1.2.1	Explicit VR Little Endian
1.2.840.10008.1.2.2	Explicit VR Big Endian

Table 6-1: Encapsulated Transfer Syntaxes Supported

Encapsulated Transfer Syntaxes NOT Supported

IDL does NOT support reading DICOM Part 10 files whose contents have compressed data that has been written using the following Transfer Syntaxes. IDL will NOT be able to access the data element (DICOM Tag field (7FE0,0010)) of files with these types of compressed data. The Transfer Syntax UID is in the file's DICOM Tag field (0002,0010).

UID Value	UID Name
1.2.840.10008.1.2.4.50	JPEG Baseline (Process 1): Default Transfer Syntax for Lossy JPEG 8 Bit Image Compression
1.2.840.10008.1.2.4.51	JPEG Extended (Process 2 & 4): Default Transfer Syntax for Lossy JPEG 12 Bit Image Compression (Process 4 only)
1.2.840.10008.1.2.4.52	JPEG Extended (Process 3 & 5)
1.2.840.10008.1.2.4.53	JPEG Spectral Selection, Non-Hierarchical (Process 6 & 8)
1.2.840.10008.1.2.4.54	JPEG Spectral Selection, Non-Hierarchical (Process 7 & 9)
1.2.840.10008.1.2.4.55	JPEG Full Progression, Non-Hierarchical (Process 10 & 12)
1.2.840.10008.1.2.4.56	JPEG Full Progression, Non-Hierarchical (Process 11 & 13)
1.2.840.10008.1.2.4.57	JPEG Lossless, Non-Hierarchical (Process 14)
1.2.840.10008.1.2.4.58	JPEG Lossless, Non-Hierarchical (Process 15)
1.2.840.10008.1.2.4.59	JPEG Extended, Hierarchical (Process 16 & 18)
1.2.840.10008.1.2.4.60	JPEG Extended, Hierarchical (Process 17 & 19)
1.2.840.10008.1.2.4.61	JPEG Spectral Selection, Hierarchical (Process 20 & 22)
1.2.840.10008.1.2.4.62	JPEG Spectral Selection, Hierarchical (Process 21 & 23)
1.2.840.10008.1.2.4.63	JPEG Full Progression, Hierarchical (Process 24 & 26)
1.2.840.10008.1.2.4.64	JPEG Full Progression, Hierarchical (Process 25 & 27)
1.2.840.10008.1.2.4.65	JPEG Lossless, Hierarchical (Process 28)

Table 6-2: Encapsulated Transfer Syntaxes NOT Supported

UID Value	UID Name
1.2.840.10008.1.2.4.66	JPEG Lossless, Hierarchical (Process 29)
1.2.840.10008.1.2.4.70	JPEG Lossless, Non-Hierarchical, First-Order Prediction (Process 14 [Selection Value 1]): Default Transfer Syntax for Lossless JPEG Image Compression
1.2.840.10008.1.2.5	RLE Lossless

Table 6-2: Encapsulated Transfer Syntaxes NOT Supported (Continued)

Encapsulated SOP Classes Supported

IDL supports reading DICOM Part 10 files whose contents encapsulate the data of the following SOP Classes. The SOP Class UID is in the file's DICOM Tag field (0008,0016).

UID Value	UID Name
1.2.840.10008.5.1.4.1.1.1	CR Image Storage
1.2.840.10008.5.1.4.1.1.2	CT Image Storage
1.2.840.10008.5.1.4.1.1.4	MR Image Storage
1.2.840.10008.5.1.4.1.1.6.1	Ultrasound Image Storage
1.2.840.10008.5.1.4.1.1.7	Secondary Capture Image Storage
1.2.840.10008.5.1.4.1.1.12.1	X-Ray Angiographic Image Storage
1.2.840.10008.5.1.4.1.1.12.2	X-Ray Radiofluoroscopic Image Storage
1.2.840.10008.5.1.4.1.1.20	Nuclear Medicine Image Storage
1.2.840.10008.5.1.4.1.1.128	Positron Emission Tomography Image Storage

Table 6-3: Encapsulated SOP Classes Supported

Handling of odd length data elements

The DICOM Standard PS 3.5 (Data Structures and Encoding) specifies that the data element values which make up a DICOM data stream must be padded to an even length. The toolkit upon which IDL's DICOM reading functionality is built strictly enforces this specification. If IDL encounters an incorrectly formed odd length data field while reading a DICOM Part 10 file it will report an error and stop the reading process.

Handling of undefined VRs

The VR (Value Representation) of a data element describes the data type and format of that data element's values. If IDL encounters an undefined VR while reading a DICOM Part 10 file, it will set that data element's VR to be UN (unknown).

Handling of retired and private data elements

Certain data elements are no longer supported under the v3.0 of the DICOM standard and are denoted as retired. Also, some DICOM implementations may require the communication of information that cannot be contained in standard data elements, and thus create private data elements to contain such information. Retired and private data elements should pose no problem to IDL's DICOM Part 10 file reading capability. When IDL encounters a retired or private data element tag during reading a DICOM Part 10 file, it will treat it just like any standard data element: read the data value and allow it to be accessed via the `IDLffDICOM::GetValue` method.

IDLffDICOM Properties

IDLffDICOM objects have the following properties. Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLffDICOM::Init](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

VERBOSE

A Boolean value that indicates whether informational messages are printed to the Output Log during the operational life of the object. If this value is true, the informational messages are printed to the Output Log. Otherwise, the messages are not printed.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLffDICOM::Cleanup

The IDLffDICOM::Cleanup procedure method destroys the IDLffDICOM object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffDICOM::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Examples

```
; Create a DICOM object, read a DICOM file and dump its contents:
obj = OBJ_NEW( 'IDLffDICOM' )
var = obj->Read(DIALOG_PICKFILE(FILTER="*"))
obj->DumpElements
OBJ_DESTROY, obj

; Executing this statement should produce an invalid object
; reference error since obj no longer exists:
obj->DumpElements
```

Version History

Introduced: 5.2

IDLffDICOM::DumpElements

The IDLffDICOM::DumpElements procedure method dumps a description of the DICOM data elements of the IDLffDICOM object to the screen or to a file.

Syntax

```
Obj->[IDLffDICOM::]DumpElements [, Filename]
```

Arguments

Filename

A scalar string containing the full path and filename of the file to which to dump the elements. The file is written as ASCII text.

Keywords

None

Examples

The columns output by DumpElements are the element reference, the (group, element) tuple, the value representation, the description, the value length, and some of the data values.

```
; Create a DICOM object, read a DICOM file and dump its contents:
obj = OBJ_NEW( 'IDLffDICOM' )
var  = obj->Read(DIALOG_PICKFILE(FILTER='*'))
obj->DumpElements

; Dump the contents of the current DICOM object to a file under
; Windows:
obj->DumpElements, 'c:\rsi\elements.dmp'

; Dump the contents of the current DICOM object to a file under
; UNIX:
obj->DumpElements, '/rsi/elements.dmp'

OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetChildren

The IDLffDICOM::GetChildren function method is used to find the member element references of a DICOM sequence. It takes as an argument a scalar reference to a DICOM element representing the parent of the sequence.

Syntax

Result = *Obj*->[IDLffDICOM::]GetChildren(*Reference*)

Return Value

Returns an array of references to the elements of the object that are members of that sequence. The scalar parent reference is possibly obtained by a previous call to GetReference or any method that generates a reference list. Any member of a sequence may also itself be the parent of another sequence. If the scalar reference argument is not the parent of a sequence, the method returns -1.

Arguments

Reference

A scalar reference to a DICOM element that is known to be the parent of a DICOM sequence.

Keywords

None

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get a list of references to all elements that are sequences:
refs = obj->GetReference(VR='SQ')

; Cycle through the returned list and print out the immediate
; children references and descriptions of each sequence:
FOR i = 0, N_ELEMENTS(refs)-1 DO BEGIN
    IF (refs[i] NE -1) THEN $
        BEGIN
            children = obj->GetChildren(refs[i])
```

```
FOR j = 0, N_ELEMENTS(children)-1 DO $
  BEGIN
    PRINT, children[j]
    PRINT, obj->GetDescription(REFERENCE=children[j])
  ENDFOR
ENDIF
ENDFOR
OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetDescription

The IDLffDICOM::GetDescription function method takes optional DICOM group and element arguments and returns an array of STRING descriptions.

Syntax

```
Result = Obj->[IDLffDICOM::]GetDescription([Group [, Element]]  
[, REFERENCE=list of element references])
```

Return Value

Returns an array of strings describing the field's contents as per the data dictionary in the DICOM specification PS 3.6. If no arguments or keywords are specified, the returned array contains the descriptions for all elements in the object. The effect of multiple keywords and parameters is to AND their results. If no DICOM elements can be found matching the search criteria, -1 will be returned.

Arguments

Group

An optional argument representing the value for the DICOM group to search for, i.e. '0018'x. If this argument is omitted, then all of the DICOM array elements are returned.

Element

An optional argument specified only if the *Group* argument has also been specified. Set this argument to the value for the DICOM element for which to search, such as '0010'x. If this argument is omitted and the *Group* argument was specified, then all elements of the specified *Group* are returned.

Keywords

REFERENCE

Set this keyword to a list of element reference values from which to return description values.

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get the description of the patient name element:
arr = obj->GetDescription('0010'x,'0010'x)
PRINT, arr

; Get array of all of the descriptions from the patient info group:
arr = obj->GetDescription('0010'x)
FOR i = 0, N_ELEMENTS(arr)-1 DO BEGIN
    PRINT, arr[i]
ENDFOR

OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetElement

The IDLffDICOM::GetElement function method takes optional DICOM group and/or element arguments and returns an array of DICOM Element numbers for those parameters.

Syntax

```
Result = Obj->[IDLffDICOM::]GetElement([Group [, Element]]  
[, REFERENCE=list of element references])
```

Return Value

Returns an array of integers representing the DICOM Element numbers for the parameters of the *Group* and *Element* arguments. If no arguments or keywords are specified, the returned array contains Element numbers for all elements in the object. The effect of multiple keywords and parameters is to AND their results. If no matching elements can be found, the function returns -1.

Arguments

Group

An optional argument representing the value for the DICOM group to search for, i.e. '0018'x. If this argument is omitted, then all of the DICOM array elements are returned.

Element

An optional argument specified only if the Group argument has also been specified. Set this argument to the value for the DICOM element to search for, such as '0010'x. If this argument is omitted and the Group argument was specified, then all elements of the specified Group are returned.

Keywords

REFERENCE

Set this keyword to a list of element reference values from which to return element number values.

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get references to all elements with "patient" in the description:
refs = obj->GetReference(DESCRIPTION='patient')

; Get the element numbers of the elements containing "patient":
FOR i = 0, N_ELEMENTS(refs)-1 DO BEGIN
    num = obj->GetElement(REFERENCE=refs[i])
    PRINT,num
ENDFOR

; Get the element numbers from the Patient Info group, 0010:
elements = obj->GetElement('0010'x)
PRINT, elements

OBJ_DESTROY,obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetGroup

The IDLffDICOM::GetGroup function method takes optional DICOM group and/or element arguments and returns an array of DICOM Group numbers for those parameters.

Syntax

```
Result = Obj->[IDLffDICOM::]GetGroup([Group[, Element]]  
[, REFERENCE=list of element references])
```

Return Value

Returns an array of integers representing the DICOM Group numbers for *Group* parameters. If no arguments or keywords are specified, the returned array contains Group numbers for all groups in the object. The effect of multiple keywords and parameters is to AND their results. If no matching elements can be found, the function returns -1.

Arguments

Group

An optional argument representing the value for the DICOM group for which to search, such as '0018'x. If this argument is omitted, then all of the DICOM array elements are returned.

Element

An optional argument specified only if the *Group* argument has also been specified. Set this to the value for the DICOM element for which to search, such as '0010'x. If this argument is omitted and the *Group* argument was specified, then all elements of the specified *Group* are returned.

Keywords

REFERENCE

Set this keyword to a list of element references from which to return group number values.

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get references to all elements with "patient" in the description:
refs = obj->GetReference(DESCRIPTION='patient')

; Get the group numbers of the elements containing "patient":
FOR i = 0, N_ELEMENTS(refs)-1 DO BEGIN
    num = obj->GetGroup(REFERENCE=refs[i])
    PRINT, num
ENDFOR

; Get the group numbers from the Patient Info group, 0010:
grp = obj->GetGroup('0010'x)
PRINT, grp

OBJ_DESTROY,obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetLength

The IDLffDICOM::GetLength function method takes optional DICOM group and/or element arguments and returns an array of LONGs.

Syntax

Result = Obj->[IDLffDICOM::]GetLength([Group [, Element]] [, REFERENCE=list of element references])

Return Value

Returns an array of longword integers. The length is the field length that explicitly exists in the DICOM file, and represents the length of the element value in bytes. If no arguments or keywords are specified, the returned array contains the lengths for all elements in the object. The effect of multiple keywords and parameters is to AND their results. If no matching elements can be found, the function returns -1.

Arguments

Group

An optional argument representing the value for the DICOM group for which to search, such as '0018'x. If this argument is omitted, all DICOM array elements are returned.

Element

An optional argument specified only if the *Group* argument has also been specified. Set this to the value for the DICOM element for which to search, such as '0010'x. If this argument is omitted and the *Group* argument was specified, then all elements of the specified *Group* are returned.

Keywords

REFERENCE

Set this keyword to a list of element references from which to return length values.

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get the length of the patient name element:
arr = obj->GetLength('0010'x,'0010'x)
PRINT, arr

; Get an array of all of the lengths from the patient info group:
arr = obj->GetLength('0010'x)
PRINT, arr
OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetParent

The IDLffDICOM::GetParent function method is used to find the parent references of a set of elements in a DICOM sequence.

Syntax

Result = *Obj*->[IDLffDICOM::]GetParent(*ReferenceList*)

Return Value

Returns the parent references of a set of elements in a DICOM sequence. It takes as an argument an array of references that represent DICOM elements. If no members of the *ReferenceList* are members of a sequence, a -1 is returned, and for each member of the *ReferenceList* which is not a member of a sequence, a -1 is returned.

Arguments

ReferenceList

An array of references to DICOM elements that are known to be members of a DICOM sequence.

Keywords

None

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER = '*'))

; Get the reference to the Referenced Study Sequence
; element, if it exists:
ref = obj->GetReference('0008'x, '1110'x)
PRINT, ref
PRINT, obj->GetDescription(REFERENCE = ref)

; Get and print the parent sequence, if it exists.
; This should result in a -1 since this element is not
; a member of a sequence:
parent = obj->GetParent(ref)
PRINT, parent
```

```
PRINT, obj->GetDescription(REFERENCE=parent)

; Get the children of the Referenced Study Sequence
; element, if it exists:
refs = obj->GetChildren(ref[0])
PRINT, refs
PRINT, obj->GetDescription(REFERENCE = refs)
OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetPreamble

The IDLffDICOM::GetPreamble function method returns the preamble of a DICOM v3.0 Part 10 file.

Syntax

Result = *Obj*->[IDLffDICOM::]GetPreamble()

Return Value

Returns a 128-element byte array containing the preamble, which is a fixed 128 byte field available for implementation specified usage. If it is not used by the implementor of the file, it will be set to all zeroes.

Arguments

None

Keywords

None

Examples

```
; Create a DICOM object, read a DICOM file:
obj = OBJ_NEW('IDLffDICOM')
var = obj->Read(DIALOG_PICKFILE(FILTER = '*'))

; Get an array of the byte contents of the DICOM file preamble:
arr = obj->GetPreamble()
PRINT, arr

OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetReference

The IDLffDICOM::GetReference function method takes optional DICOM group and/or element arguments and returns an array of references to matching elements in the object.

Syntax

```
Result = Obj->[IDLffDICOM::]GetReference([Group [, Element]]  
[, DESCRIPTION=string] [, VR=DICOM VR string])
```

Return Value

Returns an array referencing the matching elements in the object. References are opaque, meaning that they have no specific significance other than a correspondence to the element they refer to. If no arguments or keywords are specified, the returned array contains references to all elements in the object. The effect of multiple keywords and parameters is to AND their results. If no matching elements can be found, the function returns -1.

Arguments

Group

An optional argument representing the value for the DICOM group for which to search, such as '0018'x. If this argument is omitted, then all of the DICOM array elements are returned.

Element

An optional argument specified only if the *Group* argument has also been specified. Set this to the value for the DICOM element to search for, such as '0010'x. If this argument is omitted and the *Group* argument was specified, then all elements of the specified *Group* are returned.

Keywords

DESCRIPTION

Set this keyword to a string containing text to be searched for in each element's DICOM description. An element will be returned only if the text in this string can be found in the description. The text comparison is case-insensitive.

VR

Set this keyword to a DICOM VR string. An element will be returned only if its value representation matches this string.

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER = '*'))

; Get the reference to the patient name element:
ref = obj->GetReference('0010'x,'0010'x)
PRINT, ref

; Get references to all elements with "patient" in the description:
refs = obj->GetReference(DESCRIPTION = 'patient')
FOR i = 0, (N_ELEMENTS(refs) - 1) DO BEGIN
    PRINT, refs[i]
    PRINT, obj->GetDescription(REFERENCE = refs[i])
ENDFOR

; Get references to all elements with a VR of DA (date):
refs = obj->GetReference(vr = 'DA')
FOR i = 0, (N_ELEMENTS(refs) - 1) DO BEGIN
    PRINT, refs[i]
    PRINT, obj->GetDescription(REFERENCE = refs[i])
ENDFOR

OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::GetValue

This method takes optional DICOM group and/or element arguments and returns an array of POINTERS to the values of the elements matching those parameters.

Syntax

```
Result = Obj->[IDLffDICOM::]GetValue([Group [, Element]]  
    [, REFERENCE=list of element references] [, /NO_COPY])
```

Return Value

Returns an array of pointers to the values of the elements matching the *Group* and *Element* parameters. If no arguments or keywords are specified, the returned array contains pointers to all elements in the object. The effect of multiple keywords and parameters is to AND their results. If no matching elements can be found, the function returns -1.

Arguments

Group

Set this optional argument to the value for the DICOM group for which to search, such as '0018'x. If this argument is omitted, then all of the DICOM array elements are returned.

Element

This optional argument can be specified only if the Group argument has also been specified. Set this to the value for the DICOM element for which to search, such as '0010'x. If this argument is omitted and the Group argument was specified, then all elements of the specified Group are returned.

Keywords

REFERENCE

Set this keyword to a list of element references from which to return pointer values.

NO_COPY

If this keyword is set, the pointers returned point to the actual data in the object for the specified DICOM fields. If not set (the default), the pointers point to copies of the data instead, and need to be freed by using PTR_FREE.

Examples

Example 1

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get the image data
array = obj->GetValue('7fe0'x, '0010'x)
OBJ_DESTROY, obj

TVScl, *array[0]
PTR_FREE, array
```

Example 2

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get all of the image data element(s), 7fe0,0010, from the file:
array = obj->GetValue('7fe0'x,'0010'x,/NO_COPY)

; Get the row & column size of the image(s):
rows = obj->GetValue('0028'x,'0010'x,/NO_COPY)
cols = obj->GetValue('0028'x,'0011'x,/NO_COPY)

; If the image has a samples per pixel value greater than 1
; it is most likely a color image, get the samples per pixel:
isColor = 0
samples = obj->GetValue('0028'x,'0002'x,/NO_COPY)
IF (SIZE(samples,/N_DIMENSIONS) NE 0) THEN BEGIN
    IF (*samples[0] GT 1) THEN isColor = 1
ENDIF

; Next, we need to differentiate between files with color data
; that is either color-by-plane or color-by-pixel, get the planar
; configuration:
IF (isColor EQ 1) THEN BEGIN
    isPlanar = 0
    planar = obj->GetValue('0028'x,'0006'x, /NO_COPY)
    IF (SIZE(planar, /N_DIMENSIONS) NE 0) THEN BEGIN
```

```

        IF (*planar[0] EQ 1) THEN isPlanar = 1
    ENDIF
ENDIF

; Display the first NumWin images from the file:
IF N_ELEMENTS(array) GT 10 THEN NumWin = 10 $
ELSE NumWin = N_ELEMENTS(array)
offset = 0
FOR index = 0, NumWin-1 DO BEGIN
    ; Create window for each image that is the size of the image:
    WINDOW,index,XSize=*cols[0],YSize=*rows[0],XPos=offset,YPos=0
    WSET,index
    ; Display the image data
    IF (isColor EQ 1) THEN $
        IF (isPlanar EQ 1) THEN $
            ; color-by-plane
            TVScl,TRANPOSE(*array[index],[2,0,1]),/TRUE $
        ELSE $
            ; color-by-pixel
            TVScl,*array[index],/TRUE $
        ELSE $
            ; monochrome
            TVScl,*array[index]
            offset = offset+10
        ENDFOR

; Clean up
OBJ_DESTROY,obj

```

Version History

Introduced: 5.2

IDLffDICOM::GetVR

The IDLffDICOM::GetVR function method takes optional DICOM group and/or element arguments and returns an array of VR (Value Representation) STRINGS for those parameters.

Syntax

```
array = Obj->[IDLffDICOM::]GetVR([Group [, Element]]  
    [, REFERENCE=list of references])
```

Return Value

Returns an array of strings containing VRs (Value Representations) for the *Group* and *Element* parameters. A VR is a DICOM value representation as described in the DICOM specification PS 3.5. If no arguments or keywords are specified, the returned array contains VRs for all elements in the object. The effect of multiple keywords and parameters is to AND their results. If no matching elements can be found, the function returns -1.

Arguments

Group

An optional argument representing the value for the DICOM group for which to search, such as '0018'x. If this argument is omitted, then all of the DICOM array elements are returned.

Element

An optional argument specified only if the *Group* argument has also been specified. Set this to the value for the DICOM element for which to search, such as '0010'x. If this argument is omitted and the *Group* argument was specified, then all elements of the specified *Group* are returned.

Keywords

REFERENCE

Use the specified list of references from which to return VR STRING values.

Examples

```
obj = OBJ_NEW('IDLffDICOM')
read = obj->Read(DIALOG_PICKFILE(FILTER='*'))

; Get the VR of the patient name element:
arr = obj->GetVR('0010'x,'0010'x)
PRINT, arr

; Get an array of all of the VRs from the patient info group:
arr = obj->GetVR('0010'x)
PRINT, arr

OBJ_DESTROY,obj
```

Version History

Introduced: 5.2

IDLffDICOM::Init

The IDLffDICOM::Init function method creates a new IDLffDICOM object and optionally reads the specified file as defined in the IDLffDICOM::Read method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = OBJ_NEW( 'IDLffDICOM' [, Filename] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLffDICOM::]Init([Filename] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Filename

An optional scalar string argument containing the full path and filename of a DICOM v3.0 Part 10 file to open, read into memory, then close, when the object is created. It is the same as calling: result->Read(Filename).

Keywords

Any property listed under “[IDLffDICOM Properties](#)” on page 2734 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Examples

```
; Create a DICOM object:
obj = OBJ_NEW( 'IDLffDICOM' )

; Create a DICOM object and read in a DICOM file named ct_head.dcm
; under Microsoft Windows:
obj = OBJ_NEW( 'IDLffDICOM', $
    'c:\rsi\idl152\examples\data\mr_brain.dcm' )

; Create a DICOM object and allow the user to choose a DICOM file
; to be read:
obj = OBJ_NEW( 'IDLffDICOM', DIALOG_PICKFILE(FILTER='*'))
```

Version History

Introduced: 5.2

IDLffDICOM::Read

The IDLffDICOM::Read function method opens and reads from the specified disk file, places the information into the DICOM object, then closes the file.

Syntax

Result = *Obj*->[IDLffDICOM::]Read(*Filename* [, **ENDIAN**={1 | 2 | 3 | 4}])

Return Value

Return a 1 if successful and 0 otherwise.

Arguments

Filename

A scalar string argument containing the full path and filename of a DICOM Part 10 file to open and read into memory.

Keywords

ENDIAN

Set this keyword to configure the endian format when reading a DICOM file.

- 1 = Implicit VR Little Endian
- 2 = Explicit VR Little Endian
- 3 = Implicit VR Big Endian
- 4 = Explicit VR Big Endian

Examples

```
; Create a DICOM object and read a DICOM file:
obj = OBJ_NEW('IDLffDICOM')
var = obj->Read(DIALOG_PICKFILE(FILTER = '*'))
OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDICOM::Reset

The IDLffDICOM::Reset procedure method removes all of the elements from the IDLffDICOM object, leaving the object otherwise intact.

Syntax

Obj->[IDLffDICOM::]Reset

Arguments

None

Keywords

None

Examples

```
; Create a DICOM object, read a DICOM file and dump its contents:
obj = OBJ_NEW( 'IDLffDICOM' )
var  = obj->Read(DIALOG_PICKFILE(FILTER='*'))
obj->DumpElements
obj->Reset

; DumpElements should produce no output here:
obj->DumpElements
OBJ_DESTROY, obj
```

Version History

Introduced: 5.2

IDLffDXF

An IDLffDXF object contains geometry, connectivity and attributes for graphics primitives.

Note

IDL supports version 2.003 of the DXF Library.

This object treats a DXF file as a list of entities. Note, these are not directly mapped to DXF entity types, rather they are an abstraction of the DXF types. The Read method is used to read the contents of a DXF file into the current entity list. The user may then query this list using the GetContents method to determine the types and number of entities in the file. The user may retrieve arrays of entities from the list using the GetEntity method and add additional entities using the PutEntity method. Entities can also be removed from the list (RemoveEntity) or the entire list destroyed (Reset). The current list of entities can also be written to disk as a DXF file. Note, this object converts DXF entities to IDL entities and back. This conversion is not reversible; thus, if a DXF file is read and then written, the data in the file is not changed, but the internal DXF entity types may be changed by IDL. As an example, DXF face3d entities may be written as DXF polyline entities.

The object has one attribute which can be modified using the Get/SetPalette methods. This palette is used to convert color index values. The palette is not actually written to the DXF file. So, if the user wanted to specify entity colors from a 256 entry table, that table would be set using SetPalette, but the actual colors written to the file are the closest colors matched to the fixed AutoCAD color palette. There are two special color values: (0) = color by block color, (256) = color by layer color.

In this object, blocks and layers are treated as named entities with attributes, but are special in that all other entities have a block and layer entity reference in them. This allows the user to use these entity names as filters for many operations. There is a default block and a default layer. The default block has the name "" (the null string), and the default layer is '0'. The user may change the (non-name) attributes for these implicit blocks using PutEntity.

Superclasses

This class has no superclass.

Creation

See [“IDLffDXF::Init”](#) on page 2782

Properties

Objects of this class have no properties of their own.

Methods

This class has the following methods:

- [IDLffDXF::Cleanup](#)
- [IDLffDXF::GetContents](#)
- [IDLffDXF::GetEntity](#)
- [IDLffDXF::GetPalette](#)
- [IDLffDXF::Init](#)
- [IDLffDXF::PutEntity](#)
- [IDLffDXF::Read](#)
- [IDLffDXF::RemoveEntity](#)
- [IDLffDXF::Reset](#)
- [IDLffDXF::SetPalette](#)
- [IDLffDXF::Write](#)

Version History

Introduced: 5.2

IDLffDXF Properties

Objects of this class have no properties of their own.

IDLffDXF::Cleanup

The IDLffDXF::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffDXF::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.2

IDLffDXF::GetContents

The IDLffDXF::GetContents function method returns a description of the content of the DXF file.

See the [Examples](#) following the GetEntity method description for an illustration of the difference between the GetContents and the GetEntity methods.

Note

The Read or PutEntity methods must have been called previously for the results of this method to be valid.

Valid DXF ENTITY Types	DXF_TYPE (0=default)
ARC	1
CIRCLE	2
ELLIPSE	3
LINE	4
LINE3D	5
TRACE	6
POLYLINE	7
LWPOLYLINE	8
POLYGON	9
FACE3D	10
SOLID	11
RAY	12
XLINE	13
TEXT	14
MTEXT	15

Table 6-4: DXF Entity Types

Valid DXF ENTITY Types	DXF_TYPE (0=default)
POINT	16
SPLINE	17
BLOCK	18
INSERT	19
LAYER	20

Table 6-4: DXF Entity Types (Continued)

This object uses a small number of IDL named structures to return the data associated with each entity. This means that several of these DXF types are returned in the same structures, using different values of the DXF_TYPE field. The mapping of DXF entities to IDL named structures is as follows (each of these structures is documented in the GetEntity method):

IDL Structure	DXF Entity
IDL_DXF_ELLIPSE	arc, circle, ellipse
IDL_DXF_POLYLINE	line, line3d, trace, polyline, lwpolyline
IDL_DXF_POLYGON	face3d, solid, polyline (3-d mesh)
IDL_DXF_POINT	point
IDL_DXF_XLINE	ray, xline
IDL_DXF_SPLINE	spline
IDL_DXF_TEXT	text, multitext
IDL_DXF_BLOCK	block
IDL_DXF_INSERT	insert
IDL_DXF_LAYER	layer

Table 6-5: DXF mapping to IDL structures

Syntax

```
Result = Obj->[IDLffDXF::]GetContents( [Filter] [BLOCK=string]  
[, COUNT=variable] [LAYER=string] )
```

Return Value

Returns an integer array containing the DXF entity type codes and the number of occurrences of each entity type contained in the object

Arguments

Filter

An integer array of the DXF entity types to which the return types are restricted. If set, Result can contain only types given in this argument and count will also reflect that restriction.

Keywords

BLOCK

Set this keyword to a string value containing the block name to obtain the entities from. The default is all blocks.

COUNT

A long array containing the number of each entity type contained within the DXF object. If the Filter argument was provided, the numbers reflect the reduced set of entities caused by the Filter argument.

LAYER

Set this keyword to a string value containing the layer name to obtain the entities from. The default is all layers.

Version History

Introduced: 5.2

IDLffDXF::GetEntity

The IDLffDXF::GetEntity function method returns an array of data for the requested entity type.

Syntax

```
Result = Obj->[IDLffDXF::]GetEntity(Type [, BLOCK=string] [, INDEX=value]  
[, LAYER=string])
```

Return Value

Returns one of the named structure formats described in “[Structure Formats](#)” on page 2770.

Arguments

Type

The integer DXF entity type from which to obtain the geometry information.

Keywords

BLOCK

Set this keyword to a block name specifying the graphic block from which to obtain the entity geometry information. The default is all blocks. Setting this keyword to the null string "" will cause this method to only return entities from the default DXF entity block.

INDEX

Set this keyword to a scalar index or a long array of indices of entities of the given type to return. If not set, this method returns all entities for the given type.

LAYER

Set this keyword to a string value containing the layer name to obtain the entities from. The default is all layers.

Fields Common to All Structures

BLOCK

The name of the block this entity is in (these may be in the default block “”).

COLOR

A color index value into the current object palette with 0=use block color and 256=use layer color.

EXTRUSION

The DXF extrusion vector (if any).

LAYER

The name of the layer this entity is in (the default layer is '0').

LINETYPE

Defined the same as the user linestyle for IDLgrPolyline::Init.

Note

IDL will always return a solid line regardless of the linestyle in DXF

THICKNESS

In AutoCAD units.

DXF_TYPE

Set to one of the values listed in IDLffDXF::GetContents.

Note

It is the user's responsibility to free all the pointers returned in these structures when the entity is no longer needed.

Structure Formats

Structure IDL_DXF_ELLIPSE

Field	Data Type
PT0	Double [3]
PT1_OFFSET	Double [3]
MIN_TO_MAJ_RATIO	Double
START_ANGLE	Double
END_ANGLE	Double
EXTRUSION	Double [3]
LINESTYLE	Integer [2]
THICKNESS	Double
COLOR	Integer
DXF_TYPE	Integer
BLOCK	String
LAYER	String

Table 6-6: Fields of the IDL_DXF_ELLIPSE Structure

This object is centered at PT0 and has a radius defined by the vector PT1_OFFSET. This vector determines the length and orientation of the major axis of an ellipse as well.

The MIN_TO_MAJ_RATIO value specifies the length of the minor axis as a fraction of the major axis length. For a circle, this value is 1.0.

The START_ANGLE and END_ANGLE values select the portion of the curve to be drawn. If they are equal, the entire circle or ellipse is drawn.

Structure IDL_DXF_POLYGON

Field	Data Type
VERTICES	Pointer (to an array of 3-D points)
CONNECTIVITY	Pointer (to an array of integers)
VERTEX_COLORS	Pointer (to an array of integers)
MESH_DIMS	Integer [2]
CLOSED	Integer [2]
COLOR	Integer
EXTRUSION	Double [3]
FIT_TYPE	Integer
CURVE_FIT	Integer
SPLINE_FIT	Integer
DXF_TYPE	Integer
BLOCK	String
LAYER	String

Table 6-7: Fields of the IDL_DXF_POLYGON Structure

VERTICES is a pointer to an array of dimension [3, n] containing the points for this entity.

CONNECTIVITY is the array used to connect these points into polygons (see the POLYGONS keyword for IDLgrPolygon::Init). If this array is not present, the connectivity is implicit in (U, V) space defined by the values in MESH_DIMS; the vertices represent a quad mesh of dimensions (MESH_DIMS[0], MESH_DIMS[1]).

VERTEX_COLORS points to an array of color index values for each of the vertices. If a quad mesh is being returned, it can be closed in either dimension according to the CLOSED array.

FIT_TYPE, CURVE_FIT, and SPLINE_FIT return the type of curve fit (if any) this polygon assumes.

Structure IDL_DXF_POLYLINE

Field	Data Type
VERTICES	Pointer (to an array of 3-D points)
CONNECTIVITY	Pointer (to an array of integers)
VERTEX_COLORS	Pointer (to an array of integers)
COLOR	Integer
MESH_DIMS	Integer [2]
CLOSED	Integer [2]
THICKNESS	Double
LINESTYLE	Integer [2]
EXTRUSION	Double [3]
FIT_TYPE	String
CURVE_FIT	Integer
SPLINE_FIT	Integer
DXF_TYPE	Integer
BLOCK	String
LAYER	String

Table 6-8: Fields of the IDL_DXF_POLYLINE Structure

VERTICES is a pointer to an array of dimension [3, n] containing the points for this entity.

CONNECTIVITY is the array used to connect these points into polylines (see the POLYLINES keyword for IDLgrPolyline::Init). If this array is not present, the connectivity is implicit in (U, V) space defined by the values in MESH_DIMS; the vertices represent a quad mesh of dimensions (MESH_DIMS[0], MESH_DIMS[1]).

VERTEX_COLORS points to an array of color index values for each of the vertices. If a quad mesh is being returned, it can be closed in either dimension according to the CLOSED array.

FIT_TYPE, CURVE_FIT, and SPLINE_FIT return the type of curve fit (if any) this polyline assumes.

Structure IDL_DXF_POINT

Field	Data Type
PT0	Double [3]
UCSX_ANGLE	Double
THICKNESS	Double
COLOR	Integer
DXF_TYPE	Integer
BLOCK	String
LAYER	String

Table 6-9: Fields of the IDL_DXF_POINT Structure

PT0 is the location of the point in space.

UCSX_ANGLE is an internal DXF orientation parameter used for symbol plotting.

Structure IDL_DXF_SPLINE

Field	Data Type
CTR_PTS	Pointer
FIT_PTS	Pointer
KNOTS	Pointer
WEIGHTS	Pointer
COLOR	Integer
DEGREE	Integer
PERIODIC	Integer
RATIONAL	Integer
PLANAR	Integer
LINEAR	Integer
KNOT_TOLERANCE	Double
CTL_TOLERANCE	Double
FIT_TOLERANCE	Double
START_TANGENT	Double [3]
END_TANGENT	Double [3]
THICKNESS	Double
LINestyle	Integer [2]
EXTRUSION	Double [3]
DXF_TYPE	Integer
BLOCK	String
LAYER	String

Table 6-10: Fields of the IDL_DXF_SPLINE Structure

This structure is returned verbatim from the DXF spline structure without interpretation. It is up to the user to interpret these values.

Structure IDL_DXF_TXT

Field	Data Type
PT0	Double [3]
TEXT_STR	String
COLOR	Integer
HEIGHT	Double
WIDTH_FACTOR	Double
BOX_WIDTH	Double
DIRECTION	Double [3]
ROT_ANGLE	Double
JUSTIFICATION	Integer (0=left, 1=center, 2=right, 3=aligned, 4=middle, 5=fit)
VERTICAL_ALIGN	Integer (0=baseline, 1=bottom, 2=middle, 3=top)
SHAPE_FILE	String
THICKNESS	Double
EXTRUSION	Double [3]
DXF_TYPE	Integer
BLOCK	String
LAYER	String

Table 6-11: Fields of the IDL_DXF_TXT Structure

PT0 is the location of the text string.

TEXT_STR is the actual string.

HEIGHT specifies the overall scaling of the glyphs while WIDTH_FACTOR is a correction in the baseline direction (anisotropic scaling). For multi-line text, BOX_WIDTH determines where the line breaks should be placed (0.0 for single line text).

The text baseline is specified by `DIRECTION` and its rotation about the z -axis is specified by `ROT_ANGLE`. Justification is specified by `JUSTIFICATION` and `VERTICAL_ALIGN`. `SHAPE_FILE` is the name of the glyph file used to image this string. The shape file is NOT read by IDL.

Structure `IDL_DXF_XLINE`

Field	Data Type
<code>PT0</code>	Double [3]
<code>UNIT_VEC</code>	Double [3]
<code>COLOR</code>	Integer
<code>THICKNESS</code>	Double
<code>LINestyle</code>	Integer [2]
<code>EXTRUSION</code>	Double [3]
<code>DXF_TYPE</code>	Integer
<code>BLOCK</code>	String
<code>LAYER</code>	String

Table 6-12: Fields of the `IDL_DXF_XLINE` Structure

`PT0` is the start of a ray or a point on a infinite line in space in the case of an `XLINE` entity.

`UNIT_VEC` determines the direction of the line in space.

Structure IDL_DXF_INSERT

Field	Data Type
SCALE	Double [3]
PT0	Double [3]
ROTATION	Double
INSTANCE_BLOCK	String
NUM_ROW_COL	Integer [2]
DISTANCE_BETWEEN	Double [2]
DXF_TYPE	Integer
BLOCK	String
COLOR	Integer
LAYER	String

Table 6-13: Fields of the IDL_DXF_INSERT Structure

The insert entity allows for the “instancing” of a block in a grid fashion.

INSTANCE_BLOCK is the name of a block to repeat.

The block is scaled by SCALE and rotated about the Z axis by ROTATION. The grid begins at PT0 and contains the number of rows and columns specified by NUM_ROW_COL (Note: 0 rows or columns will always give a single instance of the block).

The spacing of the grid is specified by DISTANCE_BETWEEN.

Structure IDL_DXF_BLOCK

Field	Data Type
PT0	Double [3]
COLOR	Integer
NAME	String
DXF_TYPE	Integer

Table 6-14: Fields of the IDL_DXF_BLOCK Structure

This entity specifies a BLOCK. Blocks have a location in space (PT0) [objects in the block are interpreted relative to this point], a name, and a COLOR. They are not contained in layers or other blocks, so these fields are not present.

Structure IDL_DXF_LAYER

Field	Data Type
COLOR	Integer
NAME	String
DXF_TYPE	Integer

Table 6-15: Fields of the IDL_DXF_LAYER Structure

This entity specifies a LAYER. Layer is a NAME and a COLOR. They are not contained in layers or other blocks, so these fields are not present.

Examples

This example illustrates the difference between the `GetEntity` and `GetContents` methods within the `IDLffDXF` object method. The `GetContents` method gives a description of the content of the file read, listing the entity types and the number of occurrences. `GetEntity` accesses the values returned by `GetContents` and determines how the entities can be logically combined into common structures for drawing efficiency.

```

PRO view_heart
; Determine path to data file.
heartFile = FILEPATH('heart.dxf', $
    SUBDIRECTORY = ['examples', 'data'])
; Initialize DXF data access object.
oHeart = OBJ_NEW('IDLffDXF')
; Read data within DXF file into access object.
status = oHeart->Read(heartFile)
; Determine what type of entities (and how many of
; each entity) exist in the file.
heartTypes = oHeart->GetContents(COUNT = heartCounts)
PRINT, 'Entity Types: ', heartTypes
PRINT, 'Count of Types: ', heartCounts
; Initialize a model for displaying polygon and polyline
; objects.
oModel = OBJ_NEW('IDLgrModel')
; Obtain the tissue data. The tissue is accessed into
; IDL as a single polygon.
tissue = oHeart->GetEntity(heartTypes[1])
HELP, tissue
HELP, tissue, /STRUCTURE
; Initialize color parameter.
tissueColor = [255, 0, 0]
; Initialize polygon data.
vertices = tissue.vertices
connectivity = tissue.connectivity
; Initialize the polygon object.
oTissue = OBJ_NEW('IDLgrPolygon', $
    *vertices, POLYGONS = *connectivity, $
    COLOR = tissueColor)
; Add the polygon to the model.
oModel->Add, oTissue
; Clean-up all the related pointers.
PTR_FREE, tissue.vertices, tissue.connectivity, $
    tissue.vertex_colors
PTR_FREE, vertices, connectivity
; Display the polylines and the polygon in the XOBJVIEW
; utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 0.75
; Clean-up the object references.
OBJ_DESTROY, [oHeart, oModel]
END

```

An XOBJVIEW window with the mesh heart object appears when this program is compiled and run. The following lines are found in the IDLDE Output Log window.

```

IDL> view_heart
% Compiled module: VIEW_HEART.
% Compiled module: FILEPATH.

```

```

% Loaded DLM: DXF.
Entity Types:           7           10           18           20
Count of Types:        13          1624           1           2

```

The Entity Types (7, 10, 18, and 20) are the GetContents method return values corresponding to the occurrence of DXF ENTITY types found in the object: POLYLINES, FACE3D, BLOCK, and LAYER, respectively. Note there are a total of 1624 FACE3D entity types, but for efficiency and speed IDL logically combines all of the similar entities into a single structure that can be used in an IDLgrPolygon object. This is illustrated by the output resulting from the first Help command in the program as shown below.

```

TISSUE          STRUCT      ==>IDL_DXF_POLYGON Array[1]
** Structure IDL_DXF_POLYGON, 13 tags, length=72:
EXTRUSION       DOUBLE      Array[3]
VERTICES        POINTER     <PtrHeapVar3>
CONNECTIVITY    POINTER     <PtrHeapVar4>
VERTEX_COLORS   POINTER     <PtrHeapVar5>
MESH_DIMS       INT          Array[2]
CLOSED          INT          Array[2]
COLOR           INT          256
FIT_TYPE        INT          -1
CURVE_FIT       INT          0
SPLINE_FIT      INT          0
DXF_TYPE        INT          10
BLOCK           STRING      ''
LAYER           STRING      '0----'
% Compiled module: XOBJVIEW.
% Compiled module: UNIQ.
% Compiled module: IDENTITY.
% Compiled module: XMANAGER.

```

Note that the tissue is represented as an array of polygons having 1 element. This is evident by the mesh representation of the heart object showing the 1624 individual polygons as a single object.

Version History

Introduced: 5.2

IDLffDXF::GetPalette

The IDLffDXF::GetPalette procedure method returns the current color table in the object.

Syntax

Obj->[IDLffDXF::]GetPalette, *Red*, *Green*, *Blue*

Arguments

Red

Returns an array of the red components to the current color table.

Green

Returns an array of the green components to the current color table.

Blue

Returns an array of the blue components to the current color table.

Keywords

None

Version History

Introduced: 5.2

IDLffDXF::Init

The IDLffDXF::Init function method initializes the DXF object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = OBJ_NEW('IDLffDXF' [, Filename] )
```

or

```
Result = Obj->[IDLffDXF::]Init( [Filename] ) (Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Filename

A scalar string optional argument containing the full path and filename of a DXF file to be read as the object is created.

Keywords

None

Version History

Introduced: 5.2

IDLffDXF::PutEntity

The IDLffDXF::PutEntity procedure method inserts an entity into the DXF object. The type of the entity is determined from the DXF_TYPE field of the entity structure. If DXF_TYPE is set to 0, the type is implied by the entity structure.

Note

Line3D entity types will be written as Line entities due to the obsolete status of Line3D. Polyline entities will be automatically converted to Lightweight Polyline where applicable.

Syntax

Obj->[IDLffDXF:]PutEntity, *Data*

Arguments

Data

An array of Entity structures as defined by the GetEntity method.

Note

If the entity references a non-existent block or layer, one will automatically be created. Blocks and layers can also be created by passing IDL_DXF_BLOCK or IDL_DXF_LAYER structures to this routine.

Keywords

None

Version History

Introduced: 5.2

IDLffDXF::Read

The IDLffDXF::Read function method reads a file, parsing the DXF object information contained in the file, and inserts it into itself.

Syntax

Result = Obj->[IDLffDXF::]Read(Filename)

Return Value

Returns a 1 indicating success in reading the file, otherwise 0.

Arguments

Filename

A scalar string containing the full path and filename of the DXF file to be read.

Keywords

None

Examples

```
; Read all the lines from the electrical layer:
oDXF = OBJ_NEW('IDLffDXF')
IF (oDXF->Read('myDXF.dxf')) THEN BEGIN
    contents = oDXF->GetContents(4,COUNT=numLines, $
        LAYER='Electrical')
    IF (numLines ne 0) THEN BEGIN
        lines = oDXF->GetEntity(4,LAYER='Electrical')
    ENDIF
ENDIF
ENDIF
```

Version History

Introduced: 5.2

IDLffDXF::RemoveEntity

The IDLffDXF::RemoveEntity procedure method removes the specified entity or entities from the DXF object.

Syntax

Obj->[IDLffDXF::]RemoveEntity[, *Type*] [, INDEX=*value*]

Arguments

Type

An optional scalar string containing the DXF type to be removed from the DXF object.

Note

Specifying a block or layer entity will cause all the entities in that layer or block to be removed.

Keywords

INDEX

Set this keyword to a scalar long or a long array of indices to remove from the DXF object. If not set, or set negative, all entities of the given type are removed.

Version History

Introduced: 5.2

IDLffDXF::Reset

The IDLffDXF::Reset procedure method removes all the entities from the DXF object.

Syntax

Obj->[IDLffDXF::]Reset

Arguments

None

Keywords

None

Version History

Introduced: 5.2

IDLffDXF::SetPalette

The IDLffDXF::SetPalette procedure method sets the current color table in the object.

Syntax

Obj->[IDLffDXF::]SetPalette, *Red*, *Green*, *Blue*

Arguments

Red

Sets the red components of the current color table to this array.

Green

Sets the green components of the current color table to this array.

Blue

Sets the blue components of the current color table to this array.

Keywords

None

Version History

Introduced: 5.2

IDLffDXF::Write

The IDLffDXF::Write function method writes a file for the DXF entity information this object contains.

Syntax

Result = Obj->[IDLffDXF::]Write(Filename)

Return Value

Returns a 1 if successful in writing the file, 0 otherwise.

Arguments

Filename

A scalar string containing the full path and filename of the DXF file to be written.

Keywords

None

Examples

```

; Write a square to a new DXF file using lines:
oDXF = OBJ_NEW('IDLffDXF')
lines = {IDL_DXF_POLYLINE}
lines.dxf_type = 4
lines.layer='myLayer'
lines.thickness = 1.0

; Create clockwise square:
lines = REPLICATE(lines, 4)
lines[0].vertices = PTR_NEW([[0.0,0.0,0.0], $
    [0.0,1.0,0.0]])
lines[0].connectivity = PTR_NEW([0,1])
lines[1].vertices = PTR_NEW([[0.0,1.0,0.0], $
    [1.0,1.0,0.0]])
lines[1].connectivity = PTR_NEW([0,1])
lines[2].vertices = PTR_NEW([[1.0,1.0,0.0], $
    [1.0,0.0,0.0]])
lines[2].connectivity = PTR_NEW([0,1])
lines[3].vertices = PTR_NEW([[1.0,0.0,0.0], $

```

```
        [0.0,0.0,0.0]])
lines[3].connectivity = PTR_NEW([0,1])
oDXF->PutEntity, lines
IF (not oDXF->Write('mySquare.dxf')) THEN $
    PRINT, 'Write Failed.'
    ; Clean up the memory in the structs:
    OBJ_DESTROY, oDXF
    FOR i=0,3 DO BEGIN
        PTR_FREE, lines[i].vertices, lines[i].connectivity
    ENDFOR
```

Version History

Introduced: 5.2

IDLffJPEG2000

An IDLffJPEG2000 object contains the data for one or more images embedded in a JPEG2000 file as well as functionality for reading and writing JPEG2000 files. For additional information on the JPEG 2000 file format, see “[Overview of IDL and JPEG2000](#)” on page 2792.

Superclasses

This class has no superclasses.

Creation

See “[IDLffJPEG2000::Init](#)” on page 2814.

Properties

Objects of this class have the following properties. See “[IDLffJPEG2000 Properties](#)” on page 2795 for details on individual properties.

- [BIT_DEPTH](#)
- [COLOR_SPACE](#)
- [COMMENT](#)
- [DIMENSIONS](#)
- [DISPLAY_RESOLUTION](#)
- [FILENAME](#)
- [JP2](#)
- [N_LAYERS](#)
- [N_LEVELS](#)
- [N_COMPONENTS](#)
- [N_TILES](#)
- [OFFSET](#)
- [PALETTE](#)
- [PERSISTENT](#)
- [PROGRESSION](#)
- [QUIET](#)
- [READ](#)
- [REVERSIBLE](#)
- [SIGNED](#)
- [SUBSAMPLING](#)
- [TILE_DIMENSIONS](#)
- [TILE_OFFSET](#)
- [TILE_RANGE](#)
- [WRITE](#)
- [XML](#)
- [YCC](#)

Methods

This class has the following methods:

- [IDLffJPEG2000::Cleanup](#)
- [IDLffJPEG2000::GetData](#)

- [IDLffJPEG2000::GetProperty](#)
- [IDLffJPEG2000::GetTileProperty](#)
- [IDLffJPEG2000::Init](#)
- [IDLffJPEG2000::SetData](#)
- [IDLffJPEG2000::SetProperty](#)

Version History

Introduced: 6.1

Overview of IDL and JPEG2000

JPEG2000 is an open, international standard for image compression based on wavelet technology. It offers multiple advantages over other image file standards and compression methods. Among these advantages are:

- Low-bit rate compression performance equal to and often above that of other standards.
- Progressive transmission by quality, resolution, component, or region.
- Both lossless *and* lossy compression. (Lossless decompression is achieved naturally via progressions of increasing resolutions and/or quality.)
- Decompression of images which have user-selectable regions, resolutions, and quality.
- Random spatial access to the encoded stream of bits (code stream).
- Support for metadata, in the form of XML boxes.

A JPEG2000 encoded image and its most basic metadata comprise a code stream. This raw code stream may be placed in a file, usually given an extension of .j2k.

The JPEG2000 standard also allows for additional application-specific metadata to be added to a JPEG2000 file. A file that encapsulates the raw code stream and contains this additional metadata is referred to as a JP2 format file. IDL's support of the JP2 standard provides access to the following additional metadata types: color system specifications, palettes, XML containers, and display resolution information. These metadata are exposed through the [COLOR_SPACE](#), [PALETTE](#), [XML](#), and [DISPLAY_RESOLUTION](#) object properties. (See “[Properties](#)” on page 2790).

Note

JP2 format files must always have the file-type extension of .j2k or .jp2 (which are case insensitive).

The IDLffJPEG2000 object class in IDL provides access to the features of JPEG2000 and the JP2 file format for both input and output. This object class is detailed in the following section.

Note

Portions of IDL's JPEG2000 functionality were developed using Unisearch's Kakadu software, for which Kodak has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

The JPEG2000 Canvas Coordinate System

JPEG2000 uses the concept of the canvas to align multiple image components and tiles in a single coordinate system. This section explains the layout of image components and tiles within the canvas coordinate system.

Image Offsets and Image Subsampling

By default, images are placed on the canvas so that the image and canvas origins align. If an image offset is specified, the origin of all image components will be located at the specified *X* and *Y* offsets from the canvas origin.

Each image component can also be subsampled. The subsampling factors for a component indicate the scaling factor between the component dimensions and the image dimensions. If an image has dimensions X_I by Y_I , and a component has subsampling factors X_S by Y_S , the dimensions of the component will be the ceiling of X_I/X_S by the ceiling of Y_I/Y_S . For example, suppose an image has an overall size of 512 by 384 samples. An image component that has subsampling factors of 2 by 2 will have dimensions 256 by 192 samples. Calling the `GetData` method to retrieve just this component will return a 256 by 192 array.

In the canvas coordinate system, the subsampling factors have been applied to each component so that the dimensions of all components are always equal.

Note

When reading a region within the image with the `GetData` method, the values passed to the `REGION` keyword (`[StartX, StartY, Width, Height]`) are measured in the canvas coordinate system. The `StartX` and `StartY` values are measured from the canvas origin, and the `Width` and `Height` values are measured after any subsampling has been applied to the image components. This allows a single set of `Width` and `Height` values to be used for every component.

Note

If all components in an image have subsampling factors of 1 by 1, the dimensions of each component will be equal to the value contained in the `DIMENSIONS` property. To calculate the dimensions of individual components when some components have subsamplings other than 1 by 1, the following code can be used:

```
object->GetProperty, DIMENSIONS = dimensions, $
    SUBSAMPLING = subsampling, N_COMPONENTS = n_components
comp_sizes = ULONARR(2, n_components)
FOR i = 0, n_components - 1 do $
    comp_sizes[* ,i] = CEIL(FLOAT(dimensions) / subsampling[* ,i])
```

Tile Dimensions and Tile Offsets

By default, images will have one tile that has the same dimensions and offset on the canvas as the image. If the tile dimensions are smaller than the image dimensions and the tile offsets are different than the images offsets, some tiles may extend beyond the borders of the image.

The values returned via the `TILE_DIMENSIONS` and `TILE_OFFSET` keywords in the `GetTileProperty` method are the dimensions and location of the tile on the image itself, and not necessarily the dimensions and location on the canvas. To determine the overall dimensions of a tile, whether or not it falls outside the boundaries of the image, check the `TILE_DIMENSIONS` property with the `GetProperty` method.

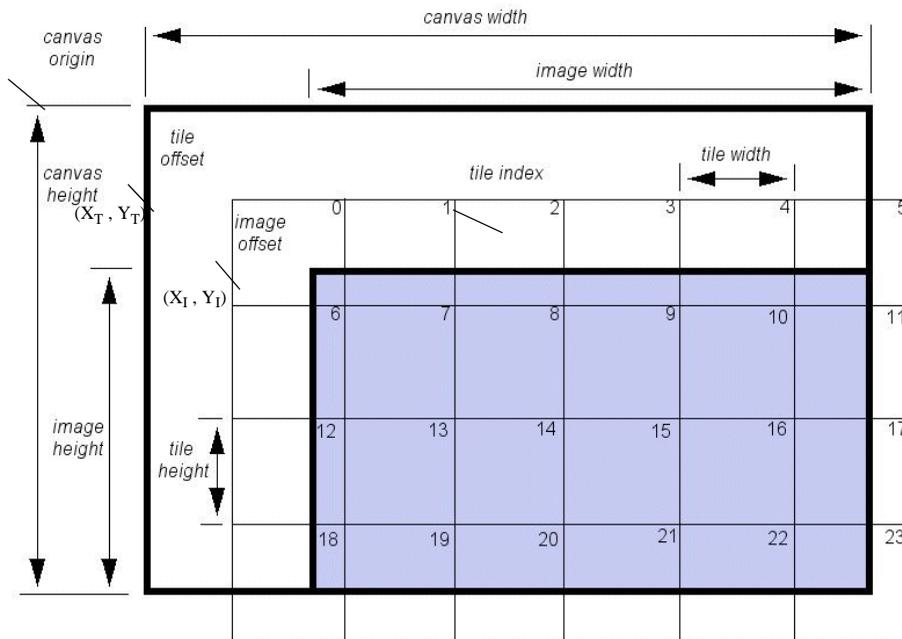


Figure 1: Image Tile Dimensions and Offsets

IDLffJPEG2000 Properties

IDLffJPEG2000 objects have the following properties. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLffJPEG2000::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLffJPEG2000::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLffJPEG2000::SetProperty](#).

Note

For more information about the “Property Type”, “Name String”, and “Registered” columns, see [“About Object Property Descriptions”](#) on page 2671.

BIT_DEPTH

An integer vector containing the bit depth for each image component. Values must be in the range 1 ... 24.

Note

When writing out JPEG2000 files, the default BIT_DEPTH is 8 for all components, regardless of the data type supplied to SetData. If you are writing out short or long integer data, you may wish to set BIT_DEPTH to a larger value for those components.

Property Type	Integer Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR_SPACE

A string containing one of the following values:

- sLUM - a luminance (monochrome) space.
- sRGB - the sRGB space is a reasonably good model for color imagery which has been corrected to look good on a monitor.
- sYCC - a three-component color space derived by applying the standard YCbCr color transform to sRGB data; the YCbCr transform is identical to

the irreversible color transform defined by JPEG2000 at the code stream level.

- iccLUM - a monochrome color space defined by an embedded ICC profile.
- iccRGB - an RGB color space defined by an embedded ICC profile.
- <Null string> - none of the above. A null string will result if the source is not a JP2 source.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COMMENT

A string vector containing an element for each comment associated with the object. If the object has no associated comments, the result will be a scalar null string.

Property Type	String Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DIMENSIONS

A two-element integer array giving the width and height of the image. Values for DIMENSIONS should be in the range $1 \dots 2^{32}-1$.

Property Type	Integer Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DISPLAY_RESOLUTION

A two-element floating-point vector giving the default horizontal and vertical display resolution (in samples per meter). This property will return a value of [0.0, 0.0] if the file has no display resolution specified. This keyword is only valid for JP2 files.

Note

The JPEG2000 code stream stores the display resolution values using only 24 bits of precision. When writing files, you may notice slight differences between your input values and the values returned when reading the file.

Property Type	Float Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FILENAME

A string containing the name of the file associated with the object

Note

FILENAME can only be set in Init as an argument.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes (as an argument)	Registered: No

JP2

A boolean returning 1 if the source or target is a JP2 file which supplies additional color interpretation and other functionality beyond the image data code stream, otherwise 0 is returned.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

N_COMPONENTS

An integer containing the number of image components (or channels) in the entire file or code stream. If N_COMPONENTS is not supplied in the first call to Init or

SetProperty, it will be the same as the number of components contained in the image parameters in the first call to SetData.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_LAYERS

A positive integer specifying the number of quality layers. A quality layer is one quality increment for the entire image at full resolution. Each layer contains the information required to represent the image at a higher quality, given the information from all the previous layers. All image data bits may be encoded in one layer, or may be broken up into a larger number of layers each containing a further quality refinement.

When writing, the default is one. If N_LAYERS is greater than one, the layers will be assigned roughly logarithmically-spaced bit rates, with the last layer containing all of the compressed bits.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_LEVELS

An integer specifying the number of wavelet decomposition levels, or stages, in the range of 0...15.

When writing files, a value of zero indicates that no wavelet transform will be used. If this keyword is omitted, then the default value of 5 is used. When writing JP2 files, if the PALETTE property is set then the default is N_LEVELS=0.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_TILES

An integer specifying the number of tiles within the file.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

OFFSET

A two-element integer array giving the X and Y offsets of all image components. Values for OFFSET should be in the range 0 ... $2^{32}-2$.

Property Type	Integer Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PALETTE

JP2 files may contain a “palette”, or color table, which efficiently stores images with only a few distinct colors. The palette associates a color index with each distinct color present in the image. Images stored in JP2 files with a palette consist of only one component which contains the color indices.

When writing a JP2 file, set this property to a 3-by- n or an n -by-3 array of byte or integer values, where n is the number of intensity values, for the three color channels of red, green, and blue. In this case n must be a power of 2 and ≤ 256 .

When reading JP2 files, this property will return an n -by- m array of byte or integer values, where n is the number of intensity values, and m is the number of lookup tables (m is usually 3, indicating red, green, and blue). If the associated object has no palette, this property will return a scalar zero.

Note

When writing JP2 files, if PALETTE is present, then the default values for other properties are N_LEVELS=0, REVERSIBLE=1, and YCC=0.

Property Type	Byte or Integer Array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PERSISTENT

A boolean value that indicates whether the image contained in the object may be accessed more than once. If set to zero, the image contained within the object may not be accessed more than once, and data are discarded whenever possible, minimizing memory usage. The default is PERSISTENT=1, which allows access to the image multiple times. This keyword is ignored if the file is opened for writing.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

PROGRESSION

A string defining the progression order of the output. Progression order may be one of the following: LRCP, RLCP, RPCL, PCRL, CPRL. The four character identifiers have the following interpretation: L = layer; R = resolution; C = component; P = position. Data is written to the output file iteratively over the four progression identifiers in this string, that is, the output is written in four nested iteration levels. The first character in the identifier refers to the outermost level of the iteration, while the last refers to the innermost level. The default is LRCP. This property is ignored when used with files opened for reading.

For example, “LRCP” means that an image is encoded in the following order, with quality being the outermost loop:

- L - a series of one or more quality layers, each contributing a quality increment for the entire image at full resolution. The quality layers may be interpreted as sub-bitplanes containing the image data at a specified, possibly fractional, bit-rate.
- R - each resolution level of the image, which corresponds to the N_LEVELS keyword used in encoding, and can be considered to contain the image data at a given size or resolution. Each level adds another level of detail to the previous levels.

- C - each image component of the above.
- P - each position (tile) within the image component.

In this example, if an image consists of 2 quality layers, 5 resolution levels, 3 components, and 16 positions or tiles, the image would be encoded in a series of packets as follows:

```
P0C0R0L0, P1C0R0L0, P2C0R0L0, ..., P15C0R0L0
P0C1R0L0, P1C1R0L0, P2C1R0L0, ..., P15C1R0L0
P0C2R0L0, P1C2R0L0, P2C2R0L0, ..., P15C2R0L0
P0C0R1L0, P1C0R1L0, P2C0R1L0, ..., P15C0R1L0
P0C1R1L0, P1C1R1L0, P2C1R1L0, ..., P15C1R1L0
P0C2R1L0, P1C2R1L0, P2C2R1L0, ..., P15C2R1L0
... ..
... ..
... ..
P0C2R4L0, P1C2R4L0, P2C2R4L0, ..., P15C2R4L0

P0C0R0L1, P1C0R0L1, P2C0R0L1, ..., P15C0R0L1
P0C1R0L1, P1C1R0L1, P2C1R0L1, ..., P15C1R0L1
P0C2R0L1, P1C2R0L1, P2C2R0L1, ..., P15C2R0L1
```

This encodes all components over the entire image at a given quality level, before sending the next quality refinement.

When encoding a large image with a large number of components, memory requirements will be minimized by sending each component by means of a separate call to `SetData`, and by specifying a progression order of “CPRL”.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

QUIET

A boolean that when set to a non-zero value suppresses the printing of warnings for this object. The default is not to suppress warnings.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

READ

A boolean set to 1 if the file is to be accessed for reading. The default is to open the file for reading.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

REVERSIBLE

A boolean that specifies the type of compression that should be used. A non-zero value indicates that reversible compression should be used. The default is to use irreversible compression. When writing JP2 files, if the PALETTE property is set then the default is REVERSIBLE=1.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SIGNED

An n -element boolean vector where n is the number of components. A value of one indicates pixels in that component are interpreted as signed numbers, otherwise the pixels are interpreted as unsigned numbers. The default is to interpret the numbers as unsigned.

Note

When writing out JPEG2000 files, the default is SIGNED=0 for all components, regardless of the data type supplied to SetData. If you are writing out signed integer data, you should set SIGNED to 1 for those components.

Property Type	Boolean Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SUBSAMPLING

A 2 by N_COMPONENTS integer array containing the subsampling factors in the horizontal and vertical dimensions for each image component. The dimensions of each component will be the ceiling of the image dimensions divided by the respective subsampling factor.

Property Type	Integer Array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

TILE_DIMENSIONS

A 2-element integer vector defining the width and height of each tile. When writing files, the default is to write one tile with the same dimensions as the image parameter. Values for TILE_DIMENSIONS should be in the range $1 \dots 2^{32}-1$.

Property Type	Integer Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TILE_OFFSET

A 2-element integer vector containing the X and Y offset of the first tile on the canvas.

When reading, the TILE_OFFSET will return the X and Y offset of the first tile. The GetTileProperty method may be used to retrieve the TILE_OFFSET for an individual tile.

When writing, the default is TILE_OFFSET=[0,0]. The values of TILE_OFFSET must be less than or equal to the corresponding values of the OFFSET property. Values for TILE_OFFSET should be in the range $0 \dots 2^{32}-2$.

Property Type	Integer Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TILE_RANGE

A two-element integer vector giving the number of tiles in the X and Y directions.

Property Type	Integer Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

WRITE

A boolean that is 1 if the object is open for writing, 0 if not.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

XML

A string vector, each element of which contains a corresponding XML box of the object. If the object has no XML boxes, the result will be a scalar null string. This keyword is only valid for JP2 files.

Property Type	String Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YCC

When reading files, this property indicates whether the data was originally converted from the RGB color space to the YCC color space. If YCC is true, then when calling GetData, you may wish to return all three image components so that the YCC data can be correctly transformed back to RGB.

When writing files, set this property to a boolean which specifies whether RGB data should be converted to YCrCb before being written. If the image being written has three components, the default is to assume the image contains RGB components, which are automatically converted to YCrCb, and then written. The default action to convert RGB to YCC may be overridden by explicitly setting the YCC keyword to zero. When writing JP2 files, if the PALETTE property is set then the default is

YCC=0. If the image being written does not have three components, this property is ignored.

Transforming the color coordinates from RGB to YCC decorrelates the three color components and improves compression performance for RGB images. JPEG2000 readers will then convert back to RGB.

Note

For the RGB-to-YCC conversion to occur, all three components must be provided at the same time (either the entire image or as separate tiles). If the components are provided in separate calls to `SetData`, the YCC conversion is not performed.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLffJPEG2000::Cleanup

The IDLffJPEG2000::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Note

Calling OBJ_DESTROY will also close any files that have been opened for reading or writing.

Syntax

OBJ_DESTROY, *Obj*

or

Obj - > [IDLffJPEG2000:]Cleanup(*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

IDLffJPEG2000::GetData

The IDLffJPEG2000::GetData function method returns image data from the IDLffJPEG2000 object.

Syntax

```
Result = Obj->[IDLffJPEG2000::]GetData( [COMPONENT=value]  
    [, DISCARD_LEVELS=value] [, MAX_LAYERS=value]  
    [, N_COMPONENTS=value] [, ORDER=value] [, REGION=value] [, /RGB]  
    [, TILE_INDEX=value])
```

Return Value

An array with two or three dimensions containing the image data from the object. Dimensions of the result are: [3, *Width*, *Height*] if the RGB keyword is set; or [*nComponents*, *Width*, *Height*] if the RGB keyword is not set. *nComponents* is the number of components present in the file. Images with only one component (which are read with RGB not set) return a two dimensional array [*Width*, *Height*].

The data type of the result is automatically determined using the bit-depth and signed properties of the returned components. For bit-depths ≤ 8 (or if RGB is set) the result will be of type byte. For bit-depths ≥ 9 and ≤ 16 the result will be of type integer or uint. For bit-depths > 16 the result will be long or ulong. Components with differing bit-depths will be returned using the largest bit-depth. Components which are a mix of signed and unsigned will be returned as signed.

Note

If the PERSISTENT keyword was set to zero when the object was created, the object may not be accessed after GetData is called, with the exception of the GetProperty and Cleanup methods.

Note

If the data in the file was transformed from RGB to YCC before saving, the library will only transform back from YCC to RGB if all three components are read in simultaneously. Reading in each component separately will return the data in the YCC space. The YCC property may be used to determine if the RGB-to-YCC conversion took place. If YCC is true, you may wish to avoid using the COMPONENT or N_COMPONENTS keywords.

Keywords

COMPONENT

Set this keyword to an integer giving the index of the component at which to start reading. The number of components to read may be specified using the `N_COMPONENTS` keyword. The default is `COMPONENT=0`.

DISCARD_LEVELS

Set this keyword to indicate the number of highest resolution levels which will not appear in the result.

The resulting image dimensions are given by:

$$\text{CEIL}((\text{Dimensions} + \text{Offset}) / (2^{\text{DL}})) - \text{CEIL}(\text{Offset} / (2^{\text{DL}}))$$

where *Dimensions* and *Offset* are the image dimensions and offset, and *DL* is the value of `DISCARD_LEVELS`.

For example, if the image in the object has dimensions of 1024 x 1026, and `DISCARD_LEVELS` is set to 3, the resulting image will have dimensions of 128 x 129, which is the ceiling of [1024, 1026] divided by 2^3 . The number of resolution levels in a file may be determined by checking the value of the `N_LEVELS` property.

MAX_LAYERS

Set this keyword to the maximum number of quality layers which will appear to be present. A quality layer is one quality increment for the entire image at full resolution. Each layer contains the information required to represent the image at a higher quality, given the information from all the previous layers. All image data bits may be encoded in one layer, or may be broken up into a larger number of layers each containing a further quality refinement. A value of 0, the default, implies that all layers should be retained. For example, specifying `MAX_LAYERS = 4` processes only the first four quality layers. The number of quality layers in a file may be determined by checking the value of the `N_LAYERS` property.

N_COMPONENTS

Set this keyword to an integer giving the number of components to read, starting at the component index given by the `COMPONENT` keyword. If neither `COMPONENT` nor `N_COMPONENTS` is specified then all components will be read. If `COMPONENT` is specified then the default is `N_COMPONENTS=1`.

ORDER

JPEG2000 images are assumed to be stored in top-to-bottom order, while IDL usually assumes images are in bottom-to-top order. By default (`ORDER = 0`), `GetData` will automatically flip the result to match IDL's bottom-to-top order. Setting `ORDER` to a nonzero value will return the JPEG2000 image without doing the vertical flip.

REGION

Set this keyword to a four-element vector containing the rectangular region of the image to read in the canvas coordinate system. The region is specified as [*StartX*, *StartY*, *Width*, *Height*], where *StartX* and *StartY* must be in the range $0 \dots 2^{32}-2$, and *Width* and *Height* must be in the range $1 \dots 2^{32}-1$. The region must contain at least some portion of the image, or an error will occur. If the region is larger than the image, or extends outside of the image, then only that portion that lies within the image will be returned.

Note

The region parameters should be specified in the canvas coordinate system, taking into account the canvas offset and canvas dimensions.

RGB

If this keyword is set, the data stream will be interpreted as RGB imagery and all details concerning color transformations (for example: `sRGB`, `sLUM` and `sYCC`), together with restricted ICC-profiles for specifying more general 3-color “RGB-like” spaces, plus indexed color with palettes, are handled automatically. If this keyword is set, the result of this function will always have three components, red, green, and blue; and the dimensions of the result will be [`3`, *Width*, *Height*].

The JPEG2000 standard allows a decorrelating color transformation to be applied to the first three image components. This transformation is identical or similar to the conventional YCbCr color transformation. To determine if this transform has been applied, examine the `YCC` keyword returned by the `::GetFileProperty` method.

If `RGB` is set and the file contains more than three components, only the first three components are returned. If `RGB` is set and the file contains less than three components, data from the first component is duplicated in the other components.

If `RGB` is not set, and three components are transferred, and a color transformation has been applied to the input stream, the three components which are encoded in a form of YCC will automatically be converted to RGB. If it is desired to avoid this conversion, then the user should transfer more or less than three components in the call or calls to `GetData`.

If RGB is not set and the number of components is not three, then each component in the data stream will be placed in the result without interpretation. Result will be: [N_COMPONENTS, *Width*, *Height*].

The RGB keyword may not be specified if the COMPONENT or N_COMPONENTS keywords are present.

TILE_INDEX

Set this keyword to an integer giving the tile index to read. If this keyword is specified then the REGION keyword is ignored.

Examples

This example illustrates the use of GetData to read an image a number of different ways:

```

;Create object, open file.
a = OBJ_NEW('IDLffJPEG2000', 'boat2.jp2')

;Read the entire image.
b = a->GetData()

;Read a 512x512 portion of the image, starting
;at column 100, row 200.
b = a->GetData(REGION = [100, 200, 512, 512])

;Read the portion of the above, but sub-sample it down by a
;factor of 2, returning an nComponents x 256 x 256 image.
b = a->GetData(REGION = [100, 200, 512, 512], DISCARD_LEVELS=1)

;Read only the 2nd component.
;The result will have two dimensions.
b = a->GetData(COMPONENT = 1)

```

Version History

Introduced: 6.1

IDLffJPEG2000::GetProperty

The IDLffJPEG2000::GetProperty procedure method returns information describing the properties of an IDLffJPEG2000 object in the supplied variable - keyword pairs.

Syntax

Obj->[IDLffJPEG2000:]GetProperty [, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under [“IDLffJPEG2000 Properties”](#) on page 2795 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.1

IDLffJPEG2000::GetTileProperty

The IDLffJPEG2000::GetTileProperty procedure method returns information describing the properties of a tile in an IDLffJPEG2000 object in the supplied variable-keyword pairs. The object must be open for reading. An error is produced if the object is open for writing.

Syntax

```
Obj->[IDLffJPEG2000:]GetTileProperty [, TileIndex [, TileComponent]],
    [, N_LAYERS=variable] [, N_LEVELS=variable] [, PROGRESSION=variable]
    [, REVERSIBLE=variable] [, TILE_DIMENSIONS=variable]
    [, TILE_OFFSET=variable] [, YCC=variable]
```

Arguments

TileIndex

This argument specifies the index of the tile whose attributes are to be returned. JPEG2000 tiles are numbered in raster scan fashion, starting from 0 and going up to a maximum of 65535. The default value for this argument is 0.

TileComponent

This argument specifies the component index of the tile whose attributes are to be queried. Indices range from 0 to N_COMPONENTS-1. The default value for this argument is 0.

Keywords

N_LAYERS

Set this keyword to a named variable in which to return the number of quality layers available for the tile. If this keyword is set, the TileComponent argument is ignored.

N_LEVELS

Set this keyword to a named variable in which to return the total number of available resolution levels for this tile component.

PROGRESSION

Set this keyword to a named variable in which to return a string containing the progression order for that tile. The returned value will be one of: LRCP, RLCP, RPCL, PCRL, CPRL.

See “[PROGRESSION](#)” on page 2800 under Properties for the definition of these values.

REVERSIBLE

Set this keyword to a named variable in which to return 1 if the tile-component is compressed reversibly, 0 if irreversible.

TILE_DIMENSIONS

Set this keyword to a named variable in which to return a two-element vector containing the width and height of the tile, taking into account any clipping of the tile to the image.

TILE_OFFSET

Set this keyword to a named variable in which to return a two-element vector containing the horizontal and vertical offset of the tile, taking into account any clipping of the tile to the image.

YCC

Set this keyword to a named variable in which to return 1 if a component color transform was used for the first 3 components of the tile specified by the TileIndex argument, 0 otherwise.

Version History

Introduced: 6.1

IDLffJPEG2000::Init

The IDLffJPEG2000::Init() function method initializes an IDLffJPEG2000 object that can be used to read (decode) image data from a JPEG2000 image file, or write (encode) image data to a JPEG2000 image file.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLffJPEG2000' , *Filename*[, *PROPERTY* = *value*])

or

Result = *Obj*->[IDLffJPEG2000::] Init(*Filename* [, *PROPERTY* = *value*] (*Only in subclass' Init method*))

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Filename

Set this argument to a scalar string containing the path and filename of a file containing JPEG 2000 compressed image data to be accessed through this object. If the file type suffix is .jp2 or .jpx, case insensitive, the file is a JP2 file. Otherwise the file contains a raw JPEG2000 code stream.

Keywords

Any property listed under [“IDLffJPEG2000 Properties”](#) on page 2795 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.1

IDLffJPEG2000::SetData

The IDLffJPEG2000::SetData procedure method writes data to the IDLffJPEG2000 object.

Syntax

```
Obj->[IDLffJPEG2000::]SetData ([P1, ..., Pn] [, COMPONENT=value]
    [, /ORDER] [, TILE_INDEX=value])
```

Arguments

Arguments may be one or more arrays containing the image components or tile components to be written to the JPEG2000 object. The arrays may have either two or three dimensions. Two-dimensional arrays contain one image or tile component with the dimensions of width and height. Three-dimensional arrays are dimensioned [*nComponents, width, height*]. All parameters must have the same width and height.

The array parameters are converted to byte data type if the bit-depth for the corresponding component is less than or equal to 8; to signed or unsigned integer, depending on the signed setting, for bit-depths from 9 - 16; and to signed or unsigned long integers, depending on the signed setting for bit-depths over 16.

Note

The bit depths used for output are determined from the BIT_DEPTH property, and not from the data type of the input arguments. If you are writing out short or long integer data, you should set the BIT_DEPTH property before calling SetData.

Note

The SIGNED property determines whether the data are written out using signed or unsigned integers, and is not affected by the data type of the input arguments. If you are writing out signed integer data, you may wish to set the SIGNED property before calling SetData.

Image Dimensions, Tile Dimensions, and Component Number

Images may be written using multiple calls to SetData with each call transferring one or more tile components. A tile component contains one image component for a given tile and is the smallest unit of data that can be supplied to SetData. The dimensions of the entire image and tiles, plus the number of components are determined the first time they are specified to Init or SetProperty, and may not be changed once

established. If the dimensions or number of components are not specified explicitly they are inferred from the image parameters supplied in the first call to `SetData`.

If transferring an image with multiple calls to `SetData`, try to supply the data to `SetData` in a manner consistent with the progression setting of `PROGRESSION` to ensure optimum performance and minimize memory requirements.

Note

If you do not complete writing all of the tiles or components, or your array dimensions are not the same as the overall image dimensions, then the library will be unable to complete the code stream. In this case, reading the data back in may result in unexpected or incorrect values.

Note

If the `YCC` property is set (indicating that you want the library to convert from RGB to YCC before compressing the data), then you must supply all three image components in the same call to `SetData` (either the entire image or one tile at a time). If the components are provided in separate calls to `SetData`, the YCC conversion is not performed.

Keywords

COMPONENT

Set this keyword to be the index of the first component to be written by this call to `SetData`. Default is 0.

ORDER

JPEG2000 images are assumed to be stored in top-to-bottom order, while IDL usually assumes images are in bottom-to-top order. By default (`ORDER = 0`), `SetData` will automatically flip the input arrays to match JPEG2000's top-to-bottom order. Setting `ORDER` to a nonzero value will save the image directly to the JPEG2000 file without doing the vertical flip.

TILE_INDEX

Set this keyword to specify the index of the first tile to be written. JPEG2000 tiles are numbered in raster scan fashion, starting from 0. The default value for this keyword is 0. The tile indices must range from 0 to one less than the number of tiles in the image file, which is determined by the tile dimensions and image dimensions.

Examples

Write a one-component image:

```
;Open file to write
jp2 = OBJ_NEW('IDLffJPEG2000', 'test.jp2', /WRITE)

;Write the image
jp2->SetData, BYTSC(L(DIST(400,300))

OBJ_DESTROY, jp2
```

Read a one-component image:

```
;Open file to read
jp2_in = OBJ_NEW('IDLffJPEG2000', 'test.jp2')

;Read it.
img = jp2_in->GetData()

OBJ_DESTROY, jp2_in
```

Write an indexed-color image with a palette:

```
;Load IDL color table #5, this example assumes we have 256 colors.
LOADCT, 5

;Read color tables
TVLCT, /GET, Red, Green, Blue

;Open file to write
jp2 = OBJ_NEW('IDLffJPEG2000', 'test.jp2', /WRITE, $
  PALETTE=[[Red], [Green], [Blue]])

;Write the image with the palette
jp2->SetData, BYTSC(L(DIST(400,300))
OBJ_DESTROY, jp2
```

Read the above image and palette:

```
;Open file to read
jp2_in = OBJ_NEW('IDLffJPEG2000', 'test.jp2')

jp2_in->GetProperty, PALETTE=palette

;Read without RGB keyword. img is a 400 by 300 byte image.
img = jp2_in->GetData()

;With the RGB keyword, the image is now [3, 400, 300]
img3d = jp2_in->GetData(/RGB)
```

```
OBJ_DESTROY, jp2_in
```

Show the images:

```
TVLCT, palette[*], palette [*, 1], palette[*], 2]
DEVICE, DECOMPOSED = 0
TV, img
DEVICE, DECOMPOSED = 1
TV, img3d, TRUE=1
```

Write a 100 component file, one component at a time:

```
j2k = OBJ_NEW('IDLffJPEG2000', 'test.j2k', /WRITE,
             N_COMPONENTS=100)
FOR i = 0, 99 DO j2k_in->SetData, $
    replicate(byte(i), 200, 300), COMPONENT = i
OBJ_DESTROY, j2k_in
```

Read the above file:

```
j2k_in = obj_new('IDLffJPEG2000', 'test.j2k')

;b is a [200, 300] array
b = j2k_in->GetData(COMPONENT=30)

;b is a [5, 200, 300] array
b = j2k_in->GetData(COMPONENT=50, N_COMPONENT = 5)

OBJ_DESTROY, j2k_in
```

Version History

Introduced: 6.1

IDLffJPEG2000:: SetProperty

This procedure sets the properties of an object that is open for writing. This method may not be called on files opened for reading.

Syntax

Obj->[IDLffJPEG2000:]SetProperty [, *PROPERTY=value*]

Arguments

None.

Keywords

Any property listed under [“IDLffJPEG2000 Properties”](#) on page 2795 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.1

IDLffLangCat

The IDLffLangCat class is used to find and load an XML language catalog. A language catalog is composed of key names and key values. The key value of the specified key name can be retrieved to support internationalization. For example:

```
<KEY NAME='Menu:File:Open'>Open...</KEY>
```

in an English language catalog would appear as the following in a French language catalog:

```
<KEY NAME='Menu:File:Open'>Ouvrir...</KEY>
```

By handling files that contain keys and values like these, the IDLffLangCat class supports internationalization of user interfaces and other applications that need to display the same strings in more than one language.

The language catalogs included with IDL are in the `<IDL_DIR>/resource/langcat` directory and end in a `.cat` extension. You can use other catalogs, but all catalogs must end with the `.cat` extension.

Note

The IDLffLangCat class supports catalog files written in 8-bit Latin alphabet languages. Also, you must have the corresponding fonts installed on your machine before you can use a particular language.

Note

For more information on this class, including examples of how to use it, see [“Using Language Catalogs”](#) in the *Building IDL Applications* manual.

Superclasses

None

Creation

See [IDLffLangCat::Init](#).

Properties

Objects of this class have the following properties. See [“IDLffLangCat Properties”](#) on page 2823 for individual properties.

- [APP_NAME](#)

- APP_PATH
- AVAILABLE_LANGUAGES
- DEFAULT_KEYS
- DEFAULT_LANGUAGE
- DEFAULT_N_KEYS
- FILENAME
- KEYS
- LANGUAGE
- N_KEYS
- VERBOSE

Methods

This class has the following methods:

- IDLffLangCat::AppendCatalog
- IDLffLangCat::Cleanup
- IDLffLangCat::GetProperty
- IDLffLangCat::Init
- IDLffLangCat::Query
- IDLffLangCat::SetProperty

Version History

Introduced: 6.1

IDLffLangCat Properties

IDLffLangCat objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLffLangCat::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLffLangCat::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLffLangCat::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

APP_NAME

The name of one or more current applications, returned in alphabetical order.

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

APP_PATH

The directory or directories searched for each given application. Each directory path must be fully qualified and unambiguous (e.g., C:\RSI\IDL61\lib\itools).

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

AVAILABLE_LANGUAGES

A list of all languages that exist in the input file or files. When APP_NAME is specified, this property returns all languages where the language-catalog files have the same internal application name. The languages return in alphabetical order.

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DEFAULT_KEYS

A list of all the currently loaded keys in the default language.

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DEFAULT_LANGUAGE

The current default language.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

DEFAULT_N_KEYS

The number of currently loaded keys in the default language.

Property Type	Long		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

FILENAME

The name of each currently loaded language catalog (.cat) file. In setting this property, each name is either a fully qualified path (e.g., C:\RSI\IDL61\resource\langcat) to a location containing a file, or the name of a file in the current directory. In getting this property, it always returns the full path.

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

KEYS

A list of all the currently loaded keys, returned in upper-case.

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

LANGUAGE

The current language.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_KEYS

The number of currently loaded keys.

Property Type	Long		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

VERBOSE

The current setting of the verbose flag. If set, all catalog-loading messages will be sent to the output window. This list contains details resulting from an IDLffLangCat object initialization or modification, including the names and numbers of keys loaded, keys overwritten, and so on.

Property Type	Byte		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLffLangCat::AppendCatalog

The IDLffLangCat::AppendCatalog function method adds keys from a file or files to those used to build the language catalog. All valid keys present in the current language will be inserted into the current key/value list. If any new languages are present in the new files, they will be added to the AVAILABLE_LANGUAGES property. If a duplicate key name is found, its value will replace the existing key name's value.

Note

Whenever the object encounters a key (or language) that already exists, the key (or language) is overwritten with the new value.

Syntax

```
Result = Obj->[IDLffLangCat:]AppendCatalog( [, APP_NAME=applicationName]  
      [, FILENAME=filename] [, APP_PATH=applicationPath] )
```

Return Value

Returns 1 if valid keys were found and loaded, or 0 otherwise.

Arguments

None

Keywords

APP_NAME

Set this keyword to a string or string array containing the name of each desired application for which to add keys. If APP_NAME is specified, it overrides any input to FILENAME. You can use the '*' wild card (e.g., 'MyApp*' instead of ['MyAppMenu', 'MyAppErrors', ...]). You must provide either APP_NAME or FILENAME.

APP_PATH

Set this keyword to a string or string array containing the name of each directory in which to search for the given application's keys. Each path must be fully qualified

(e.g., `C:\RSI\IDL61\lib\itools`). This keyword is ignored if `APP_NAME` is not specified.

FILENAME

Set this keyword to a string or string array containing the name of each language-catalog file to load. If `APP_NAME` is specified, it overrides any input to `FILENAME`. Each string is either a fully qualified path (e.g., `C:\RSI\IDL61\resource\langcat`) to a location containing a file, or the name of a file in the current directory. You must provide either `APP_NAME` or `FILENAME`.

Version History

Introduced: 6.1

IDLffLangCat::Cleanup

The IDLffLangCat::Cleanup procedure method performs all cleanup operations on the language catalog object, and should be called by the Cleanup method of a subclass.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffLangCat::]Cleanup (*Only in subclass's Cleanup method*)

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffLangCat::GetProperty

The IDLffLangCat::GetProperty procedure method retrieves a property or group of properties for an IDLffLangCat object.

Comparisons such as those done with [IDLffLangCat::Query](#) are case insensitive, but the values returned by GetProperty are exactly as the last encountered value. For example, if File 1 has LANGUAGE= 'English' and File 2 has LANGUAGE= 'english', then 'english' will be returned, although only one ENGLISH language exists in the current catalog.

Syntax

Obj->[IDLffLangCat:]GetProperty [, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under [“IDLffLangCat Properties”](#) on page 2823 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.1

IDLffLangCat::Init

The IDLffLangCat::Init function method initializes an IDLffLangCat object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Any invalid input will result in a failure to create a valid object. Invalid input includes the following cases:

- The requested language is not found in any of the input files
- Any application in APP_NAME is not found
- Any file in FILENAME is not found
- Any directory in APP_PATH does not exist
- The requested default language is not found

You can set the VERBOSE keyword to view the output log for a detailed description of the creation (or failure) of the object. You can also set [CONTINUE_ON_ERROR](#) to override the failure behavior if you have the language specified, regardless of the language's existence.

Syntax

```
Obj = OBJ_NEW( 'IDLffLangCat', Language [, PROPERTY=value]
              [, /CONTINUE_ON_ERROR]
```

or

```
Result = Obj->[IDLffLangCat::]Init( Language [, PROPERTY=value]
                                   [, /CONTINUE_ON_ERROR] )
```

(Only in a subclass's Init method)

Return Value

Returns the new object if successful or a null object otherwise if initialized with OBJ_NEW, or returns 1 if successful or 0 otherwise if called from a subclass's Init method.

Arguments

Language

A string containing the name of the language to load from the catalog.

Keywords

Any property listed under “[IDLffLangCat Properties](#)” on page 2823 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

You must specify either APP_NAME or FILENAME for this method. If APP_NAME is specified, it overrides any input to FILENAME.

CONTINUE_ON_ERROR

If set, the object will be created as long as the language is specified, regardless of the validity of any of the input parameters. If not set, the object will not be created if any input is invalid. If a language is not specified or is specified incorrectly (e.g., as a multiple element array), the object will not be created, and an error will be thrown, regardless of the value of this keyword.

Version History

Introduced: 6.1

IDLffLangCat::Query

The IDLffLangCat::Query function method returns the string or string array that corresponds to all supplied key values.

Syntax

Result = *Obj*->[IDLffLangCat::]Query(*Key* [, **DEFAULT_STRING**=*string*])

Return Value

Returns a string or string array that corresponds to all supplied key values. If no key can be found, then the default string or, if no default string is supplied, a null string is returned.

Arguments

Key

A string or string array used as the key or keys against which the query is performed. If a given key does not exist in the current language, the default language is queried (if one exists).

Keywords

DEFAULT_STRING

Set this keyword to a string or string array to be used if the key is not found in the catalog. If a scalar string is supplied, it will be returned for all keys not found. For example, if DEFAULT_STRING is “Key not found,” this string is returned for each failed query on a key or keys. If a string array is supplied, only those values whose indices match the missing keys will be returned. That is, you can have different default strings for different cases of failed query matches (e.g., “No File key found” and “No Tools key found” returned, respectively, for failed queries on keys associated with a File menu and a Tools menu).

Version History

Introduced: 6.1

IDLffLangCat:: SetProperty

The IDLffLangCat::SetProperty procedure method sets the value of a property or group of properties for an IDLffLangCat object.

Syntax

Obj->[IDLffLangCat::]SetProperty [, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLffLangCat Properties](#)” on page 2823 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.1

IDLffMrSID

An IDLffMrSID object class is used to query information about and load image data from a MrSID (.sid) image file.

Note

The IDLffMrSID object is available only under Microsoft Windows.

Superclasses

This class has no superclasses.

Creation

See [IDLffMrSID::Init](#)

Properties

Objects of this class have the following properties. See “[IDLffMrSID Properties](#)” on page 2837 for details on individual properties.

- [CHANNELS](#)
- [DIMENSIONS](#)
- [GEO_ORIGIN](#)
- [GEO_PROJTYPE](#)
- [GEO_RESOLUTION](#)
- [GEO_VALID](#)
- [LEVELS](#)
- [PIXEL_TYPE](#)
- [QUIET](#)
- [TYPE](#)

Methods

This class has the following methods:

- [IDLffMrSID::Cleanup](#)

- [IDLffMrSID::GetDimsAtLevel](#)
- [IDLffMrSID::GetImageData](#)
- [IDLffMrSID::GetProperty](#)
- [IDLffMrSID::Init](#)

Version History

Introduced: 5.5

IDLffMrSID Properties

IDLffMrSID objects have the following properties. Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLffMrSID::Init](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

QUIET

A Boolean value to indicate whether error messages are suppressed while constructing the IDLffMrSID object. If this value is true, the errors messages are suppressed. Otherwise, the messages are printed to the Output Log.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLffMrSID::Cleanup

The IDLffMrSID::Cleanup procedure method deletes all MrSID objects, closing the MrSID file in the process. It also deletes the IDL objects used to communicate with the MrSID library.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffMrSID::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.5

IDLffMrSID::GetDimsAtLevel

The IDLffMrSID::GetDimsAtLevel function method is used to retrieve the dimensions of the image at a given level. This can be used, for example, to determine what level is required to fit the image into a certain area.

Syntax

Result = *Obj*->[IDLffMrSID::]GetDimsAtLevel (*Level*)

Return Value

Returns a floating-point two-element vector containing the dimensions of the image at a given level.

Arguments

Level

A scalar integer specifying the level at which the dimensions are to be determined. This level must be in the range returned by the LEVELS keyword of IDLffMrSID::GetProperty.

Keywords

None

Examples

Initialize the MrSID file object.

```
oFile = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $  
SUBDIRECTORY = ['examples', 'data']))
```

Get the range of levels of resolution contained within the file.

```
oFile->GetProperty, LEVELS = lvl$  
PRINT, lvl$
```

IDL prints,

```
-9 4
```

Print the image dimensions at the lowest image resolution where image level = 4.

```
imgLevelA = MAX(lvls)
dimsAtA = oFile->GetDimsAtLevel(imgLevelA)
PRINT, 'Dimensions of lowest resolution image is', dimsAtA
```

IDL prints,

```
32  32
```

Print the image dimensions at full resolution where image level = 0

```
dimsAtFull = oFile->GetDimsAtLevel(0)
PRINT, 'Dimensions of full resolution image is', dimsAtFull
```

IDL prints,

```
512  512
```

Print the image dimensions at the highest resolution where image level = -9

```
highestLvl = MIN(lvls)
dimsAtHighest = oFile->GetDimsAtLevel(highestLvl)
PRINT, 'Dimensions of highest resolution image is', dimsAtHighest
```

IDL prints,

```
262144  262144
```

Clean up object references.

```
OBJ_DESTROY, [oFile]
```

Version History

Introduced: 5.5

IDLffMrSID::GetImageData

The IDLffMrSID::GetImageData function method extracts and returns the image data from the MrSID file at the specified level and location.

Syntax

```
Result = Obj->[IDLffMrSID::]GetImageData ([, LEVEL = lvl]  
[, SUB_RECT = rect])
```

Return Value

Returns an n -by- w -by- h array containing the image data where n is 1 for grayscale or 3 for RGB images, w is the width and h is the height.

Note

The returned image is ordered bottom-up, the first pixel returned is located at the bottom-left corner of the image. This differs from how data is stored in the MrSID file where the image is top-down, meaning the pixel at the start of the file is located at the top-left corner of the image.

Arguments

None

Keywords

LEVEL

Set this keyword to an integer that specifies the level at which to read the image.

If this keyword is not set, the maximum level is used which returns the minimum resolution (see the LEVELS keyword to IDLffMrSID::GetProperty).

SUB_RECT

Set this keyword to a four-element vector $[x, y, xdim, ydim]$ specifying the position of the lower left-hand corner and the dimensions of the sub-rectangle of the MrSID image to return. This is useful for displaying portions of a high-resolution image.

If this keyword is not set, the whole image will be returned. This may require significant memory if a high-resolution image level is selected.

If the sub-rectangle is greater than the bounds of the image at the selected level the area outside the image bounds will be set to black.

Note

The elements of SUB_RECT are measured in pixels at the current level. This means the point $x = 10$, $y = 10$ at level 1 will be located at $x = 20$, $y = 20$ at level 0 and $x = 5$, $y = 5$ at level 2.

Examples

```

PRO MrSID_GetImageData

; Initialize the MrSID file object.
oFile = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']))

; Get the range of levels of resolution contained within the file.
oFile->GetProperty, LEVELS = lvlS
PRINT, lvlS
; IDL prints, -9, 4

; Get the image data at level 0.
imgDataA = oFile->GetImageData(LEVEL = 0)
HELP, 'image array data at full resolution', imgDataA
; IDL prints, Array[1, 512, 512] indicating a grayscale 512 x 512
array.

; Display the full resolution image.
oImgA = OBJ_NEW('IDLgrImage', imgDataA)
oModelA = OBJ_NEW('IDLgrModel')
oModelA->Add, oImgA
XOBJVIEW, oModelA, BACKGROUND = [0,0,0], $
    TITLE = 'Full Resolution Image', /BLOCK

; Get the image data of a higher resolution image,
imgDataB = oFile->GetImageData(LEVEL = -2)
HELP, imgDataB
; IDL returns [1,2048,2048] indicating a grayscale 2048 x 2048
array.

```

```
; To save processing time, display only a 1024 x 1024 portion of
; the high resolution, using 512,512 as the origin.
imgDataSelect = oFile->GetImageData(LEVEL = -2,$
    SUB_RECT = [512, 512, 1024, 1024])
oImgSelect = OBJ_NEW('IDLgrImage', imgDataSelect)
oModel = OBJ_NEW('IDLgrModel')
oModel->Add, oImgSelect

XOBJVIEW, oModel, BACKGROUND = [0,0,0], $
    TITLE = 'Detail of High Resolution Image', /BLOCK

; Clean up object references.
OBJ_DESTROY, [oFile, oImgA, oModelA, oImgSelect, oModel]

END
```

Version History

Introduced: 5.5

IDLffMrSID::GetProperty

The IDLffMrSID::GetProperty function method is used to query properties associated with the MrSID image.

Syntax

```
Obj->[IDLffMrSID::]GetProperty [, CHANNELS=nChannels]  
    [, DIMENSIONS=Dims] [, LEVELS=Levels] [, PIXEL_TYPE=pixelType]  
    [, TYPE=strType] [, GEO_VALID=geoValid] [, GEO_PROJTYPE=geoProjType]  
    [, GEO_ORIGIN=geoOrigin] [, GEO_RESOLUTION=geoRes]
```

Arguments

None

Keywords

CHANNELS

Set this keyword to a named variable that will contain the number of image bands. For RGB images this is 3, for grayscale it is 1.

DIMENSIONS

Set this keyword equal to a named variable that will contain a two-element long integer array of the form [*width*, *height*] that specifies the dimensions of the MrSID image at level 0 (full resolution).

GEO_ORIGIN

Set this keyword to a named variable that will contain a two-element double precision array of the form [*x*, *y*] that specifies the location of the center of the upper-left pixel.

GEO_PROJTYPE

Set this keyword to a named variable that will contain an unsigned integer that specifies the geoTIFF projected coordinate system type code. For example, type code 32613 corresponds to PCS_WGS84_UTM_zone_13N.

For more information on the geoTIFF file type and available type codes see:

<http://www.remotesensing.org/geotiff/geotiff.html>

GEO_RESOLUTION

Set this keyword to a named variable that will contain a two-element double precision array of the form $[xres, yres]$ that specifies the pixel resolution.

GEO_VALID

Set this keyword to a named variable that will contain a long integer that is set to:

- 1 - If the MrSID image contains valid georeferencing data.
- 0 - If the MrSID image does not contain georeferencing data or the data is in an unsupported format.

Note

Always verify that this keyword returns 1 before using the data returned by any other GEO_* keyword.

LEVELS

Set this keyword equal to a named variable that will contain a two-element long integer array of the form $[minlvl, maxlvl]$ that specifies the range of levels within the current image. Higher levels are lower resolution. A level of 0 equals full resolution. Negative values specify higher levels of resolution.

PIXEL_TYPE

Set this keyword to a named variable that will contain the IDL basic type code for a pixel sample. For a list of the data types indicated by each type code, see the “IDL Type Codes” section of the [SIZE](#) function.

TYPE

Set this keyword to a named variable that will contain a string identifying the file format. This should always be MrSID.

Examples

```

PRO MrSID_GetProperty

; Initialize the MrSID object.
oFile = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']))

; Get the property information of the MrSID file
oFile->GetProperty, CHANNELS = chan, LEVELS = $
    lvlsl, Pixel_Type = pType, TYPE = fileType, GEO_VALID = geoQuery

; Print MrSID file information.
PRINT, 'Number of image channels = ', chan
; IDL returns 1 indicating one image band.

PRINT, 'Range of image levels = ', lvlsl
; IDL returns -9, 4, the minimum and maximum level values.

PRINT, 'Type code of image pixels = ', pType
; IDL returns 1 indicating byte data type.

PRINT, 'Image file type = ', fileType
; IDL returns "MrSID"

PRINT, 'Result of georeferencing data query = ', geoQuery
; IDL returns 0 indicating that the image does not contain
; georeferencing data.

; Destroy object references.
OBJ_DESTROY, [oFile]

END

```

Version History

Introduced: 5.5

IDLffMrSID::Init

The IDLffMrSID::Init function method initializes an IDLffMrSID object containing the image data from a MrSID image file.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = OBJ_NEW('IDLffMrSID', Filename [, PROPERTY=value])
```

or

```
Result = Obj->[IDLffMrSID::]Init(Filename [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Filename

A scalar string argument containing the full path and filename of a MrSID file to be accessed through this IDLffMrSID object.

Note

This is a required argument; it is not possible to create an IDLffMrSID object without specifying a valid MrSID file.

Keywords

Any property listed under [“IDLffMrSID Properties”](#) on page 2837 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Examples

```
oMrSID = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $  
    SUBDIRECTORY = ['examples', 'data']))
```

Version History

Introduced: 5.5

IDLffShape

An IDLffShape object contains geometry, connectivity and attributes for graphics primitives accessed from ESRI Shapefiles. See “[Overview of ESRI Shapefiles](#)” on page 2851 for more details on Shapefiles.

Superclasses

This class has no superclass.

Creation

See [IDLffShape::Init](#)

Properties

Objects of this class have the following properties. See “[IDLffShape Properties](#)” on page 2861 for details on individual properties.

- [ATTRIBUTE_INFO](#)
- [ATTRIBUTE_NAMES](#)
- [DBF_ONLY](#)
- [ENTITY_TYPE](#)
- [FILENAME](#)
- [IS_OPEN](#)
- [N_ATTRIBUTES](#)
- [N_ENTITIES](#)
- [N_RECORDS](#)
- [UPDATE](#)

Methods

This class has the following methods:

- [IDLffShape::AddAttribute](#)
- [IDLffShape::Cleanup](#)

- [IDLffShape::Close](#)
- [IDLffShape::DestroyEntity](#)
- [IDLffShape::GetAttributes](#)
- [IDLffShape::GetEntity](#)
- [IDLffShape::GetProperty](#)
- [IDLffShape::Init](#)
- [IDLffShape::Open](#)
- [IDLffShape::PutEntity](#)
- [IDLffShape::SetAttributes](#)

Version History

Introduced: 5.4

Overview of ESRI Shapefiles

An ESRI Shapefile stores nontopological geometry and attribute information for the spatial features in a data set.

A Shapefile consists of a main file (.shp), an index file (.shx), and a dBASE table (.dbf). For example, the Shapefile “states” would have the following files:

- states.shp
- states.shx
- states.dbf

Naming Conventions for a Shapefile

All the files that comprise an ESRI Shapefile must adhere to the 8.3 filename convention and must be lower case. The main file, index file, and dBASE file must all have the same prefix. The prefix must start with an alphanumeric character and can contain any alphanumeric, underscore (_), or hyphen (-). The main file suffix must use the .shp extension, the index file the .shx extension, and the dBASE table the .dbf extension.

Major Elements of a Shapefile

A Shapefile consists of the following elements that you can access through the IDLffShape class:

- Entities
- Attributes

Entities

The geometry for a feature is stored as a shape comprising a set of vector coordinates (referred to as ‘entities’). The entities in a Shapefile must all be of the same type. The following are the possible types for entities in a Shapefile:

Shape Type	Type Code
Point	1
PolyLine	3
Polygon	5

Table 6-16: Entity Types

Shape Type	Type Code
MultiPoint	8
PointZ	11
PolyLineZ	13
PolygonZ	15
MultiPointZ	18
PointM	21
PolyLineM	23
PolygonM	25
MultiPointM	28
MultiPatch	31

Table 6-16: Entity Types (Continued)

When retrieving entities using the `IDLffShape::GetEntity` method, an IDL structure is returned. This structure has the following fields:

Field	Data Type
SHAPE_TYPE	IDL_LONG
ISHAPE	IDL_LONG
BOUNDS	Double[8]
N_VERTICES	IDL_LONG
VERTICES	Pointer (to Vertices array)
MEASURE	Pointer (to Measure array)
N_PARTS	IDL_LONG
PARTS	Pointer (to Parts array).
PART_TYPES	Pointer (to part types)
ATTRIBUTES	Pointer to attribute array.

Table 6-17: Entity Structure Field Data Types

The following table describes each field in the structure:

Field	Description
SHAPE_TYPE	The entity type.
ISHAPE	A read-only field containing the index of the specific entity within the shape object. This value is set automatically when the shape file is written.
BOUNDS	<p>A bounding box that specifies the range limits of the entity. This eight element array contains the following information:</p> <ul style="list-style-type: none"> • Index 0 — X minimum value • Index 1 — Y minimum value • Index 2 — Z minimum value (if Z is supported by type) • Index 3 — Measure minimum value (if measure is supported by entity type) • Index4 — X maximum value • Index5 — Y maximum value • Index6 — Z maximum value (if Z is supported by the entity type) • Index7 — Measure maximum value (if measure is supported by entity type) <p>Note - If the entity is a point type, the values contained in the bounds array are also the values of the entity.</p>
N_VERTICES	The number of vertices in the entity. If this value is one and the entity is a POINT type (POINT, POINTM, POINTZ), the vertices pointer will be set to NULL and the entity value will be maintained in the BOUNDS field.

Table 6-18: Entity Structure Field Descriptions

Field	Description
VERTICES	<p>An IDL pointer that contains the vertices of the entity. This pointer contains a double array that has one of the following formats:</p> <ul style="list-style-type: none"> • [2, <i>N</i>] - If Z data is not present • [3, <i>N</i>] - If Z data is present <p>where <i>N</i> is the number of vertices. These array formats can be passed to the polygon and polyline objects of IDL Object Graphics.</p> <p>Note - This pointer will be null if the entity is a point type, with the values maintained in the BOUNDS array.</p>
MEASURE	<p>If the entity has a measure value (this is dependent on the entity type), this IDL pointer will contain a vector array of measure values. The length of this vector is N_VERTICES.</p> <p>Note - This pointer will be null if the entity is of type POINTM, with the values contained in the BOUNDS array.</p>
N_PARTS	<p>If the values of the entity are separated into parts, the break points are enumerated in the parts array. This field lists the number of parts in this entity. If this value is 0, the entity is one part and the PARTS pointer will be NULL.</p>
PARTS	<p>An IDL pointer that contains an array of indices into the vertex/measure arrays. These values represent the start of each part of the entity. The index range of each entity part is defined by the following:</p> <ul style="list-style-type: none"> • Start = Parts[I] • End = Parts[I+1]-1 or the end of the array

Table 6-18: Entity Structure Field Descriptions (Continued)

Field	Description
PART_TYPES	<p>This IDL pointer is only valid for entities of type MultiPatch and defines the type of the particular part. If the entity type is not MultiPatch, part types are assumed to be type RING (SHPP_RING).</p> <p>Note - This pointer is NULL if the entity is not type MultiPatch.</p>
ATTRIBUTES	<p>If the attributes for an entity were requested, this field contains an IDL pointer that contains a structure of attributes for the entity. For more information on this structure, see “Attributes” on page 2855.</p>

Table 6-18: Entity Structure Field Descriptions (Continued)

Attributes

A Shapefile provides the ability to associate information describing each entity (a geometric element) contained in the file. This descriptive information, called attributes, consists of a set of named data elements for each geometric entity contained in the file. The set of available attributes is the same for every entity contained in a Shapefile, with each entity having its own set of attribute values.

An attribute consists of two components:

- A name
- A data value

The name consists of an 11 character string that is used to identify the data value. The data value is not limited to any specific format.

The two components that form an attribute are accessed differently using the shape object. To get the name of attributes for the specific file, the `ATTRIBUTE_NAMES` keyword to the `IDLffShape::GetProperty` method is used. This returns a string array that contains the names for the attributes defined for the file.

To get the attribute values for an entity, the `IDLffShape::GetAttributes` method is called or the `ATTRIBUTES` keyword of the `IDLffShape::GetEntity` method is set. In each case, the attribute values for the specified entity is returned as an anonymous IDL structure. The numeric order of the fields in the returned structure map to the numeric order of the attributes defined for the file. The actual format of the returned structure is:

```
ATTRIBUTE_0 : VALUE,
ATTRIBUTE_1 : VALUE,
ATTRIBUTE_2 : VALUE,
...
ATTRIBUTE_<N-1> : VALUE
```

To access the values in the returned structure, you can either hard code the structure field names or use the structure indexing feature of IDL.

Accessing Shapefiles

The following example shows how to access data in a Shapefile. This example sets up a map to display parts of a Shapefile, opens a Shapefile, reads the entities from the Shapefile, and then plots only the state of Colorado:

```
PRO ex_shapefile

DEVICE, RETAIN=2, DECOMPOSED=0
!P.BACKGROUND=255

;Define a color table
r=BYTARR(256) & g=BYTARR(256) & b=BYTARR(256)
r[0]=0 & g[0]=0 & b[0]=0           ;Definition of black
r[1]=100 & g[1]=100 & b[1]=255    ;Definition of blue
r[2]=0 & g[2]=255 & b[2]=0        ;Definition of green
r[3]=255 & g[3]=255 & b[3]=0      ;Definition of yellow
r[255]=255 & g[255]=255 & b[255]=255 ;Definition of white

TVLCT, r, g, b
black=0 & blue=1 & green=2 & yellow=3 & white=255

; Set up map to plot Shapefile on
MAP_SET, /ORTHO,45, -120, /ISOTROPIC, $
/HORIZON, E_HORIZON={FILL:1, COLOR:blue}, $
/GRID, COLOR=black, /NOBORDER

; Fill the continent boundaries:
MAP_CONTINENTS, /FILL_CONTINENTS, COLOR=green
```

```
; Overplot coastline data:
MAP_CONTINENTS, /COASTS, COLOR=black

; Show national borders:
MAP_CONTINENTS, /COUNTRIES, COLOR=black

; Open the states Shapefile in the examples directory
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

; Get the number of entities so we can parse through them
myshape->IDLffShape::GetProperty, N_ENTITIES=num_ent

; Parsing through the entities and only plotting the state of
; Colorado
FOR x=1, (num_ent-1) DO BEGIN
    ; Get the Attributes for entity x
    attr = myshape->IDLffShape::GetAttributes(x)
    ; See if 'Colorado' is in ATTRIBUTE_1 of the attributes for
    ; entity x
    IF attr.ATTRIBUTE_1 EQ 'Colorado' THEN BEGIN
        ; Get entity
        ent = myshape->IDLffShape::GetEntity(x)
        ; Plot entity
        POLYFILL, (*ent.vertices)[0,*], (*ent.vertices)[1,*],
        COLOR=yellow
        ; Clean-up of pointers
        myshape->IDLffShape::DestroyEntity, ent
    ENDIF
ENDFOR

; Close the Shapefile
OBJ_DESTROY, myshape

END
```

This results in the following:

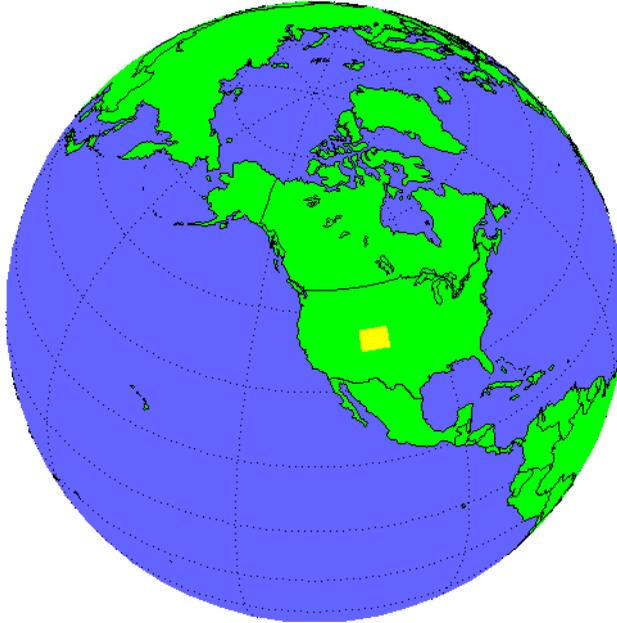


Figure 6-1: Example Use of Shapefiles

Creating New Shapefiles

To create a Shapefile, you need to create a new Shapefile object, define the entity and attributes definitions, and then add your data to the file. For example, the following program creates a new Shapefile (`cities.shp`), defines the entity type to be “Point”, defines 2 attributes (`CITY_NAME` and `STATE_NAME`), and then adds an entity to the new file:

```
PRO ex_shapefile_newfile

;Create the new shapefile and define the entity type to Point
mynewshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
  SUBDIR=['examples', 'data']), /UPDATE, ENTITY_TYPE=1)

;Set the attribute definitions for the new Shapefile
mynewshape->IDLffShape::AddAttribute, 'CITY_NAME', 7, 25, $
  PRECISION=0
mynewshape->IDLffShape::AddAttribute, 'STAT_NAME', 7, 25, $
  PRECISION=0
```

```
;Create structure for new entity
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.BOUNDS[0] = -104.87270
entNew.BOUNDS[1] = 39.768040
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -104.87270
entNew.BOUNDS[5] = 39.768040
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

;Create structure for new attributes
attrNew = mynewshape->IDLffShape::GetAttributes( $
/ATTRIBUTE_STRUCTURE)

;Define the values for the new attributes
attrNew.ATTRIBUTE_0 = 'Denver'
attrNew.ATTRIBUTE_1 = 'Colorado'

;Add the new entity to new shapefile
mynewshape->IDLffShape::PutEntity, entNew

;Add the Colorado attributes to new shapefile
mynewshape->IDLffShape::SetAttributes, 0, attrNew

;Close the shapefile
OBJ_DESTROY, mynewshape

END
```

Updating Existing Shapefiles

You can modify existing Shapefiles with the following:

- Adding new entities
- Adding new attributes (only to Shapefiles without any existing values in any attributes)
- Modifying existing attributes

Note

You cannot modify existing entities.

For example, the following program adds an entity and attributes for the city of Boulder to the `cities.shp` file we created in the previous example:

```
PRO ex_shapefile_modify

;Open the cities Shapefile
myshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE)

;Create structure for new entity
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.BOUNDS[0] = -105.25100
entNew.BOUNDS[1] = 40.026878
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -105.25100
entNew.BOUNDS[5] = 40.026878
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

;Create structure for new attributes
attrNew = myshape->IDLffShape::GetAttributes( $
/ATTRIBUTE_STRUCTURE)

;Define the values for the new attributes
attrNew.ATTRIBUTE_0 = 'Boulder'
attrNew.ATTRIBUTE_1 = 'Colorado'

;Add the new entity to new shapefile
myshape->IDLffShape::PutEntity, entNew

;Add the Colorado attributes to new shapefile
myshape->IDLffShape::SetAttributes, 0, attrNew

;Close the shapefile
OBJ_DESTROY, myshape

END
```

IDLffShape Properties

IDLffShape objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLffShape::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLffShape::Init](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ATTRIBUTE_INFO

A array of structures containing the attribute information for each attribute. The attribute information structures have the following fields:

Field	Description
NAME	A string that contains the name of the attribute.
TYPE	The IDL type code of the attribute.
WIDTH	The width of the attribute.
PRECISION	The precision of the attribute.

Table 6-19: ATTRIBUTE_INFO Fields

The file must be open to obtain this information.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ATTRIBUTE_NAMES

A string array containing the names of each attribute in the Shapefile object.

Property Type	String array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DBF_ONLY

A non-zero, positive integer value indicating whether the underlying dBASE table (.dbf) component of the shapefile is opened while all other entity related files are left closed. The following two values are accepted for this property:

- 1 - Open an existing .dbf file,
- Greater than 1 - Create a new .dbf file

Note

The UPDATE keyword is required to open the .dbf file for updating.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

ENTITY_TYPE

If retrieving this property, its value is an integer representing the type code for the entities contained in the Shapefile object. If the value is unknown, this method returns -1. For more information on entity type codes, see “[Entities](#)” on page 2851.

If setting this property, its value is an integer representing the entity type of a new Shapefile. Use this setting only when creating a new Shapefile. For more information on entity types, see “[Entities](#)” on page 2851.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

FILENAME

A string representing the fully qualified path name of the Shapefile in the current Shapefile object.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IS_OPEN

An integer value representing information about the status of a Shapefile. The following values can be returned:

Value	Description
0	File is not open.
1	File is open in read-only mode.
3	File is open in update mode.

Table 6-20: IS_OPEN Values

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

N_ATTRIBUTES

A longword integer representing the number of attributes associated with a Shapefile object. If the number of attributes is unknown, this property returns 0.

Property Type	Long		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

N_ENTITIES

A longword integer representing the number of entities contained in Shapefile object. If the number of entities is unknown, this property returns 0.

Property Type	Long		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

N_RECORDS

A longword integer representing the number of records in the dBASE table (.dbf) component of the Shapefile. In a normal operating mode, this process is accomplished by getting the number of entities. However, in DBF_ONLY mode, no entity file exists.

Property Type	Long		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

UPDATE

A Boolean value indicating whether the file opened for writing. The file is opened for writing if this property is set to true. The default is read-only.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLffShape::AddAttribute

The IDLffShape::AddAttribute method adds an attribute definition to a Shapefile. Adding the attribute definition is required before adding the actual attribute data to a file. For more information on attributes, see [“Attributes”](#) on page 2855.

Note

You can only define new attributes to Shapefiles that do not have any existing values in any attributes.

Syntax

Obj->[IDLffShape::]AddAttribute, *Name*, *Type*, *Width* [, **PRECISION**=*integer*]

Arguments

Name

Set to a string that contains the attribute name. Name values are limited to 11 characters. Arguments longer than 11 characters will be truncated.

Type

Set to the IDL type code that corresponds to the data type that will be stored in the attribute. The valid types are:

Code	Description
3	Longword Integer
5	Double-precision floating-point
7	String

Table 6-21: Type Code Descriptions

Width

Set to the width of the field for the data value of the attribute. The following table describes the possible values depending on the defined Type:

Field Type	Valid Values
Longword Integer	Maximum size of the field.
Double-precision floating-point	Maximum size of the field.
String	Maximum length of the string.

Table 6-22: Width Values

Keywords

PRECISION

Set this keyword to the number of positions to be included after the decimal point. The default is 8. This keyword is only valid for fields defined as double-precision floating-point.

Examples

In the following example, we add the attribute “ELEVATION” to an existing Shapefile. Note that if the file already contains data in an attribute for any of the entities defined in the file, this operation will fail.

```
PRO ex_addattr_shapefile

;Open a shapefile
myshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
  SUBDIR=['examples', 'data']), /UPDATE)

;Define a new attribute for the Shapefile
myshape->IDLffShape::AddAttribute, 'ELEVATION', 3, 4, $
  PRECISION=0

;Close the shapefile
OBJ_DESTROY, myshape

END
```

Version History

Introduced: 5.4

IDLffShape::Cleanup

The IDLffShape::Cleanup procedure method performs all cleanup on a Shapefile object. If the Shapefile being accessed by the object is open and the file has been modified, the new information is written to the file if one of the following conditions is met:

- The file was opened with write permissions using the UPDATE keyword to the IDLffShape::Open method
- It is a newly created file that has not been written previously.

Note

Cleanup methods are special lifecycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffShape::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.4

IDLffShape::Close

The IDLffShape::Close procedure method closes a Shapefile. If the file has been modified, it is also written to the disk if neither of the following conditions is met:

- The file was opened with write permissions using the UPDATE keyword to the IDLffShape::Open method
- It is a newly created file that has not been written previously.

If the file has been modified and one of the previous conditions is not met, the file is closed and the changes are not written to disk.

Syntax

Obj->[IDLffShape::]Close

Arguments

None

Keywords

None

Version History

Introduced: 5.4

IDLffShape::DestroyEntity

The IDLffShape::DestroyEntity procedure method frees memory associated with the entity structure. For more information on the entity structure, see “Entities” on page 2851.

Syntax

Obj->[IDLffShape::]DestroyEntity, *Entity*

Arguments

Entity

A scalar or array of entities to be destroyed.

Keywords

None

Examples

In the following example, all of the entities from the `states.shp` Shapefile are read and then the DestroyEntity method is called to clean up all pointers:

```
PRO ex_shapefile

; Open the states Shapefile in the examples directory.
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

; Get the number of entities so we can parse through them.
myshape->IDLffShape::GetProperty, N_ENTITIES=num_ent

; Read all the entities.
FOR x=1, (num_ent-1) DO BEGIN
    ;Read the entity x
    ent = myshape->IDLffShape::GetEntity(x)
    ;Clean-up of pointers
    myshape->IDLffShape::DestroyEntity, ent
ENDFOR

; Close the Shapefile.
OBJ_DESTROY, myshape
END
```

Version History

Introduced: 5.4

IDLffShape::GetAttributes

The IDLffShape::GetAttributes function method retrieves the attributes for the entities you specify from a Shapefile.

Syntax

```
Result = Obj->[IDLffShape:]GetAttributes([Index] [, /ALL]  
[, /ATTRIBUTE_STRUCTURE] )
```

Return Value

Returns an anonymous structure array. For more information on the structure, see “Attributes” on page 2855.

Arguments

Index

A scalar or array of longs specifying the entities for which you want to retrieve the attributes, with 0 being the first entity in the Shapefile.

Note

If you do not specify *Index* and the ALL keyword is not set, the attributes for the first entity (0) are returned.

Keywords

ALL

Set this keyword to retrieve the attributes for all entities in a Shapefile. If you set this keyword, the *Index* argument is not required.

ATTRIBUTE_STRUCTURE

Set this keyword to return an empty attribute structure that can then be used with the [IDLffShape::SetAttributes](#) method to add attributes to a Shapefile.

Examples

In the first example, we retrieve the attributes associated with entity at location 0 (the first entity in the file):

```
attr = myShape->getAttributes( 0)
```

In the next example, we retrieve the attributes associated with entities 10 through 20:

```
attr = myShape->getAttributes( 10+indgen(11) )
```

In the next example, we retrieve the attributes for entities 1,4, 9 and 70:

```
attr = myShape->getAttributes( [1, 4, 9, 70] )
```

In the next example, we retrieve all the attributes for a Shapefile:

```
attr = myShape->getAttributes( /ALL )
```

Version History

Introduced: 5.4

IDLffShape::GetEntity

The IDLffShape::GetEntity function method returns the entities you specify from a Shapefile.

Syntax

Result = Obj->[IDLffShape::]GetEntity([Index] [, /ALL] [, /ATTRIBUTES])

Return Value

Returns a type {IDL_SHAPE_ENTITY} structure array. For more information on the structure, see “Entities” on page 2851.

Note

Since an entity structure contains IDL pointers, you must free all the pointers returned in these structures when the entity is no longer needed using the [IDLffShape::DestroyEntity](#) method.

Note

Since entities cannot be modified in a Shapefile, an entity is read directly from the Shapefile each time you use the IDLffShape::GetEntity method even if you have already read that entity. If you modify the structure array returned by this method for a given entity and then use IDLffShape::GetEntity on that same entity, the modified data will NOT be returned, the data that is actually written in the file is returned.

Arguments

Index

A scalar or array of longs specifying the entities for which you want to retrieve with 0 being the first entity in the Shapefile. If the ALL keyword is set, this argument is not required. If you do not specify any entities and the ALL keyword is not set, the first entity (0) is returned.

Keywords

ALL

Set this keyword to retrieve all entities from the Shapefile. If this keyword is set, the Index argument is not required.

ATTRIBUTES

Set this keyword to return the attributes in the entity structure. If not set, the ATTRIBUTES tag in the entity structure will be a null IDL pointer.

Examples

In the following example, all of the entities from the `states.shp` Shapefile are read:

```
PRO ex_shapefile

; Open the states Shapefile in the examples directory.
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
  SUBDIR=['examples', 'data']))

; Get the number of entities so we can parse through them.
myshape->IDLffShape::GetProperty, N_ENTITIES=num_ent

; Read all the entities.
FOR x=1, (num_ent-1) DO BEGIN
  ;Read the entity x
  ent = myshape->IDLffShape::GetEntity(x)
  ;Clean-up of pointers
  myshape->IDLffShape::DestroyEntity, ent
ENDFOR

; Close the Shapefile.
OBJ_DESTROY, myshape

END
```

Version History

Introduced: 5.4

IDLffShape::GetProperty

The IDLffShape::GetProperty procedure method returns the values of properties associated with a Shapefile object. These properties are:

- Number of entities
- The type of the entities
- The number of attributes associated with each entity
- The names of the attributes
- The name, type, width, and precision of the attributes
- The status of a Shapefile
- The filename of the Shapefile object

Syntax

Obj->[IDLffShape::]GetProperty [, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “IDLffShape Properties” on page 2861 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Examples

In the following example, the number of entities and the entity type is returned:

```
PRO entity_info

; Open the states Shapefile in the examples directory.
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
  SUBDIR=['examples', 'data']))

; Get the number of entities and the entity type.
myshape->IDLffShape::GetProperty, N_ENTITIES=num_ent, $
  ENTITY_TYPE=ent_type
```

```

; Print the number of entities and the type.
PRINT, 'Number of Entities: ', num_ent
PRINT, 'Entity Type: ', ent_type

; Close the Shapefile.
OBJ_DESTROY, myshape

END

```

This results in the following:

```

Number of Entities:      51
Entity Type:             5

```

In the next example, the definitions for attribute 1 are returned:

```

PRO attribute_info

; Open the states Shapefile in the examples directory.
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

; Get the info for all attribute.
myshape->IDLffShape::GetProperty, ATTRIBUTE_INFO=attr_info

; Print Attribute Info.
PRINT, 'Attribute Number: ', '1'
PRINT, 'Attribute Name: ', attr_info[1].name
PRINT, 'Attribute Type: ', attr_info[1].type
PRINT, 'Attribute Width: ', attr_info[1].width
PRINT, 'Attribute Precision: ', attr_info[1].precision

; Close the Shapefile.
OBJ_DESTROY, myshape

END

```

This results in the following:

```

Attribute Number:      1
Attribute Name:       STATE_NAME
Attribute Type:       7
Attribute Width:      25
Attribute Precision:  0

```

Version History

Introduced: 5.4

N_RECORDS keyword: 5.6

IDLffShape::Init

The IDLffShape::Init function method initializes or constructs a Shapefile object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = OBJ_NEW('IDLffShape' [, Filename] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLffShape::]Init([, Filename] [, PROPERTY=value])
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Filename

A scalar string containing the full path and filename of a Shapefile (.shp) to open. If this file exists, it is opened. If the file does not exist, a new Shapefile object is constructed. You do not need to use [IDLffShape::Open](#) to open an existing file when specifying this argument.

Note

The .shp, .shx, and .dbx files must exist in the same directory for you to be able to open and access the file unless the UPDATE keyword is set.

Keywords

Any property listed under “[IDLffShape Properties](#)” on page 2861 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Examples

In the following example, we create a new Shapefile object and open the `examples/data/states.shp` file:

```
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))
```

Version History

Introduced: 5.4

DBF_ONLY keyword: 5.6

IDLffShape::Open

The IDLffShape::Open function method opens a specified Shapefile.

Syntax

```
Result = Obj->[IDLffShape:]Open( 'Filename' [, /DBF_ONLY]  
    [, ENTITY_TYPE='value'] [, /UPDATE] )
```

Return Value

Returns 1 if the file can be read successfully. If not able to open the file, it returns 0.

Arguments

Filename

A scalar string containing the full path and filename of a Shapefile (.shp) to open. Note that the .shp, .shx, and .dbx files must exist in the same directory for you to be able to open and access the file unless the UPDATE keyword is set.

Keywords

DBF_ONLY

If this keyword is set to a positive value, only the underlying dBASE table (.dbf) component of the shapefile is opened. All entity related files are left closed. Two values to this keyword are accepted: 1 - Open an existing .dbf file, > 1 - Create a new .dbf file

The UPDATE keyword is required to open the .dbf file for updating.

ENTITY_TYPE

Set this keyword to the entity type of a new Shapefile. Use this keyword only when creating a new Shapefile. For more information on entity types, see [“Entities”](#) on page 2851.

UPDATE

Set this keyword to have the file opened for writing. The default is read-only.

Examples

In the following example, the file `examples/data/states.shp` is opened for reading and writing:

```
status = myShape->Open(FILEPATH('states.shp', $  
    SUBDIR=['examples', 'data']), /UPDATE)
```

Version History

Introduced: 5.4

DBF_ONLY keyword: 5.6

IDLffShape::PutEntity

The IDLffShape::PutEntity procedure method inserts an entity into the Shapefile object. The entity must be in the proper structure. For more information on the structure, see “Entities” on page 2851.

Note

The shape type of the new entity must be the same as the shape type defined for the Shapefile. If the shape type has not been defined for the Shapefile using the ENTITY_TYPE keyword for the IDLffShape::Open or IDLffShape::Init methods, the first entity that is inserted into the Shapefile defines the type.

Note

Only new entities can be inserted into a Shapefile. Existing entities cannot be updated.

Syntax

Obj->[IDLffShape::]PutEntity, *Data*

Arguments

Data

A scalar or an array of entity structures.

Keywords

None

Examples

In the following example, we create a new shapefile, define a new entity, and then use the PutEntity method to insert it into the new file:

```
PRO ex_shapefile_newfile

; Create the new shapefile and define the entity type to Point.
mynewshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
SUBDIR=['examples', 'data']), /UPDATE, ENTITY_TYPE=1)
```

```
; Create structure for new entity.
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity.
entNew.SHAPE_TYPE = 1
entNew.BOUNDS[0] = -104.87270
entNew.BOUNDS[1] = 39.768040
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -104.87270
entNew.BOUNDS[5] = 39.768040
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

; Add the new entity to new shapefile.
mynewshape->IDLffShape::PutEntity, entNew

; Close the shapefile.
OBJ_DESTROY, mynewshape

END
```

Version History

Introduced: 5.4

IDLffShape::SetAttributes

The IDLffShape::SetAttributes procedure method sets the attributes for a specified entity in a Shapefile object.

Syntax

Obj->[IDLffShape::]SetAttributes, *Index*, *Attribute_Num*, *Value*

or

Obj->[IDLffShape::]SetAttributes, *Index*, *Attributes*

Arguments

Attribute_Num

The field number for the attribute whose value is being set. This value is 0-based.

Attributes

An Attribute structure whose fields match the fields in the attribute table. If *Attributes* is an array, the entities specified in *Index*, up to the size of the Attributes array, are set. Using this feature, all the attribute values of a set of entities can be set for a Shapefile.

The type of this Attribute structure must match the type that is generated internally for Attribute table. To get a copy of this structure, either get the attribute set for an entity or get the definition using the ATTRIBUTE_STRUCTURE keyword of the [IDLffShape::GetProperty](#) method.

Index

A scalar specifying the entity in which you want to set the attributes. The first entity in the Shapefile object is 0.

Value

The value that the attribute is being set to. If the value is not of the correct type, type conversion is attempted.

If *Value* is an array and *Index* is a scalar, the value of record is treated as a starting point. Using this feature, all the attribute values of a specific field can be set for a Shapefile.

Keywords

None

Examples

In the following example, we create a new shapefile, define the attributes for the new file, define a new entity, define some attributes, insert the new entity, and then use the `SetAttributes` method to insert the attributes into the new file:

```
PRO ex_shapefile_newfile

; Create the new shapefile and define the entity type to Point.
mynewshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE, ENTITY_TYPE=1)

; Set the attribute definitions for the new Shapefile.
mynewshape->IDLffShape::AddAttribute, 'CITY_NAME', 7, 25, $
    PRECISION=0
mynewshape->IDLffShape::AddAttribute, 'STAT_NAME', 7, 25, $
    PRECISION=0

; Create structure for new entity.
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.BOUNDS[0] = -104.87270
entNew.BOUNDS[1] = 39.768040
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -104.87270
entNew.BOUNDS[5] = 39.768040
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

; Create structure for new attributes.
attrNew = mynewshape->IDLffShape::GetAttributes( $
    /ATTRIBUTE_STRUCTURE)

; Define the values for the new attributes.
attrNew.ATTRIBUTE_0 = 'Denver'
attrNew.ATTRIBUTE_1 = 'Colorado'

; Add the new entity to new shapefile.
mynewshape->IDLffShape::PutEntity, entNew
```

```
; Add the Colorado attributes to new shapefile.  
mynewshape->IDLffShape::SetAttributes, 0, attrNew  
  
; Close the shapefile.  
OBJ_DESTROY, mynewshape  
  
END
```

Version History

Introduced: 5.4

IDLffXMLDOM Classes

IDL's XML Document Object Model (DOM) support comprises the following IDL object classes. For more information on using the XML DOM classes, see [Chapter 26, “Using the XML DOM Object Classes”](#) in the *Building IDL Applications* manual.

- [“IDLffXMLDOMAttr”](#) on page 2888
- [“IDLffXMLDOMCDATASection”](#) on page 2897
- [“IDLffXMLDOMCharacterData”](#) on page 2901
- [“IDLffXMLDOMComment”](#) on page 2914
- [“IDLffXMLDOMDocument”](#) on page 2918
- [“IDLffXMLDOMDocumentFragment”](#) on page 2940
- [“IDLffXMLDOMDocumentType”](#) on page 2945
- [“IDLffXMLDOMEElement”](#) on page 2952
- [“IDLffXMLDOMEntity”](#) on page 2965
- [“IDLffXMLDOMEntityReference”](#) on page 2972
- [“IDLffXMLDOMNamedNodeMap”](#) on page 2976
- [“IDLffXMLDOMNode”](#) on page 2986
- [“IDLffXMLDOMNodeList”](#) on page 3014
- [“IDLffXMLDOMNotation”](#) on page 3020
- [“IDLffXMLDOMProcessingInstruction”](#) on page 3026
- [“IDLffXMLDOMText”](#) on page 3033

IDLffXMLDOMAttr

The IDLffXMLDOMAttr object represents an attribute that is part of an element object. Typically, the allowable values for the attribute are defined in a DTD.

Attribute objects are subclasses of the [IDLffXMLDOMNode](#) abstract superclass, but are not actually child nodes of the elements they describe and are not considered as part of the DOM tree. Therefore, an attribute object does not have a parent or siblings, and so the `ParentNode()`, `NextSibling()`, and `PreviousSibling()` methods always return a null object reference. The DOM takes the view that attributes are properties of elements, rather than having a separate identity.

An attribute's value may be explicitly assigned or may take on a default value as described by a DTD. If neither of these alternatives is true, the attribute does not exist in the element. Use the [IDLffXMLDOMAttr::GetSpecified](#) method to determine if the value was explicitly specified or has taken on the default value.

If the value of an attribute may contain entity references, then the attribute node may have either text or entity-reference child nodes.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMAttr::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMAttr::Cleanup](#)
- [IDLffXMLDOMAttr::GetName](#)
- [IDLffXMLDOMAttr::GetSpecified](#)
- [IDLffXMLDOMAttr::GetValue](#)

- [IDLffXMLDOMAttr::Init](#)
- [IDLffXMLDOMAttr::SetValue](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMAttr Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMAttr::Cleanup

The IDLffXMLDOMAttr::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMAttr::GetName

The IDLffXMLDOMAttr::GetName function method returns the name of the attribute node.

Syntax

Result = *Obj*->[IDLffXMLDOMAttr::]GetName()

Return Value

Returns a scalar string containing the name of the attribute node.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMAttr::GetSpecified

The IDLffXMLDOMAttr::GetSpecified function method returns a scalar integer indicating how the value of the attribute node was set.

Syntax

Result = *Obj*->[IDLffXMLDOMAttr::]GetSpecified()

Return Value

Returns 1 if the value of the attribute node was explicitly set in the XML document or was set programmatically with the [IDLffXMLDOMAttr::SetValue](#) method. Returns 0 if the attribute's value was established from the default value declared in the document's DTD.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMAttr::GetValue

The IDLffXMLDOMAttr::GetValue function method returns the value of the attribute node.

Syntax

Result = *Obj*->[IDLffXMLDOMAttr::]GetValue()

Return Value

Returns a scalar string containing the value of the attribute node.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMAttr::Init

The IDLffXMLDOMAttr::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetAttributes](#) are used to create objects of this class.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMAttr::SetValue

The IDLffXMLDOMAttr::SetValue procedure method sets the value of the attribute node, using the *Value* argument.

Syntax

Obj->[IDLffXMLDOMAttr::]SetValue, *Value*

Arguments

Value

A scalar string used to set the value of the attribute node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCDATASection

The IDLffXMLDOMCDATASection object is used to escape blocks of text containing characters that would otherwise be regarded as markup. The only delimiter that is recognized in a CDATA section is the “]]>” string that ends the CDATA section. CDATA sections cannot be nested. The primary purpose is for including material such as XML fragments without needing to escape all the delimiters.

There are no methods specific to this class except Cleanup and Init.

Superclasses

[IDLffXMLDOMNode](#)

[IDLffXMLDOMCharacterData](#)

[IDLffXMLDOMText](#)

Creation

See [IDLffXMLDOMCDATASection::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMCDATASection::Cleanup](#)
- [IDLffXMLDOMCDATASection::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMCDATASection Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMCDATASection::Cleanup

The IDLffXMLDOMCDATASection::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCDATASection::Init

The IDLffXMLDOMCDATASection::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMCharacterData

The IDLffXMLDOMCharacterData class extends the [IDLffXMLDOMNode](#) class with a set of methods for accessing character data in the DOM tree. This is an abstract class in the sense that there will never be an instance of this class, but its methods are available to its subclasses.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMCharacterData::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMCharacterData::AppendData](#)
- [IDLffXMLDOMCharacterData::Cleanup](#)
- [IDLffXMLDOMCharacterData::DeleteData](#)
- [IDLffXMLDOMCharacterData::GetData](#)
- [IDLffXMLDOMCharacterData::GetLength](#)
- [IDLffXMLDOMCharacterData::Init](#)
- [IDLffXMLDOMCharacterData::InsertData](#)
- [IDLffXMLDOMCharacterData::ReplaceData](#)
- [IDLffXMLDOMCharacterData::SetData](#)
- [IDLffXMLDOMCharacterData::SubstringData](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMCharacterData::AppendData

The IDLffXMLDOMCharacterData::AppendData procedure method appends a string to the character data in the node.

Syntax

Obj->[IDLffXMLDOMCharacterData:]AppendData, *String*

Arguments

String

A scalar string to be added at the end of the character data.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::Cleanup

The IDLffXMLDOMCharacterData::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::DeleteData

The IDLffXMLDOMCharacterData::DeleteData procedure method deletes a number of characters from the character data in the node, starting at an offset.

Syntax

Obj->[IDLffXMLDOMCharacterData:]DeleteData, *Offset*, *Count*

Arguments

Offset

A scalar value that indicates where to begin deleting characters.

Count

A scalar number of characters to delete.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::GetData

The IDLffXMLDOMCharacterData::GetData function method returns the character data in the node.

Syntax

Result = *Obj*->[IDLffXMLDOMCharacterData::]GetData()

Return Value

Returns a scalar string containing the character data in the node.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::GetLength

The IDLffXMLDOMCharacterData::GetLength function method returns the number of characters in the node.

Syntax

Result = *Obj*->[IDLffXMLDOMCharacterData::]GetLength()

Return Value

Returns a scalar string containing the number of characters in the node. This value may be zero.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::Init

The IDLffXMLDOMCharacterData::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMCharacterData::InsertData

The IDLffXMLDOMCharacterData::InsertData procedure method inserts a string in the node's character data at a specified offset.

Syntax

Obj->[IDLffXMLDOMCharacterData:]InsertData, *Offset*, *String*

Arguments

Offset

A scalar value that indicates where in the character data to begin inserting the string contained in *String*.

String

A scalar string to be inserted in the character data at the point indicated by *Offset*.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::ReplaceData

The IDLffXMLDOMCharacterData::ReplaceData procedure method replaces a number of characters, starting at an offset in the node's character data, with a string.

Syntax

Obj->[IDLffXMLDOMCharacterData:]ReplaceData, *Offset*, *Count*, *String*

Arguments

Offset

A scalar value that indicates where in the character data to begin replacement.

Count

A scalar number of characters in the character data to replace.

String

A scalar string to be inserted in the character data at the point indicated by *Offset*.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::SetData

The IDLffXMLDOMCharacterData::SetData procedure method sets the node's character data to a string.

Syntax

Obj->[IDLffXMLDOMCharacterData:]SetData, *String*

Arguments

String

A scalar string to be set in the node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMCharacterData::SubstringData

The IDLffXMLDOMCharacterData::SubstringData function method returns a string composed of a substring of the node's character data.

Syntax

Result = *Obj*->[IDLffXMLDOMCharacterData::]SubstringData(*Offset*, *Count*)

Return Value

Returns a string containing a substring of the node's character data that starts at *Offset* and has a length specified by *Count*.

Arguments

Offset

A scalar value that indicates where in the character data to begin the substring.

Count

A scalar length of the substring.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMComment

The IDLffXMLDOMComment class represents the content of a comment, the characters between the starting “<!--” and the ending “-->”.

There are no methods specific to this class except Cleanup and Init.

Superclasses

[IDLffXMLDOMNode](#)

[IDLffXMLDOMCharacterData](#)

Creation

See [IDLffXMLDOMComment::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from its superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMComment::Cleanup](#)
- [IDLffXMLDOMComment::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMComment Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMComment::Cleanup

The IDLffXMLDOMComment::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMComment::Init

The IDLffXMLDOMComment::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMDocument

The IDLffXMLDOMDocument class represents the entire XML document as the root of the document tree and by providing the primary access to the document's data.

This class is also the class factory for DOM tree nodes such as [IDLffXMLDOMText](#) since all DOM tree nodes must belong to a specific document. To this end, this class includes methods that create DOM tree nodes of all the various types.

Note that nodes created with these factory methods belong to the document, but are not part of the DOM tree structure until they are inserted into the DOM tree. See the relevant methods in the [IDLffXMLDOMNode](#) class for details on inserting nodes into the DOM tree.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMDocument::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMDocument::Cleanup](#)
- [IDLffXMLDOMDocument::CreateAttribute](#)
- [IDLffXMLDOMDocument::CreateCDATASection](#)
- [IDLffXMLDOMDocument::CreateComment](#)
- [IDLffXMLDOMDocument::CreateElement](#)
- [IDLffXMLDOMDocument::CreateEntityReference](#)
- [IDLffXMLDOMDocument::CreateProcessingInstruction](#)
- [IDLffXMLDOMDocument::CreateTextNode](#)

- [IDLffXMLDOMDocument::CreateDocumentFragment](#)
- [IDLffXMLDOMDocument::GetDoctype](#)
- [IDLffXMLDOMDocument::GetDocumentElement](#)
- [IDLffXMLDOMDocument::GetElementsByTagName](#)
- [IDLffXMLDOMDocument::Init](#)
- [IDLffXMLDOMDocument::Load](#)
- [IDLffXMLDOMDocument::Save](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMDocument Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMDocument::Cleanup

The IDLffXMLDOMDocument::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffXMLDOMDocument::]Cleanup (*Only in subclass's Cleanup method*)

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateAttribute

The IDLffXMLDOMDocument::CreateAttribute function method creates and names an attribute node that is owned by the document. The new attribute can be set in an Element node with the [IDLffXMLDOMElement::SetAttributeNode](#) method.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]CreateAttribute(*Name*)

Return Value

Returns an object reference to the new [IDLffXMLDOMAttr](#) node.

Arguments

Name

A scalar string containing the attribute name.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateCDATASection

The IDLffXMLDOMDocument::CreateCDATASection function method creates a CDATASection node that is owned by the document and stores a string in the CDATASection node.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]CreateCDATASection(*String*)

Return Value

Returns an object reference to the new [IDLffXMLDOMCDATASection](#) node.

Arguments

String

A scalar string to put in the new CDATASection node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateComment

The IDLffXMLDOMDocument::CreateComment function method creates a comment node that is owned by the document and stores a string in the comment node.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]CreateComment(*String*)

Return Value

Returns an object reference to the new [IDLffXMLDOMComment](#) node.

Arguments

String

A scalar string to put in the new comment node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateDocumentFragment

The IDLffXMLDOMDocument::CreateDocumentFragment function method creates a document-fragment node that is owned by the document.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]CreateDocumentFragment()

Return Value

Returns an object reference to the new [IDLffXMLDOMDocumentFragment](#) node.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateElement

The IDLffXMLDOMDocument::CreateElement function method creates an element node that is owned by the document and sets the element's name to a tag name.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]CreateElement(*TagName*)

Return Value

Returns an object reference to the new [IDLffXMLDOMElement](#) node.

Arguments

TagName

A scalar string that contains the name of the element.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateEntityReference

The IDLffXMLDOMDocument::CreateEntityReference function method creates and names an entity reference node that is owned by the document.

Syntax

Result = Obj->[IDLffXMLDOMDocument::]CreateEntityReference(*Name*)

Return Value

Returns an object reference to the new [IDLffXMLDOMEntityReference](#) node.

Arguments

Name

A scalar string that contains the entity reference name.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateProcessingInstruction

The IDLffXMLDOMDocument::CreateProcessingInstruction function method creates a processing-instruction node that is owned by the document and stores two strings in the processing-instruction node.

Syntax

```
Result = Obj->[IDLffXMLDOMDocument::]CreateProcessingInstruction( Target,  
    Data )
```

Return Value

Returns an object reference to the new [IDLffXMLDOMProcessingInstruction](#) node.

Arguments

Target

A scalar string to store in the processing-instruction target.

Data

A scalar string to store in the processing-instruction data.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::CreateTextNode

The IDLffXMLDOMDocument::CreateTextNode function method creates a text node that is owned by the document and stores a string in the text node.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]CreateTextNode(*String*)

Return Value

Returns an object reference to the new [IDLffXMLDOMText](#) node.

Arguments

String

A scalar string to put in the new text node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::GetDoctype

The IDLffXMLDOMDocument::GetDoctype function method creates an instance of [IDLffXMLDOMDocumentType](#) and returns its object reference.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]GetDoctype()

Return Value

Returns an object reference to the new instance of [IDLffXMLDOMDocumentType](#), or a null object reference if a document does not have a DTD.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::GetDocumentElement

The IDLffXMLDOMDocument::GetDocumentElement function method creates an instance of an [IDLffXMLDOMElement](#) and returns its object reference. The IDLffXMLDOMElement object refers to the node that is the root element of the document.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]GetDocumentElement()

Return Value

Returns an object reference to the new instance of [IDLffXMLDOMElement](#).

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::GetElementsByTagName

The `IDLffXMLDOMDocument::GetElementsByTagName` function method creates an [IDLffXMLDOMNodeList](#) object containing all element nodes in the document tree with the specified tag name and returns an object reference to the [IDLffXMLDOMNodeList](#) object. The nodes are placed in the node list in the order that they are encountered during a preorder traversal of the tree.

Syntax

Result = *Obj*->[IDLffXMLDOMDocument::]GetElementsByTagName(*TagName*)

Return Value

Returns an object reference to the [IDLffXMLDOMNodeList](#) object.

The returned node list is “live,” in that any changes made to the document tree after the node list was returned will be immediately reflected in the node list.

If no nodes matching the specified tag name are found, an [IDLffXMLDOMNodeList](#) object is still returned, but contains no nodes.

Arguments

TagName

A scalar string containing the tag name that is used to match the elements. The value “*” matches all tags only; it is not a wildcard expression, and you cannot use it to make partial matches.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocument::Init

The IDLffXMLDOMDocument::Init function method initializes the object. It can also load and parse an XML document, using the same keywords that [IDLffXMLDOMDocument::Load](#) does. (For more information on loading, see that method's description.)

Syntax

```
Obj = OBJ_NEW('IDLffXMLDOMDocument'
  [, /EXCLUDE_IGNOREABLE_WHITESPACE]
  [, /EXPAND_ENTITY_REFERENCES] [, FILENAME=string]
  [, MSG_ERROR=string] [, MSG_FATAL=string] [, MSG_WARNING=string]
  [, /QUIET] [, VALIDATION_MODE=value])
```

or

```
Result = Obj->[IDLffXMLDOMDocument::]Init(
  [, /EXCLUDE_IGNOREABLE_WHITESPACE]
  [, /EXPAND_ENTITY_REFERENCES] [, FILENAME=string]
  [, MSG_ERROR=string] [, MSG_FATAL=string] [, MSG_WARNING=string]
  [, /QUIET] [, VALIDATION_MODE=value] )
```

(Only in a subclass's Init method)

Return Value

Returns an object reference to the newly created object when this method is called indirectly, as part of the call to the OBJ_NEW function.

Returns 1 if initialization was successful or 0 otherwise, when this method is called directly within a subclass Init method.

Arguments

None

Keywords

EXCLUDE_IGNOREABLE_WHITESPACE

Set this keyword to prevent text nodes containing ignorable white space from being inserted into the DOM tree. The default action is to include these nodes in the DOM

tree. Note that in order for the parser to distinguish between ignorable and non-ignorable white space, validation must be turned on with the [VALIDATION_MODE](#) keyword and there must be a DTD or schema. This keyword has no effect if [VALIDATION_MODE](#) is 0 or if there is no DTD or schema for the XML document.

EXPAND_ENTITY_REFERENCES

Set this keyword to cause the parser to replace entity references with their fully expanded substitution text. If entity references are not expanded, the DOM parser creates entity-reference nodes in the DOM tree. The children of the entity-reference nodes are read-only and are elements containing the entity values. If entity references are expanded, the DOM parser creates no entity-reference nodes in the DOM tree and instead creates read-write nodes that correspond to the fully expanded substitution text.

FILENAME

Set this keyword to a scalar string containing the name of the XML file to load and parse.

MSG_ERROR

Set this keyword to a scalar string that contains the name of an IDL procedure that will be called when the parser generates an error message. The following procedure parameters are required:

- Filename – A scalar string containing the name of the file being parsed
- Line number – A scalar integer containing the line number in the file where the parsing error occurred
- Column number – A scalar integer containing the column number in the file where the parsing error occurred
- Message – A scalar string containing the parser error message

MSG_FATAL

Set this keyword to a scalar string that contains the name of an IDL procedure that will be called when the parser generates a fatal error message. The required procedure parameters are the same as for [MSG_ERROR](#).

MSG_WARNING

Set this keyword to a scalar string that contains the name of an IDL procedure that will be called when the parser generates a warning message. The required procedure parameters are the same as for [MSG_ERROR](#).

QUIET

Set this keyword to prevent IDL from printing out informational parser warnings and error messages. The default action is to report all parser warnings and errors with messages. If an error more severe than a warning is encountered, IDL issues an additional single message after parsing is complete and returns to the interpreter, regardless of this keyword's setting.

VALIDATION_MODE

Set this keyword to one of the following values:

- 0 (none): the parser does not report validation errors
- 1 (auto): the parser reports validation errors only if a grammar (DTD, schema) is specified
- 2 (always): the parser always reports validation errors

This keyword ignores all other values. The default value is 0 (none).

Version History

Introduced: 6.1

IDLffXMLDOMDocument::Load

The IDLffXMLDOMDocument::Load procedure method loads XML data from the source specified by the **FILENAME** keyword. The XML parser then parses the data, using any related DTD or schema specifications. The results of the parse are used to construct a DOM document tree that can then be accessed through this document object.

If the **FILENAME** keyword is not specified, the other keywords have no effect and the DOM document tree is unchanged. If it is specified, then any existing DOM document tree is destroyed, and the results of the parsing operation are used to create a new DOM document tree.

Note

This method invalidates all object references to node objects created with the previous document.

IDL issues messages for any parser warnings or errors unless the **QUIET** keyword is set. If a parsing error that is more severe than a warning is encountered during parsing, IDL issues a single additional message after parsing and returns to the interpreter, even if **QUIET** is set. The IDL CATCH facility can intercept this message. If this message is issued, the document tree is empty, containing no nodes.

Syntax

```
Obj->[IDLffXMLDOMDocument::]Load
    [, /EXCLUDE_IGNOREABLE_WHITESPACE]
    [, /EXPAND_ENTITY_REFERENCES] [, FILENAME=string]
    [, MSG_ERROR=string] [, MSG_FATAL=string] [, MSG_WARNING=string]
    [, /QUIET] [, VALIDATION_MODE=value]
```

Arguments

None

Keywords

EXCLUDE_IGNOREABLE_WHITESPACE

Set this keyword to prevent text nodes containing ignorable white space from being inserted into the DOM tree. The default action is to include these nodes in the DOM tree. Note that in order for the parser to distinguish between ignorable and non-

ignorable white space, validation must be turned on with the [VALIDATION_MODE](#) keyword and there must be a DTD or schema. This keyword has no effect if [VALIDATION_MODE](#) is 0 or if there is no DTD or schema for the XML document.

EXPAND_ENTITY_REFERENCES

Set this keyword to cause the parser to replace entity references with their fully expanded substitution text. If entity references are not expanded, the DOM parser creates entity-reference nodes in the DOM tree. The children of the entity-reference nodes are read-only and are elements containing the entity values. If entity references are expanded, the DOM parser creates no entity-reference nodes in the DOM tree and instead creates read-write nodes that correspond to the fully expanded substitution text.

FILENAME

Set this keyword to a scalar string containing the name of the XML file to load and parse. If this keyword is not specified, the other keywords have no effect and the DOM document tree is unchanged. If it is specified, then any existing DOM document tree is destroyed, and the results of the parsing operation are used to create a new DOM document tree.

MSG_ERROR

Set this keyword to a scalar string that contains the name of an IDL procedure that will be called when the parser generates an error message. The following procedure parameters are required:

- Filename – A scalar string containing the name of the file being parsed
- Line number – A scalar integer containing the line number in the file where the parsing error occurred
- Column number – A scalar integer containing the column number in the file where the parsing error occurred
- Message – A scalar string containing the parser error message

MSG_FATAL

Set this keyword to a scalar string that contains the name of an IDL procedure that will be called when the parser generates a fatal error message. The required procedure parameters are the same as for [MSG_ERROR](#).

MSG_WARNING

Set this keyword to a scalar string that contains the name of an IDL procedure that will be called when the parser generates a warning message. The required procedure parameters are the same as for [MSG_ERROR](#).

QUIET

Set this keyword to prevent IDL from printing out informational parser warnings and error messages. The default action is to report all parser warnings and errors with messages. If an error more severe than a warning is encountered, IDL issues an additional single message after parsing is complete and returns to the interpreter, regardless of this keyword's setting.

VALIDATION_MODE

Set this keyword to one of the following scalar values:

- 0 (none): the parser does not report validation errors
- 1 (auto): the parser reports validation errors only if a grammar (DTD, schema) is specified
- 2 (always): the parser always reports validation errors

This method ignores all other values. The default value is 0 (none).

Version History

Introduced: 6.1

IDLffXMLDOMDocument::Save

The IDLffXMLDOMDocument::Save procedure method serializes the current DOM document belonging to this document object and writes it to the output source specified by the **FILENAME** keyword.

Syntax

```
Obj->[IDLffXMLDOMDocument::]Save [, /EXPAND_ENTITY_REFERENCES]  
    [, FILENAME=string] [, /PRETTY_PRINT]
```

Arguments

None

Keywords

EXPAND_ENTITY_REFERENCES

Set this keyword to cause the parser to replace entity references with their fully expanded substitution text as the DOM tree is written to the output file. The DOM tree itself is not changed. The default behavior is to write the entity references to the output file.

FILENAME

Set this keyword to the name of the XML file in which to store the XML data. If FILENAME is not set, this method will return without storing any data on the file system.

PRETTY_PRINT

Set this keyword to format the output by adding white space to produce a pretty-printed, indented, reader-friendly form.

Version History

Introduced: 6.1

IDLffXMLDOMDocumentFragment

The IDLffXMLDOMDocumentFragment class references a document-fragment node.

It is very common to want to be able to extract a portion of a document's tree or to create a new fragment of a document. Imagine implementing a user command such as “cut” or rearranging a document by moving fragments around. It is desirable to have an object that can hold such fragments, and it is quite natural to use a node for this purpose. While it is true that a document object could fill this role, it can potentially be a heavyweight object, depending on the underlying implementation. What you need for this purpose, instead, is a very lightweight object, and the document-fragment node is such an object.

Furthermore, various operations such as inserting nodes as children of another node may take document-fragment objects as arguments; this results in all the child nodes of the document-fragment node being moved to the child list of this node.

The children of a document-fragment node are zero or more nodes representing the roots of any sub-trees defining the structure of the document. Document-fragment nodes do not need to be well-formed XML documents (although they do need to follow the rules imposed upon well-formed XML parsed entities, which can have multiple root nodes). For example, a document-fragment node might have only one child, and that child node could be a text node. Such a structure model represents neither an HTML document nor a well-formed XML document.

When a document-fragment node is inserted into a document tree (or, indeed, any other node that may take children), the children of the document-fragment node and not the document-fragment node itself are inserted into the node. This insertion process makes the document-fragment node very useful when you want to create nodes that are siblings. Because the document-fragment node acts as the parent of these nodes, you can use the standard methods from the node object, such as `InsertBefore()` and `AppendChild()`.

There are no methods specific to this class except `Cleanup` and `Init`.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMDocumentFragment::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMDocumentFragment::Cleanup](#)
- [IDLffXMLDOMDocumentFragment::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMDocumentFragment Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMDocumentFragment::Cleanup

The IDLffXMLDOMDocumentFragment::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocumentFragment::Init

The IDLffXMLDOMDocumentFragment::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMDocument::CreateDocumentFragment](#) are used to create objects of this class.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMDocumentType

The `IDLffXMLDOMDocumentType` class references a document-type node. Each document has a `doctype` attribute whose value is stored in a document-type object or is null.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMDocumentType::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMDocumentType::Cleanup](#)
- [IDLffXMLDOMDocumentType::GetEntities](#)
- [IDLffXMLDOMDocumentType::GetName](#)
- [IDLffXMLDOMDocumentType::GetNotations](#)
- [IDLffXMLDOMDocumentType::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMDocumentType Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMDocumentType::Cleanup

The IDLffXMLDOMDocumentType::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocumentType::GetEntities

The IDLffXMLDOMDocumentType::GetEntities function method returns the general entities, both external and internal, declared in the DTD.

Syntax

Result = *Obj*->[IDLffXMLDOMDocumentType:]GetEntities()

Return Value

Returns an object reference to a named node map ([IDLffXMLDOMNamedNodeMap](#)) containing the general entities declared in the DTD. If there are no entities, this method returns a null object reference.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocumentType::GetName

The IDLffXMLDOMDocumentType::GetName function method returns the name of the DTD.

Syntax

Result = *Obj*->[IDLffXMLDOMDocumentType::]GetName()

Return Value

Returns a scalar string containing the name of the DTD (i.e., the name immediately following the DOCTYPE keyword).

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocumentType::GetNotations

The IDLffXMLDOMDocumentType::GetNotations function method returns the notations declared in the DTD.

Syntax

Result = *Obj*->[IDLffXMLDOMDocumentType:]GetNotations()

Return Value

Returns an object reference to a named node map ([IDLffXMLDOMNamedNodeMap](#)) containing the notations declared in the DTD. If there are no notations, this method returns a null object reference.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMDocumentType::Init

The IDLffXMLDOMDocumentType::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMElement

The IDLffXMLDOMElement class references an element node, which is one of the most common node types in an XML document.

Elements can have attributes associated with them, which can be obtained with the [IDLffXMLDOMNode::GetAttributes](#) method. The IDLffXMLDOMElement class also has methods for getting an attribute object or an attribute value by its name.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMElement::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMElement::Cleanup](#)
- [IDLffXMLDOMElement::GetAttribute](#)
- [IDLffXMLDOMElement::GetAttributeNode](#)
- [IDLffXMLDOMElement::GetElementsByTagName](#)
- [IDLffXMLDOMElement::GetTagName](#)
- [IDLffXMLDOMElement::Init](#)
- [IDLffXMLDOMElement::RemoveAttribute](#)
- [IDLffXMLDOMElement::RemoveAttributeNode](#)
- [IDLffXMLDOMElement::SetAttribute](#)
- [IDLffXMLDOMElement::SetAttributeNode](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMEElement Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMELEMENT::Cleanup

The IDLffXMLDOMELEMENT::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMElement::GetAttribute

The IDLffXMLDOMElement::GetAttribute function method returns the value of the attribute specified by *Name*.

Syntax

Result = *Obj*->[IDLffXMLDOMElement::]GetAttribute(*Name*)

Return Value

Returns a scalar string containing the value of the attribute specified by *Name*.

Arguments

Name

A scalar string used to specify the name of the attribute for which to get the value.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMELEMENT::GetAttributeNode

The IDLffXMLDOMELEMENT::GetAttributeNode function method creates an [IDLffXMLDOMAttr](#) object and returns its object reference.

Syntax

Result = Obj->[IDLffXMLDOMELEMENT::]GetAttributeNode(Name)

Return Value

Returns an object reference to the [IDLffXMLDOMAttr](#) object. The object refers to the attribute node specified by name. If this node does not exist, this function returns a null object reference.

Arguments

Name

A scalar string used to specify the attribute node to get.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMElement::GetElementsByTagName

The `IDLffXMLDOMElement::GetElementsByTagName` function method creates an [IDLffXMLDOMNodeList](#) object containing all descendant element nodes in the document tree with the specified tag name and returns an object reference to the [IDLffXMLDOMNodeList](#) object. The nodes are placed in the node list in the order that they are encountered during a preorder traversal of the tree.

Syntax

Result = *Obj*->[IDLffXMLDOMElement::]GetElementsByTagName(*TagName*)

Return Value

Returns an object reference to the [IDLffXMLDOMNodeList](#) object.

The returned node list is “live,” in that any changes made to the document tree after the node list was returned will be immediately reflected in the node list.

If no nodes matching the specified tag name are found, an [IDLffXMLDOMNodeList](#) is still returned, but contains no nodes.

Arguments

TagName

A scalar string containing the tag name that is used to match the elements. The value “*” matches all tags only; it is not a wildcard expression, and you cannot use it to make partial matches.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMELEMENT::GetTagName

The IDLffXMLDOMELEMENT::GetTagName function method returns the name of the element.

Syntax

Result = *Obj*->[IDLffXMLDOMELEMENT::]GetTagName()

Return Value

Returns a scalar string containing the name of the element.

Argument

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMELEMENT::Init

The IDLffXMLDOMELEMENT::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMELEMENT::RemoveAttribute

The IDLffXMLDOMELEMENT::RemoveAttribute procedure method removes a specified attribute from the element node.

Syntax

Result = *Obj*->[IDLffXMLDOMELEMENT::]RemoveAttribute(*Name*)

Arguments

Name

A scalar string that contains the name of the attribute to remove from the element.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMElement::RemoveAttributeNode

The IDLffXMLDOMElement::RemoveAttributeNode function method removes a specified attribute node from the element node.

Syntax

Result = *Obj*->[IDLffXMLDOMElement::]RemoveAttributeNode(*OldAttr*)

Return Value

Returns an object reference to an [IDLffXMLDOMAttr](#) node referring to the attribute node that was removed.

Arguments

OldAttr

An object reference to an [IDLffXMLDOMAttr](#) object that refers to an attribute node to be removed from the element node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMELEMENT::SetAttribute

The IDLffXMLDOMELEMENT::SetAttribute procedure method adds a new attribute to the element node.

Syntax

Obj->[IDLffXMLDOMELEMENT::]SetAttribute, *Name*, *Value*

Arguments

Name

A scalar string that contains the name of the attribute.

Value

A scalar string that contains the value of the attribute.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMElement::SetAttributeNode

The IDLffXMLDOMElement::SetAttributeNode function method adds an attribute node to the element node.

Syntax

Result = *Obj*->[IDLffXMLDOMElement::]SetAttributeNode(*NewAttr*)

Return Value

Returns an object reference to an [IDLffXMLDOMAttr](#) node, if an old attribute is replaced by the new. This method returns a null object reference if no attribute was replaced by this method.

Arguments

NewAttr

An object reference to an [IDLffXMLDOMAttr](#) object that refers to an attribute node that is to be set in the element node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMEntity

The IDLffXMLDOMEntity class references an entity, either parsed or unparsed, in an XML document. Entities are placeholders in XML and are inserted into documents by entity references. Entity nodes have no parents. Entity nodes and their descendants are read-only.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMEntity::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMEntity::Cleanup](#)
- [IDLffXMLDOMEntity::GetNotationName](#)
- [IDLffXMLDOMEntity::GetPublicId](#)
- [IDLffXMLDOMEntity::GetSystemId](#)
- [IDLffXMLDOMEntity::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMEntity Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMEntity::Cleanup

The IDLffXMLDOMEntity::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMEntity::GetNotationName

The IDLffXMLDOMEntity::GetNotationName function method returns the name of the notation for the entity.

Syntax

Result = *Obj*->[IDLffXMLDOMEntity::]GetNotationName()

Return Value

Returns a scalar string containing the name of the notation for the unparsed entity, or an empty string if the entity is parsed.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMEntity::GetPublicId

The IDLffXMLDOMEntity::GetPublicId function method returns the public ID of the entity, if it is specified.

Syntax

Result = *Obj*->[IDLffXMLDOMEntity::]GetPublicId()

Return Value

Returns a scalar string containing the public ID of the entity, or an empty string if the public ID is not specified.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMEntity::GetSystemId

The IDLffXMLDOMEntity::GetSystemId function method returns the system ID of the entity, if it is specified.

Syntax

Result = *Obj*->[IDLffXMLDOMEntity::]GetSystemId()

Return Value

Returns a scalar string containing the system ID of the entity, or an empty string if the system ID is not specified.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMEntity::Init

The IDLffXMLDOMEntity::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMDocumentType::GetEntities](#) coupled with [IDLffXMLDOMNamedNodeMap::GetNamedItem](#) are used to create objects of this class.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMEntityReference

The IDLffXMLDOMEntityReference class references entity reference nodes. Entity references in the source XML file are meant to be substituted by the entity contents upon parsing. Any entity reference nodes left in the document tree may exist because the entity was not found, or the parser was instructed to not expand entity references.

Entity references are read-only.

There are no methods specific to this class except Cleanup and Init.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMEntityReference::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMEntityReference::Cleanup](#)
- [IDLffXMLDOMEntityReference::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLfXMLDOMEntityReference Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMEntityReference::Cleanup

The IDLffXMLDOMEntityReference::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMEntityReference::Init

The IDLffXMLDOMEntityReference::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMNamedNodeMap

The IDLffXMLDOMNamedNodeMap class is a container for nodes (see [IDLffXMLDOMNode](#) and its subclasses) and uses the node's name to access the nodes in the container.

For example, consider this XML:

```
<show title="My Title" date="Nov 12, 1999" author="Me">
```

The entire line is a DOM element node. A `GetAttributes` call returns a `NamedNodeMap` that contains three attribute node objects. The attribute node object containing the date, for example, has a node name of 'date' and a node value of 'Nov 12, 1999'.

The benefit of the named node map is that the nodes are accessed by their node names. Thus, to access the date node, you would use the string 'date' and would not have to worry about its position in the list.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMNamedNodeMap::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMNamedNodeMap::Cleanup](#)
- [IDLffXMLDOMNamedNodeMap::GetLength](#)
- [IDLffXMLDOMNamedNodeMap::GetNamedItem](#)
- [IDLffXMLDOMNamedNodeMap::Init](#)
- [IDLffXMLDOMNamedNodeMap::Item](#)

- [IDLffXMLDOMNamedNodeMap::RemoveNamedItem](#)
- [IDLffXMLDOMNamedNodeMap::SetNamedItem](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMNamedNodeMap Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMNamedNodeMap::Cleanup

The IDLffXMLDOMNamedNodeMap::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNamedNodeMap::GetLength

The IDLffXMLDOMNamedNodeMap::GetLength function method returns the number of nodes in the named node map.

Syntax

Result = *Obj*->[IDLffXMLDOMNamedNodeMap::]GetLength()

Return Value

Returns an integer containing the number of nodes in the named node map.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNamedNodeMap::GetNamedItem

The IDLffXMLDOMNamedNodeMap::GetNamedItem function method returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the node from the named node map that corresponds to the supplied name.

Syntax

Result = *Obj*->[IDLffXMLDOMNamedNodeMap::]GetNamedItem(*Name*)

Return Value

Returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the node from the named node map that corresponds to the supplied name. If the node is not found, this function returns a null object reference.

Arguments

Name

A scalar string that contains the name used to locate the node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNamedNodeMap::Init

The IDLffXMLDOMNamedNodeMap::Init function method initializes the object, but this class should not be instantiated directly. Instead, the [IDLffXMLDOMNode::GetAttributes](#) method is used to create objects of this class.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMNamedNodeMap::Item

The IDLffXMLDOMNamedNodeMap::Item function method returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the node from the named node map that corresponds to an index.

Syntax

Result = *Obj*->[IDLffXMLDOMNamedNodeMap::]Item(*Index*)

Return Value

Returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the node from the named node map that corresponds to the supplied index. If the node at the specified index does not exist, a null object reference is returned.

Arguments

Index

An integer that contains the index into the named node map.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNamedNodeMap::RemoveNamedItem

The IDLffXMLDOMNamedNodeMap::RemoveNamedItem function method removes the node from the named node map that corresponds to a name.

Syntax

Result = *Obj*->[IDLffXMLDOMNamedNodeMap::]RemoveNamedItem(*Name*)

Return Value

Returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the removed node if it exists. If the node does not exist, this function returns a null object reference.

Arguments

Name

A scalar string that contains the name used to locate the node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNamedNodeMap::SetNamedItem

The `IDLffXMLDOMNamedNodeMap::SetNamedItem` function method adds a node to the named node map. After adding the node, the node can be accessed using the node's name attribute. If a node with the same name is already present in the map, then the node in the map is replaced by this new node.

Some node types have fixed name tags due to the nature of the node type. For example, an [IDLffXMLDOMText](#) node's name tag always has the value “#text” because it always contains text in its value tag. For a list of other nodes with fixed name tags see “[IDLffXMLDOMNode::GetNodeName Return Values](#)” on page 2997. (The fixed names are the names in quotation marks.)

Note

You cannot place nodes with fixed name tags into a named node map because it uses the nodes' name tags to search for a requested node.

Syntax

Result = *Obj*->[IDLffXMLDOMNamedNodeMap::]SetNamedItem(*Node*)

Return Value

Returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that references the node that was replaced, if the method replaces a node in the map. Otherwise, the method returns a null object reference.

Arguments

Node

An object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the node to add to the map.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode

The IDLffXMLDOMNode class is an abstract class used as a superclass for other IDLffXMLDOM node classes.

Because this class is an abstract class, there will never be an instance of this class. Rather, this class's functionality described here is accessed through an instance of one of its subclasses.

An object instance of this class is a reference to a corresponding node in the DOM document tree. Use the appropriate methods for this class, defined below, to access and manipulate the DOM data.

This class keeps track of any objects it creates (e.g., via the [IDLffXMLDOMNode::GetFirstChild](#) method) and destroys these objects when it is destroyed.

Superclasses

None

Creation

See [IDLffXMLDOMNode::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMNode::AppendChild](#)
- [IDLffXMLDOMNode::Cleanup](#)
- [IDLffXMLDOMNode::CloneNode](#)
- [IDLffXMLDOMNode::GetAttributes](#)
- [IDLffXMLDOMNode::GetChildNodes](#)
- [IDLffXMLDOMNode::GetFirstChild](#)

- [IDLffXMLDOMNode::GetLastChild](#)
- [IDLffXMLDOMNode::GetNextSibling](#)
- [IDLffXMLDOMNode::GetNodeName](#)
- [IDLffXMLDOMNode::GetNodeType](#)
- [IDLffXMLDOMNode::GetNodeValue](#)
- [IDLffXMLDOMNode::GetOwnerDocument](#)
- [IDLffXMLDOMNode::GetParentNode](#)
- [IDLffXMLDOMNode::GetPreviousSibling](#)
- [IDLffXMLDOMNode::HasChildNodes](#)
- [IDLffXMLDOMNode::Init](#)
- [IDLffXMLDOMNode::InsertBefore](#)
- [IDLffXMLDOMNode::RemoveChild](#)
- [IDLffXMLDOMNode::ReplaceChild](#)
- [IDLffXMLDOMNode::SetNodeValue](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMNode Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMNode::AppendChild

The IDLffXMLDOMNode::AppendChild function method adds a new node to the end of the calling node's children list.

If the new child node is already in the tree, it is first removed.

The ownership of the IDL object to which the *NewChild* object reference refers is changed so that it is no longer owned by its previous owner and becomes owned by the calling object. Therefore, destroying the calling object destroys the new child as well.

Syntax

```
Result = Obj->[IDLffXMLDOMNode::]AppendChild( NewChild )
```

Return Value

Returns an object reference to the new child node.

Arguments

NewChild

An object reference to an instance of an IDLffXMLDOMNode subclass that refers to the node that is being added.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::Cleanup

The IDLffXMLDOMNode::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::CloneNode

The IDLffXMLDOMNode::CloneNode function method creates a copy of the calling node (i.e., an instance of an IDLffXMLDOMNode subclass) and returns the object reference to the cloned node. The class of the cloned node is the same as the class of the calling node. The cloned node has no parent.

Cloning an element node copies all attributes and their values and does not copy child nodes unless the **DEEP** keyword is set. Cloning any other type of node simply copies the node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]CloneNode([, /**DEEP**])

Return Value

Returns an object reference to the cloned node.

Arguments

None

Keywords

DEEP

Set this keyword to cause all of a node's children to be cloned along with the calling node. By default, only the calling node is cloned.

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetAttributes

The IDLffXMLDOMNode::GetAttributes function method creates an instance of an [IDLffXMLDOMNamedNodeMap](#) and returns an object reference to that object. The named node map can then be used to access the attributes of an element object.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetAttributes()

Return Value

Returns an object reference to the new [IDLffXMLDOMNamedNodeMap](#) object. If the calling object is not an [IDLffXMLDOMElement](#), a null object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetChildNodes

The IDLffXMLDOMNode::GetChildNodes function method creates an instance of an [IDLffXMLDOMNodeList](#) object, which can then be used to access the child nodes. If there are no children, the node list object contains no nodes.

The contents of the node list object are updated as changes are made to the tree, such as children being added or removed from the calling node's children list.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetChildNodes()

Return Value

Returns an object reference to the new [IDLffXMLDOMNodeList](#) object.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetFirstChild

The IDLffXMLDOMNode::GetFirstChild function method creates an instance of one of the IDLffXMLDOMNode subclasses and returns an object reference to that object, which refers to the first child of the calling node in the DOM tree. The type of the object corresponds to the type of the first child node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetFirstChild()

Return Value

Returns an object reference to the new IDLffXMLDOMNode subclass object if there is a child in the DOM tree. If there are no children, a null object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetLastChild

The IDLffXMLDOMNode::GetLastChild function method creates an instance of one of the IDLffXMLDOMNode subclasses and returns an object reference to that object, which refers to the last child of the calling node in the DOM tree. The type of the object corresponds to the type of the last child node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetLastChild()

Return Value

Returns an object reference to the new IDLffXMLDOMNode subclass object if there is a child in the DOM tree. If there are no children, a null object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetNextSibling

The IDLffXMLDOMNode::GetNextSibling function method creates an instance of one of the IDLffXMLDOMNode subclasses and returns an object reference to that object, which refers to the next sibling of the calling node in the DOM tree. The type of the object corresponds to the type of the next sibling node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetNextSibling()

Return Value

Returns an object reference to the new IDLffXMLDOMNode subclass object if there is a next sibling in the DOM tree. If there is no sibling, a null object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetNodeName

The IDLffXMLDOMNode::GetNodeName function method returns the name of the node, depending on the calling node's subclass type.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetNodeName()

Return Value

Returns a scalar string whose value depends on the subclass type:

Subclass	Return Value
IDLffXMLDOMAttr	attribute name
IDLffXMLDOMCDATASection	"#cdata-section"
IDLffXMLDOMComment	"#comment"
IDLffXMLDOMDocument	"#document"
IDLffXMLDOMDocumentFragment	"#document-fragment"
IDLffXMLDOMDocumentType	document type name
IDLffXMLDOMEElement	tag name
IDLffXMLDOMEntity	entity name
IDLffXMLDOMEntityReference	name of entity reference
IDLffXMLDOMNotation	notation name
IDLffXMLDOMProcessingInstruction	target
IDLffXMLDOMText	"#text"

Table 6-23: IDLffXMLDOMNode::GetNodeName Return Values

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetNodeType

The IDLffXMLDOMNode::GetNodeType function method returns the subclass type of the calling node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetNodeType()

Return Value

Returns an integer whose value depends on the subclass type:

Subclass	Return Value
IDLffXMLDOMAttr	2
IDLffXMLDOMCDATASection	4
IDLffXMLDOMComment	8
IDLffXMLDOMDocument	9
IDLffXMLDOMDocumentFragment	11
IDLffXMLDOMDocumentType	10
IDLffXMLDOMElement	1
IDLffXMLDOMEntity	6
IDLffXMLDOMEntityReference	5
IDLffXMLDOMNotation	12
IDLffXMLDOMProcessingInstruction	7
IDLffXMLDOMText	3

Table 10: IDLffXMLDOMNode::GetNodeType Return Values

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetNodeValue

The IDLffXMLDOMNode::GetNodeValue function method returns the value of the calling node, depending on its subclass type.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetNodeValue()

Return Value

Returns a scalar string whose value depends on the subclass type:

Subclass	Return Value
IDLffXMLDOMAttr	attribute value
IDLffXMLDOMCDATASection	content of the CDATA section
IDLffXMLDOMComment	content of the comment
IDLffXMLDOMDocument	empty string
IDLffXMLDOMDocumentFragment	empty string
IDLffXMLDOMDocumentType	empty string
IDLffXMLDOMElement	empty string
IDLffXMLDOMEntity	empty string
IDLffXMLDOMEntityReference	empty string
IDLffXMLDOMNotation	empty string
IDLffXMLDOMProcessingInstruction	entire content excluding the target
IDLffXMLDOMText	content of the text node

Table 11: IDLffXMLDOMNode::GetNodeValue Return Values

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetOwnerDocument

The IDLffXMLDOMNode::GetOwnerDocument function method returns an object reference to the [IDLffXMLDOMDocument](#) used to create the node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode]::GetOwnerDocument()

Return Value

Returns an object reference to the [IDLffXMLDOMDocument](#) used to create the node. If the calling node is an [IDLffXMLDOMDocument](#), the method returns a null object reference.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetParentNode

The IDLffXMLDOMNode::GetParentNode function method creates an instance of one of the IDLffXMLDOMNode subclasses and returns an object reference to that object, which refers to the parent node in the DOM tree. The type of the created object corresponds to the type of the parent node.

All node classes except [IDLffXMLDOMAttr](#), [IDLffXMLDOMDocument](#), [IDLffXMLDOMDocumentFragment](#), and [IDLffXMLDOMEntity](#) may have parents.

A node that has just been created and not yet added to the tree does not have a parent.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetParentNode()

Return Value

Returns an object reference to the new object, which refers to the calling node's parent in the DOM tree. The type of the new object corresponds to the type of the parent node. If there is no parent, a null object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::GetPreviousSibling

The IDLffXMLDOMNode::GetPreviousSibling function method creates an instance of one of the IDLffXMLDOMNode subclasses and returns an object reference to that object, which refers to the previous sibling of the calling node in the DOM tree. The type of the object corresponds to the type of the previous sibling node.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]GetPreviousSibling()

Return Value

Returns an object reference to the new IDLffXMLDOMNode subclass object if there is a previous sibling in the DOM tree. If there is no sibling, a null object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::HasChildNodes

The IDLffXMLDOMNode::HasChildNodes returns a value indicating whether the calling node has children.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]HasChildNodes()

Return Value

Returns 1 if the calling node has children, or 0 otherwise.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::Init

The IDLffXMLDOMNode::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMNode::InsertBefore

The IDLffXMLDOMNode::InsertBefore function method inserts a new node into the calling node's children list at the location before a reference node.

If the reference node is not specified or is a null object reference, the new node is inserted at the end of the list of children.

If the new node is an [IDLffXMLDOMDocumentFragment](#), all of its children are inserted in the same order before the reference node.

If the new node is already in the tree, it is first removed.

The ownership of the IDL object to which the *NewChild* object reference refers is changed so that it is no longer owned by its previous owner and becomes owned by the calling object. Therefore, destroying the calling object destroys the new child as well.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]InsertBefore(*NewChild* [, *RefChild*])

Return Value

Returns an object reference to the new child node.

Arguments

NewChild

An object reference to an instance of an IDLffXMLDOMNode subclass that refers to the node to insert.

RefChild

An object reference to an instance of an IDLffXMLDOMNode subclass that refers to the reference node. The new child node is inserted before the reference node. The reference node must be a child of the calling node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::RemoveChild

The IDLffXMLDOMNode::RemoveChild function method removes an existing child node from the calling node's child list.

The ownership of the IDL object to which the *OldChild* object reference refers is transferred from this calling object to the document object. The old child node has no parent after it is replaced. The document object becomes the new owner of the old child node, which is destroyed when the document object is destroyed.

Syntax

Result = *Obj*->[IDLffXMLDOMNode::]RemoveChild(*OldChild*)

Return Value

Returns an object reference to the old child node.

Arguments

OldChild

An object reference to an instance of an IDLffXMLDOMNode subclass that refers to the node that is being removed. The old child node must be a child of the calling node.

Keywords

None

Version History

Introduced: 6.1

IDLfXMLDOMNode::ReplaceChild

The IDLfXMLDOMNode::ReplaceChild function method replaces an existing child node with a new child node into the calling node's child list.

If the new child node is an [IDLfXMLDOMDocumentFragment](#), the old child node is replaced by all the [IDLfXMLDOMDocumentFragment](#) children, which are inserted in the same order.

If the new child node is already in the tree, it is first removed.

The ownership of the IDL object to which the *NewChild* object reference refers is changed so that it is no longer owned by its previous owner and becomes owned by the calling object. Therefore, destroying the calling object destroys the new child as well.

The ownership of the IDL object to which the *OldChild* object reference refers is transferred from this calling object to the document object. The old child node has no parent after it is replaced. The document object becomes the new owner of the old child node, which is destroyed when the document object is destroyed.

Syntax

Result = *Obj*->[IDLfXMLDOMNode::]ReplaceChild(*NewChild*, *OldChild*)

Return Value

Returns an object reference to the old child node.

Arguments

NewChild

An object reference to an instance of an IDLfXMLDOMNode subclass that refers to the node to put into the tree, replacing the old child node.

OldChild

An object reference to an instance of an IDLfXMLDOMNode subclass that refers to the node that is being replaced. The old child node must be a child of the calling node.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNode::SetNodeValue

The IDLffXMLDOMNode::SetNodeValue procedure method sets the node's value to the contents of a string.

Syntax

Obj->[IDLffXMLDOMNode::]SetNodeValue(*NodeValue*)

Arguments

NodeValue

A scalar string value used to set the node's value.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNodeList

The IDLffXMLDOMNodeList class is a container for nodes (see [IDLffXMLDOMNode](#) and its subclasses).

Superclasses

None

Creation

See [IDLffXMLDOMNodeList::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMNodeList::Cleanup](#)
- [IDLffXMLDOMNodeList::GetLength](#)
- [IDLffXMLDOMNodeList::Init](#)
- [IDLffXMLDOMNodeList::Item](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMNodeList Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMNodeList::Cleanup

The IDLffXMLDOMNodeList::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNodeList::GetLength

The IDLffXMLDOMNodeList::GetLength returns the number of nodes in the node list.

Syntax

Result = *Obj*->[IDLffXMLDOMNodeList::]GetLength()

Return Value

Returns an integer containing the number of nodes in the node list.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNodeList::Init

The `IDLffXMLDOMNodeList::Init` function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetChildNodes](#) and [IDLffXMLDOMElement::GetElementsByTagName](#) are used to create objects of this class.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMNodeList::Item

The IDLffXMLDOMNodeList::Item returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to an indexed node from the node list.

Syntax

Result = *Obj*->[IDLffXMLDOMNodeList::]Item(*Index*)

Return Value

Returns an object reference to an instance of a subclass of [IDLffXMLDOMNode](#) that refers to the node from the node list that corresponds to the supplied index. If the node at the specified index does not exist, a null object reference is returned.

Arguments

Index

An integer that contains the index into the node list.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNotation

The IDLffXMLDOMNotation class represents a notation in the DTD. Notations are obtained from the [IDLffXMLDOMDocumentType](#) class.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMNotation::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMNotation::Cleanup](#)
- [IDLffXMLDOMNotation::GetPublicID](#)
- [IDLffXMLDOMNotation::GetSystemID](#)
- [IDLffXMLDOMNotation::Init](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMNotation Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMNotation::Cleanup

The IDLffXMLDOMNotation::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNotation::GetPublicID

The IDLffXMLDOMNotation::GetPublicID returns the public ID of the notation.

Syntax

Result = *Obj*->[IDLffXMLDOMNotation::]GetPublicID()

Return Value

Returns a scalar string containing the public ID of the notation.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNotation::GetSystemID

The IDLffXMLDOMNotation::GetSystemID returns the system ID of the notation.

Syntax

Result = *Obj*->[IDLffXMLDOMNotation::]GetSystemID()

Return Value

Returns a scalar string containing the system ID of the notation.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMNotation::Init

The IDLffXMLDOMNotation::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMProcessingInstruction

The IDLffXMLDOMProcessingInstruction class represents a processing instruction node in an XML document.

Superclasses

[IDLffXMLDOMNode](#)

Creation

See [IDLffXMLDOMProcessingInstruction::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMProcessingInstruction::Cleanup](#)
- [IDLffXMLDOMProcessingInstruction::GetData](#)
- [IDLffXMLDOMProcessingInstruction::GetTarget](#)
- [IDLffXMLDOMProcessingInstruction::Init](#)
- [IDLffXMLDOMProcessingInstruction::SetData](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 6.1

IDLffXMLDOMProcessingInstruction Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMProcessingInstruction::Cleanup

The IDLffXMLDOMProcessingInstruction::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMProcessingInstruction::GetData

The IDLffXMLDOMProcessingInstruction::GetData returns the content of the processing instruction.

Syntax

Result = *Obj*->[IDLffXMLDOMProcessingInstruction::]GetData()

Return Value

Returns a scalar string containing the content of the processing instruction.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMProcessingInstruction::GetTarget

The IDLffXMLDOMProcessingInstruction::GetTarget returns the target of the processing instruction.

Syntax

Result = *Obj*->[IDLffXMLDOMProcessingInstruction::]GetTarget()

Return Value

Returns a scalar string containing the target of the processing instruction.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMProcessingInstruction::Init

The IDLffXMLDOMProcessingInstruction::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMProcessingInstruction::SetData

The IDLffXMLDOMProcessingInstruction::SetData procedure method sets the content of the processing instruction.

Syntax

Obj->[IDLffXMLDOMProcessingInstruction:]SetData, *Content*

Arguments

Content

A scalar string containing the data to set in the processing instruction.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMText

The IDLffXMLDOMText class references a text node in an XML document.

Superclasses

[IDLffXMLDOMNode](#)

[IDLffXMLDOMCharacterData](#)

Creation

See [IDLffXMLDOMText::Init](#).

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLffXMLDOMText::Cleanup](#)
- [IDLffXMLDOMText::Init](#)
- [IDLffXMLDOMText::IsIgnorableWhitespace](#)
- [IDLffXMLDOMText::SplitText](#)

In addition, this class inherits the methods of any superclasses.

Version History

Introduced: 6.1

IDLffXMLDOMText Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLffXMLDOMText::Cleanup

The IDLffXMLDOMText::Cleanup procedure method destroys both the object (that is, the accessing object in the IDL tree) and any objects created by that object. It does not modify the actual DOM tree.

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

Syntax

OBJ_DESTROY, *Obj*

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMText::Init

The IDLffXMLDOMText::Init function method initializes the object, but this class should not be instantiated directly. Instead, methods such as [IDLffXMLDOMNode::GetFirstChild](#) are used to create objects of all classes except [IDLffXMLDOMDocument](#).

This class should not be subclassed by a user class because the classes that create objects, such as [IDLffXMLDOMDocument](#), will not be aware of any user-defined classes.

IDLffXMLDOMText::IsIgnorableWhitespace

The IDLffXMLDOMText::IsIgnorableWhitespace function method indicates whether the text node contains ignorable whitespace.

In order for the parser to distinguish between ignorable and non-ignorable white space, validation must be performed during parsing with a DTD or schema.

Syntax

Result = *Obj*->[IDLffXMLDOMText::]IsIgnorableWhitespace()

Return Value

Returns 1 if the text node contains ignorable whitespace, or 0 if the text node contains whitespace that is not ignorable.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLffXMLDOMText::SplitText

The IDLffXMLDOMText::SplitText function method breaks a text node into two nodes, splitting the text at the specified offset. Both nodes are kept in the same tree as siblings. The method both modifies its caller and creates an instance of a new IDLffXMLDOMText object, which contains the text following the offset point. When the offset is equal to the length of the string, the new node has no value data.

Syntax

Result = *Obj*->[IDLffXMLDOMText::]SplitText(*Offset*)

Return Value

Returns an object reference to the new IDLffXMLDOMText object.

Arguments

Offset

An integer containing the offset into the string at which to split it.

Keywords

None

Version History

Introduced: 6.1

IDLffXMLSAX

An IDLffXMLSAX object uses an XML SAX level 2 parser. The XML parser allows you to read an XML file and store arbitrary data from the file in IDL variables. The parser object's methods are *callbacks*. These methods are called automatically when the parser encounters different types of XML elements or attributes.

Note

To use the XML parser, you *must* write a subclass of this object class, overriding the object methods as necessary to process the data in a specific XML file or files. See [Chapter 25, “Using the XML Parser Object Class”](#) in the *Building IDL Applications* manual for further information and examples.

The IDLffXMLSAX object encapsulates the Xerces validating XML parser; see <http://xml.apache.org> for details.

Superclasses

This class has no superclass.

Subclasses

You *must* write a subclass of this object, overriding object methods as necessary to retrieve information from the XML file.

Creation

See [“IDLffXMLSAX::Init”](#) on page 3064

Properties

Objects of this class have the following properties. See [“IDLffXMLSAX Properties”](#) on page 3042 for details on individual properties.

- [FILENAME](#)
- [NAMESPACE_PREFIXES](#)
- [PARSER_LOCATION](#)
- [PARSER_PUBLICID](#)
- [PARSER_URI](#)

- [SCHEMA_CHECKING](#)
- [VALIDATION_MODE](#)

Methods

This class has the following methods:

- [IDLffXMLSAX::AttributeDecl](#)
- [IDLffXMLSAX::Characters](#)
- [IDLffXMLSAX::Cleanup](#)
- [IDLffXMLSAX::Comment](#)
- [IDLffXMLSAX::ElementDecl](#)
- [IDLffXMLSAX::EndCDATA](#)
- [IDLffXMLSAX::EndDocument](#)
- [IDLffXMLSAX::EndDTD](#)
- [IDLffXMLSAX::EndElement](#)
- [IDLffXMLSAX::EndEntity](#)
- [IDLffXMLSAX::EndPrefixMapping](#)
- [IDLffXMLSAX::Error](#)
- [IDLffXMLSAX::ExternalEntityDecl](#)
- [IDLffXMLSAX::FatalError](#)
- [IDLffXMLSAX::GetProperty](#)
- [IDLffXMLSAX::IgnorableWhitespace](#)
- [IDLffXMLSAX::Init](#)
- [IDLffXMLSAX::InternalEntityDecl](#)
- [IDLffXMLSAX::NotationDecl](#)
- [IDLffXMLSAX::ParseFile](#)
- [IDLffXMLSAX::ProcessingInstruction](#)
- [IDLffXMLSAX::SetProperty](#)
- [IDLffXMLSAX::SkippedEntity](#)
- [IDLffXMLSAX::StartCDATA](#)

- [IDLffXMLSAX::StartDocument](#)
- [IDLffXMLSAX::StartDTD](#)
- [IDLffXMLSAX::StartElement](#)
- [IDLffXMLSAX::StartEntity](#)
- [IDLffXMLSAX::StartPrefixmapping](#)
- [IDLffXMLSAX::StopParsing](#)
- [IDLffXMLSAX::UnparsedEntityDecl](#)
- [IDLffXMLSAX::Warning](#)

Version History

Introduced: 5.6

IDLffXMLSAX Properties

IDLffXMLSAX objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLffXMLSAX::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLffXMLSAX::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLffXMLSAX::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

FILENAME

A string containing the filename of the XML file being parsed.

Note

This property is only available during a parse operation.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

NAMESPACE_PREFIXES

A Boolean value indicating whether namespace prefixes are enabled. Namespace prefixes are enabled if this property is set to true. By default, namespace prefixes are disabled.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PARSER_LOCATION

A two-element array containing the approximate location of the parser within the entity being parsed. The first element of the array is set to the line number and the second element is set to the column number.

Note

This property is only available during a parse operation.

Property Type	Array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

PARSER_PUBLICID

A string containing the Public ID for the entity being parsed, if it is available. If the Public ID is not available, an empty string is returned.

Note

This property is only available during a parse operation.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

PARSER_URI

A string containing the base URI (System ID) for the entity being parsed, if it is available. If the value is available, it is always an absolute URI. If the System ID is not available, an empty string is returned.

Note

This property can be used to identify the document or external entity in diagnostics, or to resolve relative URIs. However, it is only available during a parse operation.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

SCHEMA_CHECKING

An integer value to indicating the type of validation the parser should perform. XML *Schemas* describe the structure and allowed contents of an XML document. Schemas are more robust than, and are envisioned as a replacement for, DTDs. By default, the parser will validate the parsed XML file against the specified schema, if one is provided. If no schema is provided, no validation will occur. Possible values are:

Value	Description
0	No validation.
1	Validate only if a schema is provided (the default).
2	Perform full schema constraint checking, if a schema is provided. This feature checks the schema grammar itself for additional errors. It does not affect the level of checking performed on document instances that use schema grammars.

Table 6-24: SCHEMA_CHECKING Values

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

VALIDATION_MODE

An integer value indicating the type of XML validation that the parser should perform. XML *Document Type Definitions* (DTDs) describe the structure and allowed contents of an XML document. By default, the parser will validate the parsed

XML file against the specified DTD, if one is provided; if no DTD specification is provided, no validation will occur. Possible values are:

Value	Description
0	No validation.
1	Validate only if a DTD is provided (the default).
2	Always perform validation. If this option is in force and no DTD is provided, every XML element in the document will generate an error.

Table 6-25: *VALIDATION_MODE* Values

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLffXMLSAX::AttributeDecl

The IDLffXMLSAX::AttributeDecl procedure method is called when the parser detects an `<!ATTLIST . . . >` declaration in a DTD. This method is called once for each attribute declared by the tag.

Syntax

Obj->[IDLffXMLSAX::]AttributeDecl, *eName*, *aName*, *Type*, *Mode*, *Value*

Arguments

eName

A string containing the name of the element for which the attribute is being declared.

aName

A string containing the name of the attribute being declared.

Type

A string that specifying the type of attribute being defined. Possible values are:

- 'CDATA'
- 'ID'
- 'IDREF'
- 'IDREFS'
- 'NMTOKEN'
- 'NMTOKENS'
- 'ENTITY'
- 'ENTITIES'

or two types of enumerated values. Enumerated values are encoded with parenthesized strings such as `(a|b|c)` to indicate that strings `a`, `b`, or `c` are permissible. If the string is an enumeration of notation names, the string `"NOTATION "` (note the space after the second "N") precedes the parenthesized string.

Mode

A string specifying restrictions on the value of the attribute. Possible values are:

- '#IMPLIED' - the application determines the value
- '#REQUIRED' - the value must be given; defaulting is not permitted
- '#FIXED' - only one value is permitted
- '' - a null string (the value specified by the *Value* argument is used as the default)

Value

A string containing the default value for the attribute. If *Value* contains a null string, no default value was specified.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::Characters

The IDLffXMLSAX::Characters procedure method is called when the parser detects text in the parsed document.

Syntax

Obj->[IDLffXMLSAX::]Characters, *Chars*

Arguments

Chars

A string containing the text detected by the parser.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::Cleanup

The IDLffXMLSAX::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. In most cases, you cannot call the Cleanup method directly. However, one exception to this rule does exist. If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLffXMLSAX:]Cleanup(*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::Comment

The IDLffXMLSAX::Comment procedure method is called when the parser detects a comment section of the form `<!-- . . . -->`.

Syntax

Obj->[IDLffXMLSAX:]Comment, *Comment*

Arguments

Comment

A string containing the text within the detected comment section, without the delimiting characters (“<!--” and “-->”).

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::ElementDecl

The IDLffXMLSAX::ElementDecl procedure method is called when the parser detects an `<!ELEMENT . . . >` declaration in the DTD.

Syntax

Obj->[IDLffXMLSAX::]ElementDecl, *Name*, *Model*

Arguments

Name

A string containing the name of the element.

Model

A string containing the *content model* (sometimes called the *content specification*) for the element, with all whitespace removed.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::EndCDATA

The IDLffXMLSAX::EndCDATA procedure method is called when the parser detects the end of a `<![CDATA[. . .]>` text section.

Syntax

Obj->[IDLffXMLSAX:]EndCDATA

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::EndDocument

The IDLffXMLSAX::EndDocument procedure method is called when the parser detects the end of the XML document.

Syntax

Obj->[IDLffXMLSAX:]EndDocument

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::EndDTD

The IDLffXMLSAX::EndDTD procedure method is called when the parser detects the end of a Document Type Definition (DTD).

Syntax

Obj->[IDLffXMLSAX::]EndDTD

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::EndElement

The IDLffXMLSAX::EndElement procedure method is called when the parser detects the end of an element.

Syntax

Obj->[IDLffXMLSAX:]EndElement, *URI*, *Local*, *qName*

Arguments

URI

A string containing the namespace URI with which the element is associated, if any.

Note

A URI (or Uniform Resource Identifier) refers to the generic set of all names and addresses which are short strings which refer to objects.

Local

A string containing the element name with any prefix removed, if the element is associated with a namespace URI. If the element is not associated with a namespace URI, this variable will contain an empty string.

qName

A string containing the element name found in the XML file.

Note

If the element is associated with a namespace URI, this variable may contain an empty string.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::EndEntity

The IDLffXMLSAX::EndEntity procedure method is called when the parser detects the end of an internal or external entity expansion.

Syntax

Obj->[IDLffXMLSAX::]EndEntity, *Name*

Arguments

Name

A string containing the name of the entity.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::EndPrefixMapping

The IDLffXMLSAX::EndPrefixMapping procedure method is called when a previously declared prefix mapping goes out of scope.

Syntax

Obj->[IDLffXMLSAX::]EndPrefixMapping, *Prefix*

Arguments

Prefix

A string containing the namespace prefix that is going out of scope.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::Error

The IDLffXMLSAX::Error procedure method is called when the parser detects an error that is not expected to be fatal. This method prints an IDL error string to the IDL output log and allows the parser to continue processing.

For example, a violation of XML validity constraints is generally a non-fatal error.

Note

This method will cause error messages to be printed to the IDL output log. If you would like your application to hide error messages from the user (or display them in some other fashion), override this method in your subclass of the IDLffXMLSAX object class. If you do override this method, the error message will not be printed to the output log unless you explicitly call the superclass method.

Syntax

Obj->[IDLffXMLSAX::]Error, *SystemID*, *LineNumber*, *ColumnNumber*, *Message*

Arguments

SystemID

A string containing the URI of the associated text.

LineNumber

A longword integer representing the line number that contains the error.

ColumnNumber

A longword integer representing the column number that contains the error.

Message

A string containing the error message sent to the IDL output log.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::ExternalEntityDecl

The IDLffXMLSAX::ExternalEntityDecl procedure method is called when the parser detects an `<!ENTITY . . . >` declarations in the DTD for a parsed external entity.

Syntax

Obj->[IDLffXMLSAX::]ExternalEntityDecl, *Name*, *PublicID*, *SystemID*

Arguments

Name

A string containing the entity name.

PublicID

A string containing the Public ID for the entity.

Note

If this value is not specified in the entity declaration, this variable will contain an empty string.

SystemID

A string containing the System ID for the entity, provided as an absolute URI.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::FatalError

The IDLffXMLSAX::FatalError procedure method is called when the parser detects a fatal error. When called, parsing will normally stop, but may sometimes continue long enough to report further errors. This method prints an IDL error string to the IDL output log.

Syntax

Obj->[IDLffXMLSAX::]FatalError, *SystemID*, *LineNumber*, *ColumnNumber*,
Message

Arguments

SystemID

A string containing the URI of the associated text.

LineNumber

A longword integer representing the line number that contains the error.

ColumnNumber

A longword integer representing the column number that contains the error.

Message

A string containing the error message sent to the IDL output log.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::GetProperty

The IDLffXMLSAX::GetProperty procedure method is used to get the values of various properties of the parser.

Syntax

Obj->[IDLffXMLSAX::]GetProperty [, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLffXMLSAX Properties](#)” on page 3042 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 5.6

IDLffXMLSAX::IgnorableWhitespace

The IDLffXMLSAX::IgnorableWhitespace procedure method is called when the parser detects whitespace that separates elements in an element content model.

Syntax

Obj->[IDLffXMLSAX::]IgnorableWhitespace, *Chars*

Arguments

Chars

A string containing the whitespace detected by the parser. Whitespace can consist of spaces, tabs, or newline characters in any combination.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::Init

The IDLffXMLSAX::Init function method initializes an XML parser object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. In most cases, you cannot call the Init method directly. However, one exception to this rule does exist. If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLffXMLSAX' [, *PROPERTY=value*])

or

Result = *Obj*->[IDLffXMLSAX::]Init([*PROPERTY=value*]) (*Only in a subclass' Init method.*)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLffXMLSAX Properties](#)” on page 3042 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.6

IDLffXMLSAX::InternalEntityDecl

The IDLffXMLSAX::InternalEntityDecl procedure method is called when the parser detects an `<!ENTITY . . . >` declaration in a DTD for (parsed) internal entities. The entity can be either a general entity or a parameter entity.

Syntax

Obj->[IDLffXMLSAX::]InternalEntityDecl, *Name*, *Value*

Arguments

Name

A string containing the entity name. Names that start with the “%” character are parameter entities; all others are general entities.

Value

A string containing the entity value. The entity value can contain arbitrary XML content, which will be reparsed when the entity is expanded.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::NotationDecl

The IDLffXMLSAX::NotationDecl procedure method is called when the parser detects a `<!NOTATION . . . >` declaration in a DTD.

Syntax

Obj->[IDLffXMLSAX:]NotationDecl, *Name*, *PublicID*, *SystemID*

Arguments

Name

A string containing the notation name.

PublicID

A string containing the Public ID for the notation.

Note

If this value is not specified in the notation declaration, this variable will contain an empty string.

SystemID

A string containing the System ID for the notation, provided as an absolute URI.

Note

If this value is not specified in the notation declaration, this variable will contain an empty string.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::ParseFile

The IDLffXMLSAX::ParseFile procedure method parses the specified XML file. During the parsing operation, different object methods are called as different items within the XML file are detected. When this method returns, the parse operation is complete.

Syntax

Obj->[IDLffXMLSAX::]ParseFile, *Filename*

Arguments

Filename

A string containing the full path name of the XML file to parse.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::ProcessingInstruction

The IDLffXMLSAX::ProcessingInstruction procedure method is called when the parser detects a processing instruction.

Syntax

Obj->[IDLffXMLSAX::]ProcessingInstruction, *Target*, *Data*

Arguments

Target

A string specifying the target, which is the application that should process the instruction.

Data

A string specifying the data to be passed to the application specified by *Target*.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX:: SetProperty

The IDLffXMLSAX::SetProperty procedure method is used to set the values of various properties of the parser.

Syntax

Obj->[IDLffXMLSAX::]SetProperty [, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under [“IDLffXMLSAX Properties”](#) on page 3042 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.6

IDLffXMLSAX::SkippedEntity

The IDLffXMLSAX::SkippedEntity procedure method is called when the parser skips an entity and validation is not being performed. This method is rarely called by SAX parsers.

Syntax

Obj->[IDLffXMLSAX:]SkippedEntity, *Name*

Arguments

Name

A string containing the name of the entity that was skipped.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StartCDATA

The IDLffXMLSAX::StartCDATA procedure method is called when the parser detects the beginning of a `<![CDATA[. . .]>` text section.

Syntax

Obj->[IDLffXMLSAX:]StartCDATA

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StartDocument

The IDLffXMLSAX::StartDocument procedure method is called when the parser begins processing a document, and before any data is processed.

Syntax

Obj->[IDLffXMLSAX::]StartDocument

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StartDTD

The IDLffXMLSAX::StartDTD procedure method is called when the parser detects the beginning of a Document Type Definition (DTD).

Syntax

Obj->[IDLffXMLSAX::]StartDTD, *Name*, *PublicID*, *SystemID*

Arguments

Name

A string containing the declared name of the root element for the document.

PublicID

A string containing the *normalized* version of the Public ID (a URI) declared for the external subset, or an empty string if no external subset was declared. *Normalization* involves removal of unnecessary “.” and “. .” segments from the URI.

SystemID

A string containing the System ID (a URI) of the external subset, or an empty string if no external subset was declared.

Note

This URI has not been resolved into an absolute URI.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StartElement

The IDLffXMLSAX::StartElement procedure method is called when the parser detects the beginning of an element.

Syntax

Obj->[IDLffXMLSAX::]StartElement, *URI*, *Local*, *qName* [, *attName*, *attValue*]

Arguments

URI

A string containing the namespace URI with which the element is associated, if any.

Local

A string containing the element name with any prefix removed, if the element is associated with a namespace URI. If the element is not associated with a namespace URI, this variable will contain an empty string.

qName

A string containing the element name found in the XML file.

Note

If the element is associated with a namespace URI, this variable may contain an empty string.

attrName

A string array representing the names of the attributes associated with the element, if any.

attrValue

A string array representing the values of each attribute associated with the element, if any. The returned array will have the same number of elements as the array returned in the *attrName* keyword variable.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StartEntity

The IDLffXMLSAX::StartEntity procedure method is called when the parser detects the start of an internal or external entity expansion.

Syntax

Obj->[IDLffXMLSAX::]StartEntity, *Name*

Arguments

Name

A string containing the name of the entity.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StartPrefixmapping

The IDLffXMLSAX::StartPrefixMapping procedure method is called when the parser detects the beginning of a namespace declaration.

Syntax

Obj->[IDLffXMLSAX::]StartPrefixmapping, *Prefix*, *URI*

Arguments

Prefix

A string containing the prefix, which is being mapped. If the variable specified by *Prefix* contains an empty string, the mapping is for the default element namespace.

URI

A string containing the URI of the prefix namespace.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::StopParsing

Call the IDLffXMLSAX::StopParsing procedure method during a parse operation to halt the operation and cause the ParseFile method to return. This may be useful when parsing large XML files and the desired information is known to have been returned.

Syntax

Obj->[IDLffXMLSAX:]StopParsing

Arguments

None

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::UnparsedEntityDecl

The IDLffXMLSAX::UnparsedEntityDecl procedure method is called when the parser detects an `<!ENTITY . . . >` declaration that includes the `NDATA` keyword, indicating that the entity is not meant to be parsed. The value of the `NDATA` keyword generally specifies the name of a *notation*, which in turn specifies the type of data.

Syntax

Obj->[IDLffXMLSAX::]UnparsedEntityDecl, *Name*, *PublicID*, *SystemID*, *Notation*

Arguments

Name

A string containing the name of the unparsed entity.

PublicID

A string containing the Public ID of the notation specified by the entity's `NDATA` keyword, or an empty string if no Public ID was declared.

SystemID

A string containing the System ID of the notation specified by the entity's `NDATA` keyword. This value is normally an absolute URI.

Notation

A string containing the name of the notation specified by the entity's `NDATA` keyword.

Keywords

None

Version History

Introduced: 5.6

IDLffXMLSAX::Warning

The IDLffXMLSAX::Warning procedure method is called when the parser detects a problem during processing. This method prints an IDL error string to the IDL output log and allows the parser to continue processing.

Note

This method will cause error messages to be printed to the IDL output log. If you would like your application to hide error messages from the user (or display them in some other fashion), override this method in your subclass of the IDLffXMLSAX object class. If you do override this method, the error message will not be printed to the output log unless you explicitly call the superclass method.

Syntax

Obj->[IDLffXMLSAX::]Warning, *SystemID*, *LineNumber*, *ColumnNumber*,
Message

Arguments

SystemID

A string containing the URI of the text that generated the error.

LineNumber

A longword integer representing the line number that contains the error.

ColumnNumber

A longword integer representing the column number that contains the error.

Message

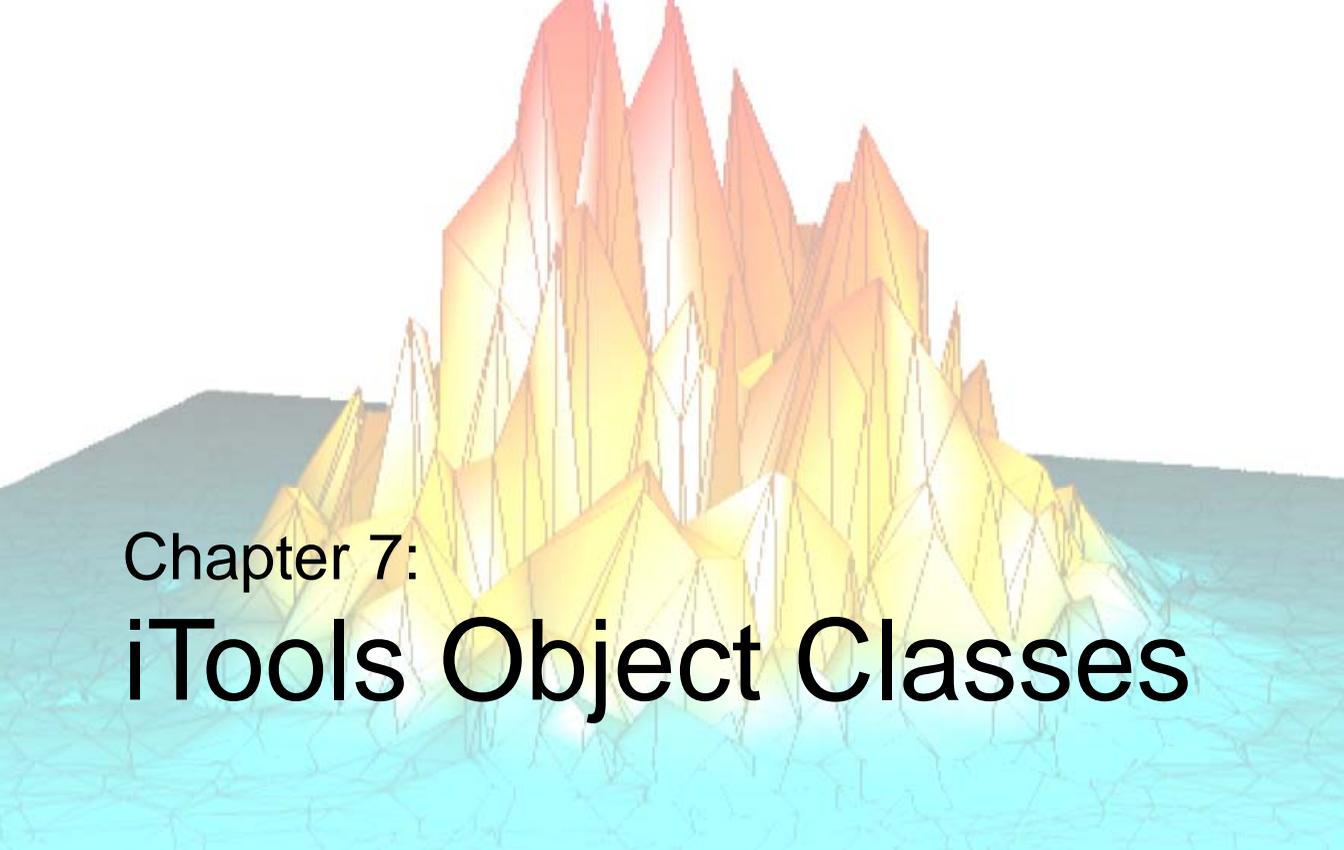
A string containing the error message.

Keywords

None

Version History

Introduced: 5.6



Chapter 7: iTools Object Classes

This chapter describes IDL's built-in iTool object classes.

IDLitCommand	3083	IDLitManipulatorVisual	3263
IDLitCommandSet	3095	IDLitOperation	3271
IDLitComponent	3101	IDLitParameter	3292
IDLitContainer	3127	IDLitParameterSet	3315
IDLitData	3141	IDLitReader	3330
IDLitDataContainer	3159	IDLitTool	3343
IDLitDataOperation	3172	IDLitToolbase	3404
IDLitIMessaging	3187	IDLitUI	3406
IDLitManipulator	3205	IDLitVisualization	3425
IDLitManipulatorContainer	3235	IDLitWindow	3491
IDLitManipulatorManager	3254	IDLitWriter	3525

IDLitCommand

The IDLitCommand class provides a dynamic data dictionary storage system. Its methods and properties allow the iTool developer to store information about the execution of an iTool operation on a single target object. It provides a tag-based mechanism for storing and retrieving this information, which allows the iTool developer to easily implement undo and redo functionality for an operation.

This class is written in the IDL language. Its source code can be found in the file `idlitcommand__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitComponent](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitCommand::Init](#)” on page 3092.

Properties

Objects of this class have the following properties. See “[IDLitCommand Properties](#)” on page 3085 for details on individual properties.

- [TARGET_IDENTIFIER](#)
- [OPERATION_IDENTIFIER](#)

In addition, objects of this class inherit the properties of the superclass of this class.

Methods

This class has the following methods:

- [IDLitCommand::AddItem](#)
- [IDLitCommand::Cleanup](#)
- [IDLitCommand::GetItem](#)

- [IDLitCommand::GetProperty](#)
- [IDLitCommand::GetSize](#)
- [IDLitCommand::Init](#)
- [IDLitCommand::SetProperty](#)

In addition, this class inherits the methods of its superclass.

Examples

See “[Creating a New Generalized Operation](#)” in Chapter 7 of the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitCommand Properties

IDLitCommand objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitCommand::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitCommand::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitCommand::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

TARGET_IDENTIFIER

The iTool identifier of the *target* object for this command. This is the item on which the operation associated with the command will execute.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

OPERATION_IDENTIFIER

The iTool identifier for the *operation* associated with this command. When an undo-redo operation takes place, the command object is passed to this operation.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLitCommand::AddItem

The IDLitCommand::AddItem function method adds the specified data item to the data dictionary associated with this object.

Note

If a data item with the specified identifying string is already in the data dictionary, the new value is not added unless the **OVERWRITE** keyword is set, in which case the old value is discarded.

Syntax

Result = *Obj*->[IDLitCommand:]AddItem(*StrItem*, *Item*[, /**OVERWRITE**])

Return Value

Returns 1 if the item was successfully added to the data dictionary, or 0 if the new value was not added.

Arguments

StrItem

A scalar string used to identify the data specified by the *Item* argument in the data dictionary. This string is treated in a case-insensitive manner.

Item

A data item of any IDL data type.

Keywords

OVERWRITE

Normally, if an item already exists in the data dictionary of an IDLitCommand object, it is not overwritten. If this keyword is set, the original value will be overwritten.

Note

If the item being replaced uses dynamic memory (that is, if it is a pointer or object reference), the memory must be released by calling **PTR_FREE** or **OBJ_DESTROY** before overwriting the value.

Version History

Introduced: 6.0

IDLitCommand::Cleanup

The IDLitCommand::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitCommand::]Cleanup() (*In a subclass' Init method only.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitCommand::GetItem

The IDLitCommand::GetItem function method retrieves the specified item from the data dictionary associated with this object.

Syntax

Result = *Obj*->[IDLitCommand::]GetItem(*StrItem*, *Item*)

Return Value

Returns 1 if the *Item* is retrieved successfully, or 0 if *Item* is not found in the data dictionary.

Arguments

StrItem

A scalar string used to identify the data to be retrieved from the data dictionary. This string is treated in a case-insensitive manner.

Item

A named IDL variable that will contain the IDLitData object retrieved from the data dictionary.

Keywords

None

Version History

Introduced: 6.0

IDLitCommand::GetProperty

The IDLitCommand::GetProperty procedure method retrieves the value of a property or group of properties of a command object.

Syntax

Obj->[IDLitCommand::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitCommand Properties](#)” on page 3085 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s super-class.

Version History

Introduced: 6.0

IDLitCommand::GetSize

The IDLitCommand::GetSize function method returns an approximate value for the amount of memory being used by the items in the data dictionary associated with this object.

Syntax

Result = *Obj*->[IDLitCommand::]GetSize([, /KILOBYTES])

Return Value

Returns the number of bytes of memory used by the data dictionary associated with this object. The returned value is an approximation.

Arguments

None

Keywords

KILOBYTES

Set this keyword to return the number of kilobytes of memory being used by the data dictionary. By default, this function returns the number of bytes of memory being used.

Version History

Introduced: 6.0

IDLitCommand::Init

The IDLitCommand::Init function method initializes the object and allows specification of items associated with it.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitCommand' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLitCommand::]Init([PROPERTY=value])  
(In a subclass' Init method only.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLitCommand Properties](#)” on page 3085 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object ([IDLitComponent](#)).

Version History

Introduced: 6.0

IDLitCommand:: SetProperty

The IDLitCommand::SetProperty procedure method sets the value of a property or group of properties for the command object.

Syntax

Obj->[IDLitCommand::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLitCommand Properties](#)” on page 3085 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitCommandSet

The IDLitCommandSet class stores a collection of IDLitCommand objects, allowing a group of commands to be managed as a single item. Command sets are useful as containers for IDLitCommand objects generated by the application of an iTool operation to multiple target items; the generated command set objects are used to provide undo/redo functionality to the iTool system.

This class is written in the IDL language. Its source code can be found in the file `idlitcommandset__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDL_Container](#)

[IDLitCommand](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitCommandSet::Init](#)” on page 3100.

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLitCommandSet::Cleanup](#)
- [IDLitCommandSet::GetSize](#)
- [IDLitCommandSet::Init](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [“Creating a New Generalized Operation”](#) in Chapter 7 of the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitCommandSet Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLitCommandSet::Cleanup

The IDLitCommandSet::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, Obj

or

Obj->[IDLitCommandSet::]Cleanup() (*In a subclass' Init method only.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitCommandSet::GetSize

The IDLitCommandSet::GetSize function method returns an approximate value for the amount of memory being used by the items contained by this command set. When called, the method will call the GetSize method on all of the command objects that it contains and generate a sum of the values.

Syntax

Result = *Obj*->[IDLitCommand:]GetSize([/KILOBYTES])

Return Value

Returns the number of bytes of memory used by the data dictionary associated with this object. The returned value is an approximation.

Arguments

None

Keywords

KILOBYTES

Set this keyword to return the number of kilobytes of memory being used. By default, this function returns the number of bytes of memory being used.

Version History

Introduced: 6.0

IDLitCommandSet::Init

The IDLitCommandSet::Init function method initializes the object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitCommandSet')
```

or

```
Result = Obj->[IDLitCommandSet::]Init() (In a subclass' Init method only.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

All keywords are passed to the superclass of this object ([IDLitCommand](#)).

Version History

Introduced: 6.0

IDLitComponent

The IDLitComponent class is the base component from which all iTool components should subclass. The IDLitComponent class provides support for property descriptors and property registration, and defines properties that are common to all iTool component objects.

Note

This class is now a superclass of all the atomic graphical (IDLgr*) object classes.

Superclasses

This class has no superclasses.

Note

See the [“iTools Object Model Diagram”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See [“IDLitComponent::Init”](#) on page 3113.

Properties

Objects of this class have the following properties. See [“IDLitComponent Properties”](#) on page 3103 for details on individual properties.

- [COMPONENT_VERSION](#)
- [DESCRIPTION](#)
- [HELP](#)
- [ICON](#)
- [IDENTIFIER](#)
- [NAME](#)
- [PRIVATE](#)
- [UVALUE](#)

Methods

This class has the following methods:

- [IDLitComponent::Cleanup](#)
- [IDLitComponent::EditUserDefProperty](#)
- [IDLitComponent::GetFullIdentifier](#)
- [IDLitComponent::GetProperty](#)
- [IDLitComponent::GetPropertyAttribute](#)
- [IDLitComponent::GetPropertyByIdentifier](#)
- [IDLitComponent::Init](#)
- [IDLitComponent::QueryProperty](#)
- [IDLitComponent::RegisterProperty](#)
- [IDLitComponent::Restore](#)
- [IDLitComponent::SetProperty](#)
- [IDLitComponent::SetPropertyAttribute](#)
- [IDLitComponent::SetPropertyByIdentifier](#)
- [IDLitComponent::UpdateComponentVersion](#)

Version History

Introduced: 6.0

COMPONENT_VERSION added: 6.1

IDLitComponent Properties

IDLitComponent objects have the following properties. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitComponent::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitComponent::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitComponent::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

COMPONENT_VERSION

A integer value representing the version of the component object. This value is set by RSI and is intended to aid in determining how a component object should be restored by the Restore method.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DESCRIPTION

A string giving the full name or description of this object.

Property Type	STRING		
Name String	Description		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HELP

A scalar string representing the help topic associated with this object. If this property is not set, or is set to a null string, then the object class name will be used as the default help topic.

Property Type	String
----------------------	--------

Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ICON

A string specifying the name of an icon to be associated with this object. Typically, this property is the name of a bitmap file used when displaying this object in a tree view. See “[System Resources](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for details.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDENTIFIER

A string containing the *object identifier* for this object. If this property is not specified, then the NAME property is used as the identifier. See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for details about how identifiers are named.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

NAME

A string giving the human-readable name of this object.

Property Type	STRING		
Name String	Name		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PRIVATE

A boolean value that indicates whether the object should be marked as private. Objects marked private (and all of their children) are not displayed in the graphical iTool browser windows.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

UVALUE

A value of any type containing any information you wish. If you set this user value equal to a pointer or object reference that does not belong to a container, you should explicitly destroy that pointer or object reference when destroying the object of which this property is a user value.

Property Type	User Defined		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Version History

COMPONENT_VERSION property added: 6.1

IDLitComponent::Cleanup

The IDLitComponent::Cleanup procedure method performs all cleanup on the object, and should be called by the Cleanup method of a subclass.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitComponent::]Cleanup (*only in subclass' Cleanup method*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitComponent::EditUserDefProperty

The IDLitComponent::EditUserDefProperty function method defines the interface that is displayed when a user selects the “Edit” button on a user-defined property in the property sheet interface. Typically, this method will display an interactive dialog that allows the user to change the value of the property.

Note

An iTool object class that includes a user-defined property *must* implement this method if the property is displayed in a property sheet — that is, if the property is registered and not hidden. If an iTool object class has no user-defined properties that are displayed in a property sheet, there is no need to supply an EditUserDefProperty method.

See “[User Defined Property Types](#)” in Chapter 4 of the *iTool Developer’s Guide* manual for a discussion of how the property sheet interface displays user-defined properties.

Syntax

Result = *Obj*->[IDLitComponent::] EditUserDefProperty(*iTool*, *PropertyIdentifier*)

Return Value

Returns a 1 if the property value was changed, or a 0 if the property value was not changed.

Note

If the return value is 1, the property sheet interface automatically updates the displayed property value using the value of the property’s USERDEF attribute. If the return value is 0, no update takes place.

Arguments

iTool

An object reference to the current iTool object.

PropertyIdentifier

A string containing the property identifier of the user-defined property.

Tip

Since there can only be one `EditUserDefProperty` method for each class, you can use the *PropertyIdentifier* argument to determine which user-defined property is being edited.

Keywords

None

Example

The following is the `EditUserDefProperty` method of the `IDLitOpConvolution` operation class. Selecting and editing the `Kernel` property of this operation displays a dialog that allows the user to edit a convolution kernel.

```
FUNCTION IDLitOpConvolution::EditUserDefProperty, oTool, $
    identifier

CASE identifier OF

    'KERNEL': RETURN, oTool->DoUIService('ConvolKernel', self)

ELSE:

ENDCASE

RETURN, 0

END
```

This method simply checks the property identifier to determine whether it matches the string `'KERNEL'`. If it does, it returns the value returned by the `DoUIService` method; otherwise it returns zero. In this case, the `DoUIService` method actually handles the modification of the property value. See [“User Defined Property Types”](#) in Chapter 4 of the *iTool Developer’s Guide* manual for additional discussion.

Version History

Introduced: 6.0

IDLitComponent::GetFullIdentifier

The `IDLitComponent::GetFullIdentifier` function method navigates the iTool object container hierarchy of the object on which it is called and retrieves the fully-qualified object identifier. The full identifier is constructed by recursively retrieving the parent object reference and prepending this to the identifier, separated by `/` characters. The parent must inherit from the `IDLitContainer` class. The full identifier string has the following form:

```
/TOPID/GRANDPARENTID/PARENTID/OBJID
```

If the *Objref* argument is specified, the function returns the object identifier path relative to the specified object.

See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for additional discussion of object identifiers.

Syntax

```
Result = Obj->[IDLitComponent::]GetFullIdentifier([Objref])
```

Return Value

Returns a string containing the object identifier.

Arguments

Objref

An object reference to an iTool component object. If this argument is specified, the returned value is an relative object identifier, beginning with `Objref`. If this argument is not specified, the returned value is a fully-qualified object identifier.

Keywords

None

Version History

Introduced: 6.0

IDLitComponent::GetProperty

The IDLitComponent::GetProperty procedure method retrieves the value of an IDLitComponent property or properties.

Syntax

Obj->[IDLitComponent::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitComponent Properties](#)” on page 3103 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitComponent::GetPropertyAttribute

The IDLitComponent::GetPropertyAttribute procedure method retrieves property attribute values for a registered property.

Syntax

```
Obj->[IDLitComponent::]GetPropertyAttribute, PropertyIdentifier  
[, TYPE=variable]
```

Arguments

PropertyIdentifier

A string containing the property identifier of the registered property.

Keywords

Any keywords to the RegisterProperty method followed by the word “Get” can be retrieved. In addition, the following keyword is available:

TYPE

Set this argument to a named variable that returns the TYPE code as an integer, as specified under the *Type* argument to [IDLitComponent::RegisterProperty](#).

Version History

Introduced: 6.0

IDLitComponent::GetPropertyByIdentifier

The IDLitComponent::GetPropertyByIdentifier function method retrieves the value of an IDLitComponent property.

Tip

The IDLitComponent::GetPropertyByIdentifier method is similar to the IDLitComponent::GetProperty method, but is useful for cases where the *property identifier* is defined at runtime.

Syntax

```
Result = Obj->[IDLitComponent::]GetPropertyByIdentifier(PropertyIdentifier,  
Value)
```

Return Value

Returns 1 if the property value is defined, or 0 if the property value is undefined.

Arguments

PropertyIdentifier

A string containing the property identifier of the property. *PropertyIdentifier* does not need to be a registered property, but must be a valid keyword name for the GetProperty method of the component.

Value

A named variable that will contain the value of the property. If the property value is currently undefined, then the contents of the variable specified by *Value* will not be modified.

Keywords

None

Version History

Introduced: 6.0

IDLitComponent::Init

The IDLitComponent::Init function method initializes the IDLitComponent object, and should be called by the Init method of a subclass.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitComponent'[, PROPERTY=value] )
```

or

```
Result = Obj->[IDLitComponent:]Init([, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under [“IDLitComponent Properties”](#) on page 3103 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitComponent::QueryProperty

The IDLitComponent::QueryProperty function method checks whether a property identifier is registered, or retrieves a list of all registered properties.

Syntax

Result = *Obj*->[IDLitComponent::]QueryProperty([*PropertyIdentifier*])

Return Value

Returns a 1 if *PropertyIdentifier* is a scalar string that corresponds to a valid registered property, or a 0 otherwise. If *PropertyIdentifier* is an array, the result is an array of 1s and 0s.

If *PropertyIdentifier* is not specified, *Result* is a string array containing the identifiers of all registered properties.

Arguments

PropertyIdentifier

A scalar string or string array containing property identifiers.

Keywords

None

Version History

Introduced: 6.0

IDLitComponent::RegisterProperty

The IDLitComponent::RegisterProperty procedure method registers a property as belonging to the component. Only registered properties are displayed in the graphical property sheet interface for the object.

Syntax

```
Obj->[IDLitComponent::]RegisterProperty, PropertyIdentifier [, Type]
  [, /BOOLEAN] [, /COLOR] [, DESCRIPTION=string]
  [, ENUMLIST=stringvector] [, /FLOAT] [, /HIDE] [, /INTEGER]
  [, /LINESTYLE] [, NAME=string] [, /SENSITIVE] [, /STRING] [, /SYMBOL]
  [, /THICKNESS] [, /UNDEFINED] [, USERDEF=string]
  [, VALID_RANGE=vector]
```

Arguments

PropertyIdentifier

A scalar string containing the property identifier. GetProperty and SetProperty methods accept this string as a keyword.

Type

An optional integer argument representing the property type. Values recognized by Property Sheets are:

- 0 = USERDEF
- 1 = BOOLEAN
- 2 = INTEGER
- 3 = FLOAT
- 4 = STRING
- 5 = COLOR
- 6 = LINESTYLE
- 7 = SYMBOL
- 8 = THICKNESS
- 9 = ENUMLIST

If *Type* is not supplied, then one of the type keywords must be set instead.

Keywords

Some of the following keywords correspond to *attributes* of the property being registered. Keywords followed by the word “Get” indicate attributes that can be retrieved by the `IDLitComponent::GetPropertyAttribute` method. Keywords followed by the word “Set” indicate attributes that can be set by the `IDLitComponent::SetPropertyAttribute` method.

Some of the following keywords correspond to iTool property data types, which can also be specified via the *Type* argument. Note that if both the *Type* argument and one of the corresponding keywords are set, the keyword value will be honored.

BOOLEAN

Set this keyword to indicate that the property type is BOOLEAN. Properties of type BOOLEAN must accept an integer value of either 0 (“False”) or 1 (“True”). Setting this keyword is equivalent to setting the TYPE argument equal to 1.

COLOR

Set this keyword to indicate that the property type is COLOR. Properties of type COLOR must accept a three-element integer array containing an RGB triplet. Setting this keyword is equivalent to setting the TYPE argument equal to 5.

DESCRIPTION

(*Get, Set*)

Set this keyword to a string containing the description of this property.

Note

Do not confuse the DESCRIPTION *property attribute* with the DESCRIPTION *property*. The DESCRIPTION property attribute is *not* displayed in the graphical property sheet interface.

ENUMLIST

(*Get, Set*)

Set this keyword equal to an array of strings to be displayed in the dropdown menu displayed by a property of type ENUMLIST. The default is a scalar null string. Properties of type ENUMLIST report their value as an integer between 0 and $n-1$, where n is the number of elements in the enumerated list. Setting this keyword implies that the TYPE argument is equal to 9.

FLOAT

Set this keyword to indicate that the property type is FLOAT. Properties of type FLOAT must accept a scalar double-precision floating-point value. Setting this keyword is equivalent to setting the TYPE argument equal to 3.

HIDE

(Get, Set)

Set this keyword to hide the given property when displaying the graphical property sheet interface. By default, all registered properties are displayed.

INTEGER

Set this keyword to indicate that the property type is INTEGER. Properties of type INTEGER must accept a scalar integer value. Setting this keyword is equivalent to setting the TYPE argument equal to 2.

LINestyle

Set this keyword to indicate that the property type is LINestyle. Properties of type LINestyle must accept an integer specifying a pre-defined linestyle, as described under the [LINestyle](#) property of the [IDLgrPolyline](#) class. Setting this keyword is equivalent to setting the TYPE argument equal to 6.

NAME

(Get, Set)

Set this keyword to a string giving the human-readable name for the property. The NAME string will be used when displaying the graphical property sheet interface. If NAME is not set, the value of the *PropertyIdentifier* argument is used.

Note

Do not confuse the NAME *property attribute* with the NAME *property*.

SENSITIVE

(Get, Set)

Set this keyword to zero to make the given property insensitive in the graphical property sheet interface. By default, registered properties are sensitive.

STRING

Set this keyword to indicate that the property type is **STRING**. Properties of type **STRING** must accept a scalar string of any length. Setting this keyword is equivalent to setting the **TYPE** argument equal to 4.

SYMBOL

Set this keyword to indicate that the property type is **SYMBOL**. Properties of type **SYMBOL** must accept an integer specifying one of the pre-defined symbol types described under [IDLgrSymbol::Init](#). Setting this keyword is equivalent to setting the **TYPE** argument equal to 7.

THICKNESS

Set this keyword to indicate that the property type is **THICKNESS**. Properties of type **THICKNESS** must accept an integer between 1 and 10 specifying the line thickness in points. Setting this keyword is equivalent to setting the **TYPE** argument equal to 8.

UNDEFINED

(Get, Set)

Set this keyword to indicate that the property should appear as a blank cell when displayed in the graphical property sheet interface. This is useful in situations where properties of multiple objects are displayed in the property sheet (either because multiple objects are selected, or because the objects have been grouped).

Note

It is the iTool developer's responsibility to set this property attribute back to zero. Use the **SET_DEFINED** field of the **WIDGET_PROPERTY SHEET** event structure to determine when to set the **UNDEFINED** attribute back to zero.

USERDEF

(Get, Set)

Set this keyword to a string that represents a user-defined property. Properties of type **USERDEF** can accept and return variables of any type. When the actual property value changes, it is assumed that the string value of the **USERDEF** attribute will also be modified to reflect the new property value. Setting this keyword implies that the **TYPE** argument is equal to 0.

VALID_RANGE

(Get, Set)

For INTEGER or FLOAT types, set this keyword to a two- or three-element vector specifying the [minimum, maximum] or [minimum, maximum, increment] for valid values of the property.

What is displayed for the property sheet number cell depends upon the following:

- If this attribute is not specified — the property sheet displays an editable text field where masked editing is enforced, and the range is that of the data type. The only accepted keystrokes are the ten digits and the plus and minus signs. If the float type is specified, the decimal, and “d” and “e” (scientific exponent notation tokens) are also allowed.
- If a range is specified without an increment — the property sheet displays a spinner control that allows the user to click, or click and hold the up or down buttons to change the value. For an integer type, the increment is one. For a float type, the increment equals approximately 1/100 of the range. For example, a range of 100 results in an increment of 1. A value is snapped to the nearest allowable value when a value outside the range, or not equal to an incremental value, is entered. The editable text field (featuring masked editing) also allows the user to enter a numerical value.
- If a range and increment are specified — the property sheet displays a slider with a marker that can be repositioned to change the value. A value is snapped to the nearest allowable value when a value outside the range, or not equal to an incremental value, is entered. The increment value must be positive and will be converted to an integer for INTEGER types. Specifying an increment of 0 (zero) is the same as specifying a range without an increment. The editable text field (featuring masked editing) also allows the user to enter a numerical value.

The following code creates properties containing various number cells:

```

; Create a number cell that supports masked editing.
self->RegisterProperty, 'integer1', /INTEGER, $
    NAME = 'Integer1', description = 'Integer'

; Create a number cell with spinner control and masked
; editing as only the range is specified.
self->RegisterProperty, 'integer2', /INTEGER, $
    NAME = 'Integer2', VALID_RANGE = [ -100, 100 ]

; Creates a number cell with slider control and masked
; editing as a range and increment are specified.
self->RegisterProperty, 'float1', /FLOAT, $
    NAME = 'Float1', VALID_RANGE = [ -20.0D, 5.0D, 0.5D ]

```

Version History

Introduced: 6.0

Spinner control functionality added to `VALID_RANGE`: 6.1

IDLitComponent::Restore

The IDLitComponent::Restore procedure method performs any transitional work required after an object of this class has been restored from a SAVE file, to ensure that its state is appropriate for the current version.

Note

This method is designed to be implemented by a subclass of IDLitComponent, to perform any transition work necessary when a component object is restored.

Syntax

Obj->[IDLitComponent::]Restore

Arguments

None

Keywords

None

Version History

Introduced: 6.1

See Also

[IDLitComponent::UpdateComponentVersion](#), [IDL_Savefile](#), [SAVE](#), [RESTORE](#)

IDLitComponent:: SetProperty

The IDLitComponent::SetProperty procedure method sets the value of an IDLitComponent property or properties.

Syntax

Obj->[IDLitComponent::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLitComponent Properties](#)” on page 3103 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitComponent:: SetPropertyAttribute

The IDLitComponent::SetPropertyAttribute procedure method sets one or more property attributes for a registered property.

Syntax

Obj->[IDLitComponent::]SetPropertyAttribute , *PropertyIdentifier*

Arguments

PropertyIdentifier

A string containing the property identifier of the registered property.

Keywords

Any keywords to the [IDLitComponent::RegisterProperty](#) method followed by the word “Set” can be set.

Version History

Introduced: 6.0

IDLitComponent:: SetPropertyByIdentifier

The IDLitComponent::SetPropertyByIdentifier procedure method sets the value of an IDLitComponent property.

Tip

The IDLitComponent::SetPropertyByIdentifier method is similar to the IDLitComponent::SetProperty method, but is useful for cases where the *PropertyIdentifier* is defined at runtime, and avoids the overhead of creating an *_EXTRA* structure to pass to SetProperty.

Syntax

Obj->[IDLitComponent::]SetPropertyByIdentifier, *PropertyIdentifier*, *Value*

Arguments

PropertyIdentifier

A string containing the property identifier of the property. *PropertyIdentifier* does not need to be a registered property, but must be a valid keyword name for the SetProperty method of the component.

Value

The new value for the property.

Keywords

None

Version History

Introduced: 6.0

IDLitComponent::UpdateComponentVersion

The IDLitComponent::UpdateComponentVersion procedure method updates the value of the COMPONENT_VERSION property for the specified object to match the version associated with the current release of IDL. This method is intended for use after calling the Restore method on a component object, so that the object can be identified as having been restored.

Syntax

Obj->[IDLitComponent::]UpdateComponentVersion

Arguments

None

Keywords

None

Version History

Introduced: 6.1

See Also

[IDLitComponent::Restore](#)

IDLitContainer

The IDLitContainer class is a specialization of the IDL_Container class designed to manage collections of IDLitComponent objects and provide methods to work with the iTools *identifier* system. IDLitContainer object methods allow you to add, retrieve, and remove values from a IDLitContainer-based hierarchy using either object references or iTool identifiers.

This class is written in the IDL language. Its source code can be found in the file `idlitcontainer__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDL_Container](#)

[IDLitComponent](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitContainer::Init](#)” on page 3138.

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLitContainer::Add](#)
- [IDLitContainer::AddByIdentifier](#)
- [IDLitContainer::Cleanup](#)
- [IDLitContainer::FindIdentifiers](#)
- [IDLitContainer::Get](#)

- [IDLitContainer::GetByIdentifier](#)
- [IDLitContainer::Init](#)
- [IDLitContainer::Remove](#)
- [IDLitContainer::RemoveByIdentifier](#)

In addition, this class inherits the methods of its superclasses.

Version History

Introduced: 6.0

IDLitContainer Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLitContainer::Add

The IDLitContainer::Add procedure method adds items to the container object.

Syntax

Obj->[IDLitContainer::]Add, *Components*[, /NO_NOTIFY]

Arguments

Components

An object reference (or array of object references) to the IDLitComponent object or objects to be added to the container. When this method is called, the IDENTIFIER properties of objects specified by *Components* are validated, verifying that no items in the container have the same value. If an object with the same identifier already exists in the container, a unique identifier is created by appending a number to the original value.

Keywords

NO_NOTIFY

Normally, when an item is added to a container object that inherits from both the IDLitContainer class and the IDLitIMessaging class, an ADDITEMS notification message is broadcast to all iTool components registered as monitoring this container. If this keyword is set, no message is sent.

Version History

Introduced: 6.0

IDLitContainer::AddByIdentifier

The IDLitContainer::AddByIdentifier procedure method adds an object to the container hierarchy in the position specified by the *Identifier* argument.

Syntax

Obj->[IDLitContainer::]AddByIdentifier, *Identifier*, *Item*

Arguments

Identifier

An object identifier specifying the location in the container hierarchy where *Item* should be added. *Identifier* can be either a fully-qualified object identifier (beginning with a “/” character) or a relative to the container on which this method is called. If *Identifier* is a null string (' '), *Item* is added to the root of the container object.

See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object identifier strings.

Item

An object reference to the iTool component object being added to the container.

Keywords

None

Version History

Introduced: 6.0

IDLitContainer::Cleanup

The IDLitContainer::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitContainer::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitContainer::FindIdentifiers

The IDLitContainer::FindIdentifiers function method allows you to retrieve the full identifiers for items within an iTool container.

Syntax

```
Result = Obj->IDLitContainer::FindIdentifiers([Pattern] [, COUNT=variable] [, /LEAF_NODES] )
```

Return Value

Returns a scalar string or string array containing the full object identifiers for all items matching the *Pattern*. If no object identifiers are matched, the function returns a scalar null string.

Arguments

Pattern

A string specifying a search pattern to be used for filtering the returned object identifiers. Only object identifiers that match *Pattern* are returned. Matching is case insensitive. If *Pattern* is not supplied, identifiers for all objects within the object on which this method is called are returned. See [STRMATCH](#) for a description of the rules used when matching patterns.

Note

Objects for which the PRIVATE property contains a True value are not included in the *Result*. Only objects that subclass from IDLitComponent can be marked as private.

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of identifiers in *Result*.

LEAF_NODES

Set this keyword to return only leaf nodes. Leaf nodes are objects which are either not a container, or are an empty container.

Note

If all of the objects within a container are marked as PRIVATE, then that container is considered a leaf node. Only objects that subclass from IDLitComponent can be marked as private.

Version History

Introduced: 6.1

See Also

[IDLitTool::FindIdentifiers](#)

IDLitContainer::Get

The IDLitContainer::Get function method retrieves items from the container.

Syntax

Result = Obj->[IDLitContainer::]Get([, /ALL] [, COUNT=variable] [, ISA=string or array of strings] [, POSITION=index or array of indices] [, /SKIP_PRIVATE])

Return Value

Returns an object reference or array of object references to the iTool components stored in the container. Unless either the ALL keyword or the POSITION keyword is specified, the first object in the container is returned. If no objects are found in the container, the return value is -1.

Arguments

None

Keywords

ALL

Set this keyword to return an array of object references to all of the objects in the container.

COUNT

Set this keyword equal to a named variable that will contain the number of objects selected by the function.

ISA

Set this keyword equal to a class name or vector of class names. This keyword is used in conjunction with the ALL keyword. The ISA keyword filters the array returned by the ALL keyword, returning only the objects that inherit from the class or classes specified by the ISA keyword.

Note

This keyword is ignored if the ALL keyword is not provided.

POSITION

Set this keyword equal to a scalar or array containing the zero-based indices of the positions of the objects to return.

SKIP_PRIVATE

Set this keyword to ignore any components for which the `PRIVATE` property is set to true. This keyword is ignored unless the `ALL` keyword is also set.

Version History

Introduced: 6.0

IDLitContainer::GetByIdentifier

The IDLitContainer::GetByIdentifier function method retrieves an object from a container hierarchy using the specified identifier to locate the object.

Syntax

Result = *Obj*->[IDLitContainer:]GetByIdentifier(*Identifier*)

Return Value

Returns an object reference to the object that was requested from the hierarchy, or a null object reference if no object was located.

Arguments

Identifier

The object identifier of the object that should be retrieved from the container hierarchy. *Identifier* can be either a fully-qualified object identifier (beginning with a “/” character) or a relative to the container on which this method is called.

See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object identifier strings.

Keywords

None

Version History

Introduced: 6.0

IDLitContainer::Init

The IDLitContainer::Init function method initializes the object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitContainer')
```

or

```
Result = Obj->[IDLitContainer::]Init() (Only in subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitContainer::Remove

The IDLitContainer::Remove procedure method removes items from the container.

Syntax

Obj->[IDLitContainer::]Remove, *Components*[, [/NO_NOTIFY](#)]

Arguments

Components

An object reference (or array of object references) to the objects to be removed from the container.

Keywords

NO_NOTIFY

Normally, when an item is removed from a container object that inherits from both the IDLitContainer class and the IDLitMessaging class, a REMOVEITEMS notification message is broadcast to all iTool components registered as monitoring this container. If this keyword is set, no message is sent.

Version History

Introduced: 6.0

IDLitContainer::RemoveByIdentifier

The `IDLitContainer::RemoveByIdentifier` function method removes an object from a container hierarchy using the specified identifier to locate the object.

Syntax

Result = *Obj*->[IDLitContainer:]RemoveByIdentifier(*Identifier*)

Return Value

Returns an object reference to the object that was removed from the hierarchy, or a null object reference if no object was removed.

Arguments

Identifier

The object identifier of the object that should be removed from the container hierarchy. The *Identifier* argument can be either a fully-qualified object identifier (beginning with a “/” character) or a relative to the container on which this method is called.

See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object identifier strings.

Keywords

None

Version History

Introduced: 6.0

IDLitData

The IDLitData class provides a generic data storage object that can hold data items of any IDL data type. The IDLitData class provides typing and data change notification functionality, and when coupled with the IDLitDataContainer object forms the base element for the construction of composite data types.

The IDLitData class implements the iTools *notifier interface*, which provides a mechanism by which observers of a data item can be alerted when the state of the information contained in the data object changes.

This class is written in the IDL language. Its source code can be found in the file `idlitdata__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitComponent](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitData::Init](#)” on page 3152.

Properties

Objects of this class have the following properties. See “[IDLitData Properties](#)” on page 3143 for details on individual properties.

- [HIDE](#)
- [NO_COPY](#)
- [READ_ONLY](#)
- [TYPE](#)

In addition, objects of this class inherit the properties of the superclass.

Methods

This class has the following methods:

- [IDLitData::AddDataObserver](#)
- [IDLitData::Cleanup](#)
- [IDLitData::Copy](#)
- [IDLitData::GetByType](#)
- [IDLitData::GetData](#)
- [IDLitData::GetProperty](#)
- [IDLitData::GetSize](#)
- [IDLitData::Init](#)
- [IDLitData::NotifyDataChange](#)
- [IDLitData::NotifyDataComplete](#)
- [IDLitData::RemoveDataObserver](#)
- [IDLitData::SetData](#)
- [IDLitData::SetProperty](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [Chapter 3, “Data Management”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitData Properties

IDLitData objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitData::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitData::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitData::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

HIDE

A Boolean value that controls the visibility of the data with respect to queries performed by the IDLitDataContainer methods GetData, SetData and GetIdentifiers. If the HIDE property has a True value, the data object is not found by these methods and the methods behave as if the data object did not exist.

Property Type	BOOLEAN		
Name String	HIDE		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

NO_COPY

If an initial value of the data is provided, this property will cause the movement of the data into the object without creating an additional copy, leaving the IDL variable named by the Data argument undefined.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

READ_ONLY

A Boolean value that controls the ability to modify the data with the `SetData` method. If the `READ_ONLY` property has a `True` value, an attempt to use `SetData` on the object fails. IDL prints an informational message in this case.

Property Type	BOOLEAN		
Name String	READ_ONLY		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TYPE

A scalar string containing the type of the data object. The default value of this property is `NULL (' ')` and cannot be changed after the class is instantiated. Subclasses of this class set the value of this property to a value that reflects the type of data that the subclass is implementing.

Property Type	STRING		
Name String	TYPE		
Get: Yes	Set: No	Init: Yes	Registered: Yes

IDLitData::AddDataObserver

The IDLitData::AddDataObserver procedure method specifies an object (the *Observer*) that will be notified when the contents of the data object are changed.

The object specified as the *Observer* must implement three methods that form the data observer interface:

- onDataChange
- onDataComplete
- onDelete

The onDataChange method is called when the contents of a data object are changed, and the onDataComplete method is called when the change messages are completed. This two-part notification process is used to allow observers of data to transact the change event and minimize the required system updates (such as graphic redrawing). The onDelete method is called when the data item is deleted.

Syntax

```
Obj->[IDLitData::]AddDataObserver, Observer
```

Arguments

Observer

An object reference to an iTool component object that implements the data observer interface methods. When the contained data value changes, the *Observer* is notified of this change.

Keywords

None

Version History

Introduced: 6.0

IDLitData::Cleanup

The IDLitData::Cleanup procedure method performs all cleanup operations on the object, and should be called by the Cleanup method of any subclass of this class. This method removes all data stored in the IDLitData object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitData::]Cleanup (*only in subclass' Cleanup method*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitData::Copy

The IDLitData::Copy function method returns an exact copy of the data object and its contents, including registered property values.

The following values are *not* copied:

- Values of unregistered properties.
- Values of properties registered after the data object is created.
- Property attribute values.

Note

If an object has been added to the data object, only the object reference to the added object is copied.

Syntax

Result = *Obj*->[IDLitData::]Copy()

Return Value

Returns a copy of the object on which it is called. If the copy operation fails, a null object is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitData::GetByType

The IDLitData::GetByType function method returns all contained objects of the specified iTool data type.

Syntax

Result = *Obj*->[IDLitData::]GetByType(*Type*[, **COUNT**=*variable*])

Return Value

Returns an object array containing references to the contained data objects that are of the iTool data type specified by *Type*. If no objects matching *Type* exist, a null object reference is returned.

Arguments

Type

A string containing the iTool data type to search for in the data hierarchy rooted by this data container.

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of valid objects returned by this method.

Version History

Introduced: 6.0

IDLitData::GetData

The IDLitData::GetData function method retrieves the data stored in the object.

Syntax

```
Result = Obj->[IDLitData::]GetData(Data[, Identifier] [, NAN=variable]  
[, /NO_COPY])
```

Return Value

Returns a 1 if the operation succeeds, and 0 if it fails.

Arguments

Data

A named variable that will contain the data retrieved from the data object. If the GetData method fails, the variable is not modified.

Identifier

A string argument that is not used by this method, but is accepted for parameter compatibility with the IDLitDataContainer::GetData method. If a value for this argument is supplied, the method does not retrieve any data and returns the value 0.

Keywords

NAN

Set this keyword to a named variable that will contain 1 if *Data* contains any non-finite values (either NaN or Infinity), or 0 otherwise.

NO_COPY

Set this keyword to move the data from the data object to the variable specified by the *Data* argument, leaving the data value of the data object undefined.

Version History

Introduced: 6.0

IDLitData::GetProperty

The IDLitData::GetProperty procedure method retrieves the value of an IDLitData property, and should be called by the GetProperty method of any subclass of this class.

Syntax

Obj->[IDLitData::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitData Properties](#)” on page 3143 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitData::GetSize

The IDLitData::GetSize function method returns an approximate value for the amount of memory being used by the data object.

Syntax

Result = *Obj*->[IDLitData:]GetSize()

Return Value

Returns the number of bytes of memory used by the data object. The returned value is an approximation.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitData::Init

The IDLitData::Init function method initializes the IDLitData object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitData'[, Data] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLitData::]Init([Data] [, PROPERTY=value]) (Only in subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Data

An IDL variable of any type that is stored in the data object.

Keywords

Any property listed under [“IDLitData Properties”](#) on page 3143 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitData::NotifyDataChange

The IDLitData::NotifyDataChange procedure method is called when a data object has been changed; it is part of the notification process that allows data updates to be reflected by visualizations that use the data. It works in conjunction with the NotifyDataComplete method to provide a two-pass change notification system that minimizes the number of operations performed when a data object changes.

When NotifyDataChange is called, the onDataChange methods of all observers of the data object (usually visualizations) are automatically called. This causes the observers to perform any necessary updates *without committing those updates*. When the NotifyDataComplete method is called, the observers' onDataComplete methods are called to commit the updates. This two-pass system allows visualizations to be notified that the data has changed without forcing an immediate redraw of the iTool window; when the NotifyDataComplete method is called, all visualizations can be redrawn at once.

Every call to the NotifyDataChange method must have an accompanying NotifyDataComplete call; without the call to NotifyDataComplete, no updates will take place.

Note

If the data object is being changed by an operation based on the IDLitDataOperation class, the call to NotifyDataChange is handled automatically. The only time you will need to call this method directly is if the operation that modifies the data is based on the more generic IDLitOperation class.

Syntax

```
Obj->[IDLitData::]NotifyDataChange
```

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitData::NotifyDataComplete

The IDLitData::NotifyDataComplete procedure method is called after a data object has been changed; it is part of the notification process that allows data updates to be reflected by visualizations that use the data. It works in conjunction with the NotifyDataChange method to provide a two-pass change notification system that minimizes the number of operations performed when a data object changes.

When NotifyDataChange is called, the onDataChange methods of all observers of the data object (usually visualizations) are automatically called. This causes the observers to perform any necessary updates *without committing those updates*. When the NotifyDataComplete method is called, the observers' onDataComplete methods are called to commit the updates. This two-pass system allows visualizations to be notified that the data has changed without forcing an immediate redraw of the iTool window; when the NotifyDataComplete method is called, all visualizations can be redrawn at once.

Every call to the NotifyDataChange method must have an accompanying NotifyDataComplete call; without the call to NotifyDataComplete, no updates will take place.

Note

If the data object is being changed by an operation based on the IDLitDataOperation class, the call to NotifyDataChange is handled automatically. The only time you will need to call this method directly is if the operation that modifies the data is based on the more generic IDLitOperation class.

Syntax

Obj->[IDLitData::]NotifyDataComplete

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitData::RemoveDataObserver

The IDLitData::RemoveDataObserver method unregisters an object that has been registered as an *observer* of this data object. After this method is called, the Observer will no longer be notified when the contents of the data object change. If the specified *Observer* doesn't exist, this method will return quietly.

Syntax

Obj->[IDLitData:]RemoveDataObserver, *Observer*

Arguments

Observer

An object reference to an iTool component object that was previously registered as an observer of this data object.

Keywords

None

Version History

Introduced: 6.0

IDLitData::SetData

The IDLitData::SetData function method copies the data from an IDL variable or expression into the data object, and notifies all its observers that the data has changed.

Syntax

```
Result = Obj->[IDLitData::]SetData(Data [, Identifier] [, /NO_COPY] [, /NULL])
```

Return Value

Returns a 1 if the operation succeeds, and 0 if it fails.

Arguments

Data

An IDL variable or expression whose value is copied into the data object.

Identifier

A string argument that is not used by this method. It is accepted by this method only for parameter compatibility with IDLitDataContainer::SetData. If a value for this argument is supplied, the method does not retrieve any data and returns the value 0.

Keywords

NO_COPY

Set this keyword to move the data into the specified IDLitData object without creating an additional copy. This leaves the original IDL variable named by the *Data* argument undefined.

NULL

Set this keyword to remove any data stored in the IDLitData object, leaving it empty. If this keyword is set, the *Data* argument is ignored.

Version History

Introduced: 6.0

IDLitData:: SetProperty

The IDLitData::SetProperty procedure method sets the value of an IDLitData property, and should be called by the SetProperty method of any subclass of this class.

Syntax

Obj->[IDLitData::]SetProperty[, *PROPERTY=**value*]

Arguments

None

Keywords

Any property listed under “[IDLitData Properties](#)” on page 3143 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitDataContainer

The IDLitDataContainer class is used to store collections of IDLitData and IDLitDataContainer objects to form hierarchical data structures. Objects can be added to and removed from an IDLitDataContainer at any time, allowing for the dynamic creation of composite data types.

Objects stored in an IDLitDataContainer hierarchy are referenced using iTool *object identifiers*. Object identifiers are simple scalar strings assigned to the IDENTIFIER property of an object when it is created. For a complete discussion of object identifiers and their role in the iTool system, see “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual.

This class is written in the IDL language. Its source code can be found in the file `idlitdatacontainer__define.pro` in the `lib/itool/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitContainer](#)

[IDLitData](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitDataContainer::Init](#)” on page 3167.

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLitDataContainer::Add](#)
- [IDLitDataContainer::Cleanup](#)

- [IDLitDataContainer::GetData](#)
- [IDLitDataContainer::GetIdentifiers](#)
- [IDLitDataContainer::GetProperty](#)
- [IDLitDataContainer::Init](#)
- [IDLitDataContainer::SetData](#)
- [IDLitDataContainer::SetProperty](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [Chapter 3, “Data Management”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitDataContainer Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLitDataContainer::Add

The IDLitDataContainer::Add procedure method adds items to the data container object, and sends out an onDataChange and onDataComplete messages to all observers of the data container.

Syntax

Obj->[IDLitDataContainer::]Add, *Data* [, /NO_NOTIFY]

Arguments

Data

An object reference (or array of object references) to the IDLitData object or objects to be added to the container. Objects that are not of class IDLitData are quietly ignored.

Keywords

NO_NOTIFY

Normally, when an item is added, an ADDITEMS notification message is broadcast to all iTool components registered as monitoring this container. If this keyword is set, no ADDITEMS message is sent.

Note

Regardless of the setting of NO_NOTIFY, the onDataChange and onDataComplete messages are always sent.

Version History

Introduced: 6.1

IDLitDataContainer::Cleanup

The IDLitDataContainer::Cleanup procedure method performs all cleanup operations on the object, and should be called by the Cleanup method of any subclass of this class. This method destroys all data objects stored in this container.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitDataContainer::]Cleanup (*only in subclass' Cleanup method*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitDataContainer::GetData

The IDLitDataContainer::GetData function method retrieves the data value contained in the data object specified by the *Identifier* argument.

Syntax

```
Result = Obj->[IDLitDataContainer::]GetData(Data [, Identifier] [, /NO_COPY])
```

Return Value

Returns a 1 if the operation succeeds, and 0 if it fails.

Arguments

Data

A named variable that will contain the data value of the specified data object.

Identifier

A string containing the relative object identifier path to the target data object for this method. See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object identifier strings.

If *Identifier* is not set, or is set to a null string, the data contained in the data container itself is returned.

Keywords

NO_COPY

Set this keyword to move the data from the data container object to the variable specified by the *Data* argument, leaving the data value of the data object undefined.

Version History

Introduced: 6.0

IDLitDataContainer::GetIdentifiers

The IDLitDataContainer::GetIdentifiers function method retrieves the object identifiers for all data and data container objects contained in the data container object.

Syntax

Result = *Obj*->[IDLitDataContainer::]GetIdentifiers([*Pattern*] [, /[LEAF](#)])

Return Value

Returns an array of strings containing the object identifiers for all data and data container objects contained by the data container object. If the container is empty, or if the *Pattern* argument is supplied and no object identifiers are matched, the function returns a scalar null string.

Arguments

Pattern

A string specifying a search pattern to be used for filtering the returned object identifiers. Only object identifiers that match *Pattern* are returned. See [STRMATCH](#) for a description of the rules used when matching patterns.

Keywords

LEAF

Set this keyword to return only object identifiers whose final node is *not* a container object.

Version History

Introduced: 6.0

IDLitDataContainer::GetProperty

The IDLitDataContainer::GetProperty procedure method retrieves the value of an IDLitDataContainer property, and should be called by the GetProperty method of any subclass of this class.

Syntax

Obj->[IDLitDataContainer::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitDataContainer Properties](#)” on page 3161 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitDataContainer::Init

The IDLitDataContainer::Init function method initializes the IDLitDataContainer object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitDataContainer'[, Data] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLitDataContainer::]Init([Data] [, PROPERTY=value])  
(Only in subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Data

An IDL variable of any type or shape that is stored into the object at initialization.

Keywords

Any property listed under [“IDLitDataContainer Properties”](#) on page 3161 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitDataContainer::SetData

The IDLitDataContainer::SetData function method stores data in the IDLitData object specified by *Identifier*. This method acts in exactly the same way as the IDLitData::SetData method.

Syntax

```
Result = Obj->[IDLitDataContainer::]SetData(Data, Identifier[, /NO_COPY]  
[, /NULL])
```

Return Value

Returns a 1 if the operation succeeds, or 0 if it fails.

Arguments

Data

An IDL variable of any type that is copied into the specified data object.

Identifier

A scalar string containing the object identifier of the IDLitData object in which the data specified by *Data* should be stored. This path must specify a valid IDLitData object, or the function fails. See “iTool Object Identifiers” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object identifier strings.

Keywords

NO_COPY

Set this keyword to move the data into the specified IDLitData object without creating an additional copy. This leaves the original IDL variable named by the *Data* argument undefined.

NULL

Set this keyword to remove any data stored in the IDLitData object, leaving it empty. If this keyword is set, the *Data* argument is ignored, but must still be present.

Version History

Introduced: 6.0

IDLitDataContainer:: SetProperty

The IDLitDataContainer::SetProperty procedure method sets the value of an IDLitDataContainer property, and should be called by the SetProperty method of any subclass of this class.

Syntax

Obj->[IDLitDataContainer::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLitDataContainer Properties](#)” on page 3161 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitDataOperation

The IDLitDataOperation class is a subclass of IDLitOperation that automates data access and automatically records information for the undo-redo system. By automating this functionality, the IDLitDataOperation class eliminates much of the work required to implement a standard subclass of the IDLitOperation class. See [IDLitOperation](#) for additional information on iTool operations.

Note

The IDLitDataOperation class is designed for use in operations that affect data values. If you are creating an operation that acts on items other than the data that underlies a visualization, you should base your operation class on the IDLitOperation class.

While the implementation of a standard subclass of an IDLitOperation class requires the developer to provide implementations of the DoAction, UndoOperation, and RedoOperation methods, the IDLitDataOperation class provides these methods automatically. The only method the developer of an IDLitDataOperation class is *required* to implement is the Execute method, which contains the logic for the specific operation being performed. Depending on the operation being performed, the developer of an IDLitDataOperation may also implement methods to undo the operation, display a user interface before performing the operation, get and set properties specific to the operation, and initialize the IDLitDataOperation object.

When an IDLitDataOperation is requested by a user, the following things occur:

1. As with any operation, execution commences when the DoAction method is called on this object. When called, the IDLitDataOperation retrieves the currently-selected items. If nothing is selected, the operation returns.
2. For each selected item, the data objects of the parameters registered as “operation targets” are retrieved.
3. The data objects are queried for iTool data types that match the types supported by the IDLitDataOperation.

For each data object that includes data of an iTool data type supported by the IDLitDataOperation, the following things occur:

1. The data from the data object is retrieved.
2. If the IDLitDataOperation does not have the REVERSIBLE_OPERATION property set, a copy of the data is created and placed into the undo-redo command set.

3. The Execute method is called, with the retrieved data as its argument.
4. If the Execute method succeeds and the IDLitDataOperation has the EXPENSIVE_OPERATION property set, a copy of the results is placed into the undo-redo command set.
5. The result of the IDLitDataOperation is placed in the data object. This action will cause all visualization items that use the data object to update when the operation is completed.

Once all selected data items have been processed, the undo-redo command set is placed into the system undo-redo buffer for later use.

For a detailed discussion of both the IDLitOperation and IDLitDataOperation classes, see [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual.

This class is written in the IDL language. Its source code can be found in the file `idlitdataoperation__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitOperation](#)

Note

See the [“iTools Object Model Diagram”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See [“IDLitDataOperation::Init”](#) on page 3182.

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLitDataOperation::Cleanup](#)
- [IDLitDataOperation::DoExecuteUI](#)

- [IDLitDataOperation::Execute](#)
- [IDLitDataOperation::GetProperty](#)
- [IDLitDataOperation::Init](#)
- [IDLitDataOperation::SetProperty](#)
- [IDLitDataOperation::UndoExecute](#)

In addition, this class inherits the methods of its superclasses (if any).

Examples

See “[Creating a New Data-Centric Operation](#)” in Appendix of the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

See Also

[IDLitOperation](#)

IDLitDataOperation Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

IDLitDataOperation::Cleanup

The IDLitDataOperation::Cleanup procedure method performs all cleanup on the object.

Note

An operation based on the IDLitDataOperation class need not implement this method if the operation does not allocate any pointers or object references of its own.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitOperation::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitDataOperation::DoExecuteUI

The `IDLitDataOperation::DoExecuteUI` function method provides a way for the iTool developer to request user input before performing an operation. If the `SHOW_EXECUTION_UI` property is set, this method will be called before the operation's `Execute` method is called.

Note

Every operation based directly on the `IDLitDataOperation` class that requires user interaction before execution must implement its own `DoExecuteUI` method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer's Guide* manual for details.

The `DoExecuteUI` method itself can present any interface. In practice, most iTool `DoExecuteUI` methods use the `IDLitTool::DoUIService` method to display a user interface requesting user input. User interfaces and UI Services are discussed in [Chapter 11, “iTool User Interface Architecture”](#) in the *iTool Developer's Guide* manual.

Warning

If the `DoExecuteUI` method does not return success, the `Execute` method will not be called.

Syntax

```
Result = Obj->[IDLitDataOperation:]DoExecuteUI()
```

Return Value

Returns a 1 if the displayed user interface code executed successfully, and the operation should be executed. If user interface code does not execute successfully, or if the user cancels the operation, this function returns a 0.

Arguments

None

Keywords

None

Example

The following is an example of a DoExecuteUI method, taken from the iTools “Scale Factor” operation:

```
FUNCTION IDLitopScalefactor::DoExecuteUI

oTool = self->GetTool()
IF (oTool EQ OBJ_NEW()) THEN RETURN, 0

RETURN, oTool->DoUIService('ScaleFactor', self)

END
```

This implementation of the DoExecuteUI method does the following:

1. Retrieves an object reference to the current iTool using the GetTool method.
2. Checks to make sure the returned object reference is not a null object (which would be the case if the GetTool method failed).
3. Calls the ScaleFactor UI Service.

Version History

Introduced: 6.0

IDLitDataOperation::Execute

The IDLitDataOperation::Execute function method contains the execution logic for the operation. This method is called automatically when the iTool user requests an operation based on the IDLitDataOperation class.

Note

Every operation based directly on the IDLitDataOperation class must implement its own Execute method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

When the iTool system executes an IDLitDataOperation, it passes the *raw data* (of the appropriate iTools data type) from the selected objects to the Execute method. This means that the Execute method itself does not need to “unpack” a data object before performing the operations, allowing rapid and simple operation execution. For example, if the operation expects data of the iTools data type IDLARRAY2D, the iTool system will include the selected two-dimensional array as the *Data* argument.

Syntax

Result = *Obj*->[IDLitDataOperation:]Execute(*Data*)

Return Value

The return value is 1 if the operation executed successfully, or 0 otherwise.

Arguments

Data

A single data item on which the operation should be performed. Note that *Data* is *not* an IDLitData object, but actual data.

Keywords

None

Version History

Introduced: 6.0

See Also

[IDLitDataOperation::UndoExecute](#)

IDLitDataOperation::GetProperty

The IDLitDataOperation::GetProperty procedure method retrieves the value of a property or group of properties of an operation object.

Note

An operation based on the IDLitDataOperation class *must* implement this method if the operation defines one or more properties not inherited from the superclass. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Syntax

```
Obj->[IDLitDataOperation::]GetProperty[, PROPERTY=variable]
```

Arguments

None

Keywords

Any property listed under [“IDLitDataOperation Properties”](#) on page 3175 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Example

To retrieve the value of the REVERSIBLE_OPERATION property:

```
Obj->IDLitDataOperation::GetProperty, REVERSIBLE_OPERATION = rev
```

Version History

Introduced: 6.0

IDLitDataOperation::Init

The IDLitDataOperation::Init function method initializes the IDLitDataOperation object and sets properties that define the behavior the operation provides.

Note

An operation based on the IDLitDataOperation class *must* implement this method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitDataOperation' [, PROPERTY=variable] )
```

or

```
Result = Obj->[IDLitDataOperation::]Init([, PROPERTY=variable])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLitDataOperation Properties](#)” on page 3175 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal the appropriate property value.

Version History

Introduced: 6.0

IDLitDataOperation:: SetProperty

The IDLitDataOperation::SetProperty procedure method sets the value of a property or group of properties for the operation.

Note

An operation based on the IDLitDataOperation class *must* implement this method if the operation defines one or more properties not inherited from the superclass. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Syntax

Obj->[IDLitDataOperation::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under [“IDLitDataOperation Properties”](#) on page 3175 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitDataOperation::UndoExecute

The IDLitDataOperation::UndoExecute function method is called when a user selects the Undo operation after executing an IDLitDataOperation that sets the value of the [REVERSIBLE_OPERATION](#) property to 1. (If the IDLitDataOperation does not set this property, the UndoExecute method is not called.) If implemented, this method generally applies the inverse of the operation performed by the Execute method.

Note

An operation based on the IDLitDataOperation class *must* implement this method if the operation sets the [REVERSIBLE_OPERATION](#) property. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

When the iTool system undoes an IDLitDataOperation, it passes the *raw data* (of the appropriate iTools data type) from the selected objects to the UndoExecute method. This means that the UndoExecute method itself does not need to “unpack” a data object before performing the operations, allowing rapid and simple operation execution. For example, if the operation expects data of the iTools data type IDLARRAY2D, the iTool system will include the selected two-dimensional array as the *Data* argument.

Syntax

```
Result = Obj->[IDLitDataOperation:]UndoExecute(Data)
```

Return Value

The return value is 1 if the operation was undone successfully, or 0 otherwise.

Arguments

Data

A single data item on which the operation should be performed. Note that *Data* is *not* an IDLitData object, but actual data.

Keywords

None

Version History

Introduced: 6.0

See Also

[IDLitDataOperation::Execute](#)

IDLitMessaging

The IDLitMessaging interface class provides a set of methods used to allow an object that implements the interface to send notification messages and to react appropriately when notification messages are received from other iTool component objects. This class also provides methods that allow the iTool developer to notify the iTool user of error conditions via graphical dialogs, to prompt the user for input, and to retrieve a reference to the current iTool object.

IDLitMessaging objects are not intended to be created as standalone entities; rather, this class should be included as the superclass of another iTool class.

Note

In the iTools system, management of messages is handled by the IDLitVisualization class, of which the IDLitMessaging class is a superclass. In practice, this means that if you need to override any methods of the IDLitMessaging class, you will do so in the definition of your visualization class.

This class is written in the IDL language. Its source code can be found in the file `idlitmessaging__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

This class has no superclasses.

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

Objects of this class are not created as standalone objects.

Properties

Objects of this class have no properties of their own.

Methods

This class has the following methods:

- [IDLitIMessaging::AddOnNotifyObserver](#)
- [IDLitIMessaging::DoOnNotify](#)
- [IDLitIMessaging::ErrorMessage](#)
- [IDLitIMessaging::GetTool](#)
- [IDLitIMessaging::ProbeStatusMessage](#)
- [IDLitIMessaging::ProgressBar](#)
- [IDLitIMessaging::PromptUserText](#)
- [IDLitIMessaging::PromptUserYesNo](#)
- [IDLitIMessaging::RemoveOnNotifyObserver](#)
- [IDLitIMessaging::SignalError](#)
- [IDLitIMessaging::StatusMessage](#)

Version History

Introduced: 6.0

IDLitMessaging Properties

Objects of this class have no properties of their own.

IDLitMessaging::AddOnNotifyObserver

The `IDLitMessaging::AddOnNotifyObserver` procedure method is used to register a specified iTool component object to receive messages generated by the `DoOnNotify` method of another specified iTool component object.

Syntax

Obj->[IDLitMessaging::]AddOnNotifyObserver, *IdObserver*, *IdSubject*

Arguments

IdObserver

The object identifier of an iTool component object that is the *observer* expressing interest in the subject specified by *IdSubject*. Often, *IdObserver* is the object identifier of the object on which this method is being called.

The object specified by *IdObserver* must implement the `OnNotify` callback method, which is called when a notification message is dispatched to *IdObserver* by the `DoOnNotify` method of another iTool component object (usually the object specified by *IdSubject*). The `OnNotify` method has the following signature:

```
PRO ::OnNotify, idOriginator, idMessage, message
```

where

- *idOriginator* is the object identifier of the iTool component that is the source of the message
- *idMessage* is a string that identifies the type of message being sent
- *message* is the message itself.

In general, the *idMessage* string is used by the `OnNotify` method to determine what type of action to take. See “[IDLitMessaging::DoOnNotify](#)” on page 3192 for additional details.

IdSubject

A string value identifying the item that *IdObserver* is interested in. This is normally the object identifier of a particular iTool component object, but it can be any string value. When a message sent via `IDLitMessaging::DoOnNotify` specifies *IdSubject* as the originator, the *IdObserver* object’s `OnNotify` method is called.

Keywords

None

Version History

Introduced: 6.0

IDLitMessaging::DoOnNotify

The `IDLitMessaging::DoOnNotify` procedure method is used to broadcast a notification message to iTool component objects that are observing the source of the message.

See the “[IDLitMessaging::AddOnNotifyObserver](#)” on page 3190, and “[IDLitMessaging::RemoveOnNotifyObserver](#)” on page 3202 methods for more information on notification messages.

See “[iTool Messaging System](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a more in-depth discussion of notification.

Syntax

```
Obj->[IDLitMessaging::]DoOnNotify, IdOriginator, IdMessage, Value
```

Arguments

IdOriginator

A string value identifying the item that is the source of the message. Normally, *IdOriginator* is the object identifier of the object that calls this method, but it can be any string value.

IdMessage

A string that will uniquely identify the message being sent. The value of *IdMessage* is determined by the iTool developer; it is used by the `OnNotify` methods of any observer objects to determine what type of message has been received, and to act accordingly. For example, if a property value has changed, this argument might be set to the string `SETPROPERTY`.

Value

Any value that is associated with the message being sent. For example, if a `SETPROPERTY` message was sent, *Value* might contain the name of the property changed. If no value is associated with the message, set this argument to a null string (`''`).

Keywords

None

Version History

Introduced: 6.0

IDLitMessaging::ErrorMessage

The IDLitMessaging::ErrorMessage procedure method is used to display an error message to the user.

The actual method used to display the message to the user depends on the user interface in use. Normally, this method will display a modal dialog.

Syntax

```
Obj->[IDLitMessaging::]ErrorMessage, StrMessage [, SEVERITY=integer]  
[, TITLE=string] [, /USE_LAST_ERROR]
```

Arguments

StrMessage

A scalar string or string array containing a description of the error.

Keywords

SEVERITY

Set this keyword to the severity code to use for this message. The system recognizes the following values:

- 0 = Informational
- 1 = Warning
- 2 = Error

TITLE

Set this keyword to the title that should be displayed as part of the prompt. In most cases, this value is placed in the title bar of the modal dialog.

USE_LAST_ERROR

If this keyword is set, the system will use the information that was part of the last error signaled to the iTool. The last error may have been set either by a previous call to this method or by a call to the [IDLitMessaging::SignalError](#) method.

Version History

Introduced: 6.0

IDLitIMessaging::GetTool

The IDLitIMessaging::GetTool function method returns an object reference to the iTool object associated with the object on which it is called.

Syntax

Result = *Obj*->[IDLitIMessaging::]GetTool()

Return Value

Returns an object reference to the iTool object associated with the object on which the GetTool is called. If the object is not contained by an iTool, this function returns a null object reference.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitMessaging::ProbeStatusMessage

The IDLitMessaging::ProbeStatusMessage procedure method is used to display a status message to the user, which is displayed in a data-specific region of the user interface. Unlike other messages that require immediate acknowledgement by the user, a status message is passive and no response is needed.

The actual method used to display the value to the user depends on the user interface in use. In a standard iTool included with IDL, this status message is displayed in the lower right corner of the iTool window.

Syntax

Obj->[IDLitMessaging::]ProbeStatusMessage, *StrMessage*

Arguments

StrMessage

A scalar string that is displayed in the data-specific status area.

Keywords

None

Version History

Introduced: 6.0

IDLitIMessaging::ProgressBar

The IDLitIMessaging::ProgressBar function method is used to display a progress bar to the user and update the displayed values.

On the initial call to this method, a dialog is displayed. The dialog remains active until the method is called with the /SHUTDOWN keyword.

The actual method used to display the message to the user depends on the user interface in use. Normally, the progress bar is displayed as a modal dialog.

Syntax

```
Result = Obj->[IDLitIMessaging::]ProgressBar(StrMessage [, CANCEL=string]
[, PERCENT=value] [, /SHUTDOWN])
```

Return Value

Returns 1 if the progress bar was displayed successfully, or 0 otherwise.

Arguments

StrMessage

A scalar string or string array containing a description of the action taking place.

Keywords

CANCEL

Set this keyword to a string containing the text for the Cancel button on the Progress Bar. If this keyword is not specified, or is set to a null string, then the string “Cancel” is used.

PERCENT

Set this keyword to a value between 0 and 100 that represents the “percentage complete” value to be shown on the progress bar.

SHUTDOWN

If set, any progress bar that is being displayed is removed from the user interface.

Version History

Introduced: 6.0

CANCEL keyword added: 6.1

IDLitIMessaging::PromptUserText

The IDLitIMessaging::PromptUserText function method is used to prompt the iTool user with a question and retrieve a text answer. The text answer is returned to the caller of this method via a supplied IDL variable.

The actual method used to prompt the user depends on the user interface in use. Normally, the prompt is displayed as a modal dialog.

Syntax

```
Result = Obj->[IDLitIMessaging::PromptUserText(StrPrompt, Answer  
[, TITLE=string)
```

Return Value

Returns 1 if the prompt was displayed and the user clicked the **OK** button. If the user clicks the **Cancel** button or dismisses the dialog, the return value is 0.

Arguments

StrPrompt

A scalar string or string array containing a prompt for the user to respond to.

Answer

A named variable that will contain the string value entered by the user.

Keywords

TITLE

Set this keyword to a string representing the title of the prompt. This value is placed in the title bar of the modal dialog.

Version History

Introduced: 6.0

IDLitMessaging::PromptUserYesNo

The IDLitMessaging::PromptUserYesNo function method is used to prompt the user with a yes or no question and return an answer.

The actual method used to prompt the user depends on the user interface in use. Normally, the prompt is displayed as a modal dialog.

Syntax

```
Result = Obj->[IDLitMessaging::PromptUserYesNo(StrPrompt, Answer  
[, TITLE=string])
```

Return Value

Returns 1 if the dialog executed properly and returned a value in the variable specified. You must check the value stored in the variable specified as the *Answer* argument to determine which button the user pressed.

Arguments

StrPrompt

A scalar string or string array containing a prompt for the user to respond to.

Answer

A named variable that will contain 1 if the user answered Yes to the prompt and 0 if the user answered No.

Keywords

TITLE

Set this keyword to the title that should be displayed as part of the prompt. In most cases, this value is placed in the title bar of the modal dialog.

Version History

Introduced: 6.0

IDLitMessaging::RemoveOnNotifyObserver

The `IDLitMessaging::RemoveOnNotifyObserver` procedure method is used to unregister a specified iTool component object as wishing to receive messages generated by the `DoOnNotify` method of another specified iTool component object. This method reverses the action of calling the [IDLitMessaging::AddOnNotifyObserver](#) method.

Syntax

Obj->[IDLitMessaging:]RemoveOnNotifyObserver, *IdObserver*, *IdSubject*

Arguments

IdObserver

The object identifier of an iTool component object that is currently registered as an observer of the component specified by *IdSubject*. Often, this is the object identifier of the object on which method is being called.

IdSubject

The object identifier of the iTool component object that *IdObserver* is currently registered as observing. This is normally the object identifier of a particular iTool component object, but it can be any scalar string.

Keywords

None

Version History

Introduced: 6.0

IDLitMessaging::SignalError

The IDLitMessaging::SignalError procedure method is used to signal an error in the system. No user interface is displayed to the user, but the error is registered with the system, made the current error, and recorded by the iTool log service (if enabled).

Note

If this method is used to signal an error, the [IDLitMessaging::ErrorMessage](#) method can be called with the /USE_LAST_ERROR keyword to use the information set by this method.

Syntax

```
Obj->[IDLitMessaging::]SignalError, StrMessage [, CODE=integer]
      [, SEVERITY=integer]
```

Arguments

StrMessage

A scalar string or string array containing a description of the error.

Keywords

CODE

Set this keyword to an integer that represents an error code that can be associated with the error. Error codes can be any integer value specified by the iTool developer.

SEVERITY

Set this keyword to the severity code to use for this message. The system recognizes the following values:

- 0 = Informational
- 1 = Warning
- 2 = Error

Version History

Introduced: 6.0

IDLitIMessaging::StatusMessage

The IDLitIMessaging::StatusMessage procedure method is used to display a status message to the user. Unlike other messages that require immediate acknowledgement by the user, a status message is passive and no response is needed.

The actual method used to display the value to the user depends on the user interface in use. In a standard iTool included with IDL, this status message is displayed in the lower left corner of the iTool window.

Syntax

```
Obj->[IDLitIMessaging::]StatusMessage, StrMessage  
[, SEGMENT_IDENTIFIER=string]
```

Arguments

StrMessage

A scalar string that is displayed as a status message to the user.

Keywords

SEGMENT_IDENTIFIER

Set this keyword to a string containing the relative identifier of the status bar segment in which the message is to be displayed. By default, iTools have two status bar segments, as described in “[Status Messages](#)” in Chapter 12 of the *iTool Developer’s Guide* manual. By default, the StatusMessage method changes the value of the Status segment; that segment’s identifier is 'MESSAGE'.

To change the value of the Probe Status segment, use the segment identifier 'PROBE'. Alternately, you can use the [IDLitIMessaging::ProbeStatusMessage](#) method.

Version History

Introduced: 6.0

SEGMENT_IDENTIFIER keyword added: 6.1

IDLitManipulator

The IDLitManipulator class provides the base functionality of the iTools manipulator system. Most automation and functional capabilities of the manipulator system are provided by this class. Tasks performed by this class include:

- Selection operations and associated selection state management.
- Automated mouse cursor management (changing the cursor as the mouse traverses different items in the current visualization).
- System event settings management.
- Coordination between the manipulator and manipulator visualization system.

This class is written in the IDL language. Its source code can be found in the file `idlitmanipulator__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Note

See [Chapter 8, “Creating a Manipulator”](#) in the *iTool Developer’s Guide* manual for information on creating a custom manipulator

Superclasses

[IDLitComponent](#)

[IDLitMessaging](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitManipulator::Init](#)” on page 3219.

Properties

Objects of this class have the following properties. See “[IDLitManipulator Properties](#)” on page 3208 for details on individual properties.

- [BUTTON_EVENTS](#)

- `DEFAULT_CURSOR`
- `DESCRIPTION`
- `DISABLE`
- `KEYBOARD_EVENTS`
- `MOTION_EVENTS`
- `OPERATION_IDENTIFIER`
- `PARAMETER_IDENTIFIER`
- `TRANSIENT_DEFAULT`
- `TRANSIENT_MOTION`
- `TYPES`
- `VIEWS_ONLY`
- `VISUAL_TYPE`

In addition, objects of this class inherit the properties of the superclass of this class.

Methods

This class has the following methods:

- `IDLitManipulator::Cleanup`
- `IDLitManipulator::CommitUndoValues`
- `IDLitManipulator::GetCursorType`
- `IDLitManipulator::GetProperty`
- `IDLitManipulator::Init`
- `IDLitManipulator::OnKeyboard`
- `IDLitManipulator::OnLoseCurrentManipulator`
- `IDLitManipulator::OnMouseDown`
- `IDLitManipulator::OnMouseMove`
- `IDLitManipulator::OnMouseUp`
- `IDLitManipulator::RecordUndoValues`
- `IDLitManipulator::RegisterCursor`
- `IDLitManipulator::SetCurrentManipulator`

- [IDLitManipulator::SetProperty](#)

In addition, this class inherits the methods of its superclass.

Examples

See [Chapter 8, “Creating a Manipulator”](#) in the *iTool Developer’s Guide* manual for examples.

Version History

Introduced: 6.0

IDLitManipulator Properties

IDLitManipulator objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitManipulator::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitManipulator::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitManipulator::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

BUTTON_EVENTS

A boolean value controlling whether mouse button events are generated. If set, mouse button up and button down events will be dispatched to this manipulator. These events are enabled by default.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEFAULT_CURSOR

A string containing the name of the default cursor for the manipulator. This is the name of the cursor that is displayed if no items are hit during mouse cursor motion. By default, the default cursor is set to 'ARROW'. Allowable values are:

- ARROW
- CROSSHAIR
- ICON
- IBEAM
- MOVE
- ORIGINAL
- SIZE_NE
- SIZE_NW
- SIZE_SE
- SIZE_SW
- SIZE_NS
- SIZE_EW
- UP_ARROW

Property Type	String		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

DESCRIPTION

A string representing the full name or description of this object.

Property Type	STRING		
Name String	Description		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DISABLE

A boolean value indicating that the manipulator should be disabled. Disabling a manipulator prevents it from being chosen by a user or by the iTool system.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

KEYBOARD_EVENTS

A boolean value controlling whether keyboard events are generated. If set, keyboard events will be dispatched to the manipulator. These events are enabled by default.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MOTION_EVENTS

A boolean value controlling whether motion events are generated. If set, motion events will be dispatched to this manipulator. These events are enabled by default.

Property Type	Boolean		
----------------------	---------	--	--

Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

OPERATION_IDENTIFIER

A string identifier for the operation that is associated with the manipulator. This identifier specifies the operation that is used by the undo-redo system to transact the actions performed by this manipulator. If not specified, the manipulator actions are not recorded by the undo-redo system.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Note

See [“Manipulators and the Undo/Redo System”](#) in Chapter 8 of the *iTool Developer’s Guide* manual for information on configuring a custom manipulator to support undo/redo operations.

PARAMETER_IDENTIFIER

A string identifier specifying which parameter the operation should query when transacting the manipulator. If specified, this identifier is used in conjunction with the operation specified by the OPERATION_IDENTIFIER property to transact the operations performed by a manipulator. The parameter is primarily used to specify the property identifier for manipulators that change the values of properties.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Note

See [“Retrieving Property Information”](#) in Appendix A of the *iTool Developer’s Guide* manual for more information about property identifiers and names. See [“Manipulators and the Undo/Redo System”](#) in Chapter 8 of the *iTool Developer’s Guide* manual for information on configuring a custom manipulator to support undo/redo operations.

TRANSIENT_DEFAULT

An integer value used to indicate that the manipulator is *transient*, which causes the system to automatically switch to the default manipulator when this manipulator's mouse actions are complete (that is, when the `OnMouseUp` method has been called) or when an end-of-keyboard-entry character (carriage return or escape key) is detected.

This property can be set to any of the following values:

- 0 = No transient behavior
- 1 = Mouse transient behavior is enabled. When a mouse button up is detected, the manipulation system will switch to the default manipulator for the system.
- 2 = Keyboard transient behavior is enabled. When a carriage return or escape key is pressed, the system will switch to the default manipulator for the system.
- 3 = Both keyboard and mouse transient behavior are enabled.

Property Type	Enumerated List		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TRANSIENT_MOTION

A boolean value controlling whether transient motion events are generated. If set, the manipulator will generate motion events beginning when the `OnMouseDown` method is called and ending when the `OnMouseUp` method is called.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TYPES

A string array specifying the visualization types that this manipulator can operate on. If not specified, the manipulator will support all registered visualization types.

Allowable TYPES values are those returned by the `IDLitVisualization::GetTypes` method.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

VIEWS_ONLY

A boolean value controlling the content of the manipulator hierarchy. If set, the manipulator will only operate on views in the manipulator hierarchy.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

VISUAL_TYPE

A string representing the type of selection visual to use with this manipulator. The VISUAL_TYPE property should match the corresponding property of an IDLitManipulatorVisual object. This high-level abstraction allows similar selection visuals to be assigned across multiple manipulators.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLitManipulator::Cleanup

The IDLitManipulator::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitManipulator::]Cleanup() (*In a subclass' Init method only.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::CommitUndoValues

The IDLitManipulator::CommitUndoValues function method is used to complete a transaction that is occurring as a result of the manipulator interaction. This method works in conjunction with the RecordUndoValues method.

When a manipulator interaction is complete, the manipulator should call this method, which in turn calls the RecordFinalValues method of the operation associated with the manipulator. The RecordFinalValues method records any information needed to undo the operation and places the information in the iTool's undo-redo transaction buffer.

For interactive manipulators, the CommitUndoValues method should be called by the OnMouseUp method.

Note

This method must be called on the current manipulator before being called on a superclass to ensure that the current selection state maintained by the manipulator is valid.

Syntax

Result = *Obj*->[IDLitManipulator::]CommitUndoValues([UNCOMMIT])

Return Value

Returns 1 if the commit was successful, or 0 otherwise.

Arguments

None

Keywords

UNCOMMIT

Set this keyword to discard any pending information from a previous call to RecordUndoValues. The interaction is not recorded in the iTool's undo-redo transaction buffer.

Version History

Introduced: 6.0

IDLitManipulator::GetCursorType

The IDLitManipulator::GetCursorType function method retrieves the name of the cursor to display for the manipulator. The name of the cursor can be used with the SetCurrentCursor method of the IDLitWindow component.

The system will use this method to automatically change the cursor when the mouse traverses the contents of a visualization window. As the mouse transverses the window, the name for the hit item that is part of the current selection group is retrieved and passed to this routine. This action gives the manipulator the ability to change cursors depending on the area of the visualization the cursor is over.

Syntax

Result = *Obj*->[IDLitManipulator::]GetCursorType(*TypeIn*, *KeyMods*)

Return Value

Returns a string containing the name of the cursor. If nothing is matched, an empty string is returned.

Note

This name is defined during cursor registration using the [IDLitManipulator::RegisterCursor](#) method.

Arguments

TypeIn

A string variable containing the name of the portion of the selection visual for which the cursor is being requested.

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values

1 = SHIFT Key

2 = CTRL Key

4 = CAPS LOCK key

8 = ALT key

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::GetProperty

The IDLitManipulator::GetProperty procedure method retrieves the value of an IDLitManipulator property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclass.

Syntax

Obj->[IDLitManipulator:]GetProperty[, *PROPERTY*=*variable*]

Arguments

None

Keywords

Any property listed under [“IDLitManipulator Properties”](#) on page 3208 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitManipulator::Init

The IDLitManipulator::Init function method initializes the IDLitManipulator object, and should be called by the subclass' Init method. This method also calls the superclass' Init method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitManipulator'[, PROPERTY=value])
```

or

```
Result = Obj->[IDLitManipulator::]Init([, PROPERTY=value])  
(In a subclass' Init method only.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under [“IDLitManipulator Properties”](#) on page 3208 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitManipulator::OnKeyboard

The IDLitManipulator::OnKeyboard procedure method is used when a keyboard event occurs on the target IDLitWindow.

Syntax

Obj->[IDLitManipulator:]OnKeyboard, *Win, IsASCII, Character, KeyValue, X, Y, Press, Release, KeyMods*

Arguments

Win

An object reference for the IDLitWindow from which the event came.

IsASCII

A Boolean value to determine if the character is in the ASCII character set. This value is True if the character is in the ASCII character set.

KeyValue

The numeric value representing the key pressed if the key is non-ASCII. The possible numeric values are:

- 1 = Shift
- 2 = Control
- 3 = Caps Lock
- 4 = Alt
- 5 = Left
- 6 = Right
- 7 = Up
- 8 = Down
- 9 = Page Up
- 10 = Page Down
- 11 = Home
- 12 = End

Character

A string containing the ASCII character of the key pressed.

X

A floating-point value representing the *x*-coordinate in the window's coordinates.

Y

A floating-point value representing the *y*-coordinate in the window's coordinates.

Press

A Boolean value to determine if the current event is a key press. This value is True if the event is pressing the key

Release

A Boolean value to determine if the current event is a key release. This value is True if event is releasing the key

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values:

- 1 = SHIFT key
- 2 = CTRL key
- 4 = CAPS LOCK key
- 8 = ALT key

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::OnLoseCurrentManipulator

The IDLitManipulator::OnLoseCurrentManipulator procedure method is used when this manipulator is no longer the current manipulator in the system.

Note

This method is designed to be implemented by a subclass of IDLitManipulator, to perform any cleanup work necessary when a new manipulator is selected.

Syntax

Obj->[IDLitManipulator::]OnLoseCurrentManipulator

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::OnMouseDown

The IDLitManipulator::OnMouseDown procedure method is used when a mouse down event occurs on the target IDLitWindow. When applied, this method will perform a select operation on the current contents of the target IDLitWindow.

When overriding this method in a sub-class, this method is normally the first task performed so that the instance data of the object is properly setup.

If the manipulator is in the middle of a mouse-down, mouse-up transaction, the following instance data will be valid:

- ButtonPress — Set to the mouse buttons pressed.
- nSelectionList — The number of items selected.
- pSelectionList — A pointer to the objects current selected in the iTool.

These values can be helpful in performing actions in the manipulator and are automatically managed. See “[Using Manipulator Public Instance Data](#)” in Chapter 8 of the *iTool Developer’s Guide* manual for more information.

Syntax

Obj->[IDLitManipulator:]OnMouseDown, *Win*, *X*, *Y*, *IButton*, *KeyMods*, *NClicks*

Arguments

Win

An object reference for the IDLitWindow from which the event came.

X

A floating-point value representing the *x*-coordinate in the window’s coordinates.

Y

A floating-point value representing the *y*-coordinate in the window’s coordinates.

IButton

An integer value representing the mask for the button pressed:

- 1 = Left
- 2 = Middle

- 4 = Right

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values:

- 1 = SHIFT key
- 2 = CTRL key
- 4 = CAPS LOCK key
- 8 = ALT key

NClicks

An integer value representing number of clicks

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::OnMouseMotion

The IDLitManipulator::OnMouseMotion procedure method implements the OnMouseMotion method. If no mouse button is down, this method manages the setting of the cursor on the IDLitWindow object.

If the manipulator is in the middle of a mouse-down, mouse-up transaction, the following instance data will be valid:

- ButtonPress — Set to the mouse buttons pressed.
- nSelectionList — The number of items selected.
- pSelectionList — A pointer to the objects current selected in the iTool.

These values can be helpful in performing actions in the manipulator and are automatically managed. See “[Using Manipulator Public Instance Data](#)” in Chapter 8 of the *iTool Developer’s Guide* manual for more information.

Syntax

Obj->[IDLitManipulator::]OnMouseMotion, *Win*, *X*, *Y*, *KeyMods*

Arguments

Win

An object reference for the IDLitWindow object from which the event came.

X

A floating-point value representing the *x*-coordinate in the window’s coordinates.

Y

A floating-point value representing the *y*-coordinate in the window’s coordinates.

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values:

- 1 = SHIFT key
- 2 = CTRL key
- 4 = CAPS LOCK key

- 8 = ALT key

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::OnMouseUp

The IDLitManipulator::OnMouseUp procedure method is used when a mouse up event occurs on the target IDLitWindow object.

When overriding this method in a sub-class, this method is normally the last task performed so that the instance data of the object is properly setup.

Syntax

Obj->[IDLitManipulator::]OnMouseUp, *Win, X, Y, IButton*

Arguments

Win

An object reference for the IDLitWindow object from which the event came.

X

A floating-point value representing the *x*-coordinate in the window's coordinates.

Y

A floating-point value representing the *y*-coordinate in the window's coordinates.

IButton

An integer value representing the mask for the button pressed:

- 1 = Left
- 2 = Middle
- 4 = Right

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::RecordUndoValues

The IDLitManipulator::RecordUndoValues function method is used to begin recording the transaction that is occurring as a result of the manipulator interaction. This method works in conjunction with the CommitUndoValues method.

To capture information for the undo-redo system of the tools framework, the initial and final values for the items that the manipulator will adjust must be recorded. This recording process is accomplished with the RecordUndoValues and CommitUndoValues methods.

These two methods make use of the operation that is associated with this manipulator (specified by the OPERATION_IDENTIFIER keyword to the Init method) to capture the needed information. When the RecordUndoValues method is called, the manipulator will retrieve the operation associated with this manipulator and call the RecordInitialValues method on this operation. The RecordInitialValues method is called with a command set, the list of currently selected items (which the manipulator is operating on) and the value that was provided by the PARAMETER_IDENTIFIER keyword to this manipulator. This method will record any information needed to undo the operation and place the values in the provided command set. See IDLitOperation for more information on operations.

For interactive manipulators, the RecordUndoValues method is called as part of the OnMouseDown method.

Syntax

Result = *Obj*->[IDLitManipulator::]RecordUndoValues()

Return Value

Returns a 1 if recording was successful, or 0 otherwise.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitManipulator::RegisterCursor

The IDLitManipulator::RegisterCursor procedure method defines the appearance and name of a cursor associated with the manipulator.

Syntax

Obj->[IDLitManipulator:]RegisterCursor, *ArrCursor*, *Name* [, /DEFAULT]

Return Value

None

Arguments

ArrCursor

This must be a 16 element string array of 16 characters each that represents a window cursor. The array can consist of the “#” character, space and non-space characters, and optionally, a “\$” character as follows:

- The “#” characters define the black outline of the cursor.
- Any non-space characters define the shape of the white mask. Adding a mask ensures the cursor’s visibility in light and dark areas of a visualization.
- The “\$” defines a hot spot.

See [CREATE_CURSOR](#) for more information on the structure of the cursor array.

Name

A string value identifying the cursor.

Note

This name should be returned by the GetCursorType function method for your manipulator.

Keywords

DEFAULT

Set this keyword to specify this cursor as the default cursor for the manipulator. If no cursor is specified as the default cursor, then the standard arrow is used.

Note

This will be the cursor used if the `GetCursorType` method returns a null string.

Example

The `RegisterCursor` method is typically called within the scope of a manipulator's initialization so that the cursor appears when the manipulator (`my_manip`) is activated. The following code defines the appearance of the manipulator cursor and sets it as the default.

```

; Define the default cursor for this manipulation.
strArray = [ $
'          ', $
'          ', $
'          ', $
'          ', $
'          ', $
'          ', $
'  .#.      .#. ', $
'  .#. . . . .#. ', $
' .#####. ', $
'###. . . .$.### ', $
' .#####. ', $
'  .#. . . . .#. ', $
'  .#.      .#. ', $
'          ', $
'          ', $
'          ', $
'          ', $
'          ', $
'          ' ]

; Register the new cursor with the manipulator.
my_manip->RegisterCursor, strArray, 'SCROLL', /DEFAULT

```

Version History

Introduced: 6.1

IDLitManipulator::SetCurrentManipulator

The IDLitManipulator::SetCurrentManipulator procedure method sets the manipulator as the current manipulator in the system.

Syntax

Obj->[IDLitManipulator:]SetCurrentManipulator [, *Item*]

Arguments

Item

If this method is being called on a manipulator container object, *Item* can optionally be set to a string containing the relative object ID of the manipulator that should be set as the current manipulator. If *Item* is not supplied, the manipulator on which the method is called is set as the current manipulator.

Keyword

None

Version History

Introduced: 6.0

IDLitManipulator:: SetProperty

The IDLitManipulator::SetProperty procedure method sets the value of an IDLitManipulator property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitManipulator::]SetProperty[, *PROPERTY*=*value*]

Arguments

None

Keywords

Any property listed under “[IDLitManipulator Properties](#)” on page 3208 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitManipulatorContainer

The IDLitManipulatorContainer class allows for the construction of manipulator hierarchies. Building on the functionality provided by the IDLitManipulator class, the manipulator container implements the concept of a *current manipulator* for the items it contains.

This class is written in the IDL language. Its source code can be found in the file `idlitmanipulatorcontainer__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitContainer](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitManipulatorContainer::Init](#)” on page 3242.

Properties

Objects of this class have the following properties. See “[IDLitManipulatorContainer Properties](#)” on page 3237 for details on individual properties.

- [AUTO_SWITCH](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLitManipulatorContainer::Add](#)
- [IDLitManipulatorContainer::GetCurrent](#)
- [IDLitManipulatorContainer::GetCurrentManipulator](#)
- [IDLitManipulatorContainer::GetProperty](#)

- [IDLitManipulatorContainer::Init](#)
- [IDLitManipulatorContainer::OnKeyboard](#)
- [IDLitManipulatorContainer::OnMouseDown](#)
- [IDLitManipulatorContainer::OnMouseMove](#)
- [IDLitManipulatorContainer::OnMouseUp](#)
- [IDLitManipulatorContainer::SetCurrent](#)
- [IDLitManipulatorContainer::SetCurrentManipulator](#)
- [IDLitManipulatorContainer::SetProperty](#)

In addition, this class inherits the methods of its superclasses.

Version History

Introduced: 6.0

IDLitManipulatorContainer Properties

IDLitManipulatorContainer objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the table are retrieved via [IDLitManipulatorContainer::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table are set via [IDLitManipulatorContainer::Init](#). Properties with the word “Yes” in the “Set” column in the property table are set via [IDLitManipulatorContainer::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

AUTO_SWITCH

A boolean that, if set, causes the manipulator container to automatically change the value of its current child manipulator depending on the portion of the selection visual hit during a mouse down operation. This is performed by matching the name of the portion of the selection visual hit during the mouse down operation and the name of the sub-manipulators of this container.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLitManipulatorContainer::Add

The IDLitManipulatorContainer::Add procedure method is used to add a new manipulator to the container.

Syntax

Obj->[IDLitManipulatorContainer::]Add, *Manipulator*

Arguments

Manipulator

The new manipulator to add to this manipulator container. When completed, this manipulator is also set to the current manipulator for this manipulator container.

Keywords

All keywords are passed to the IDL_Container superclass during the add operation.

Version History

Introduced: 6.0

IDLitManipulatorContainer::GetCurrent

The IDLitManipulatorContainer::GetCurrent function method is used to get the object reference of the current manipulator of the container.

Syntax

Result = *Obj*->[IDLitManipulatorContainer::]GetCurrent()

Return Value

Returns an object reference to the current manipulator for this container or a null object reference if no manipulators are contained by the container.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer::GetCurrentManipulator

The IDLitManipulatorContainer::GetCurrentManipulator function method is used to get the current manipulator of the system. When applied, this function method causes a traversal of the manipulator hierarchy, returning the current leaf manipulator of the system.

Syntax

Result =
Obj->[IDLitManipulatorContainer::]GetCurrentManipulator([, /IDENTIFIER])

Return Value

Returns an object reference to the current manipulator in the system. If the IDENTIFIER keyword is set, the routine will return the identifier for the current manipulator.

Arguments

None

Keywords

IDENTIFIER

If set, the identifier of the current manipulator is returned. If not set, the current manipulator itself is returned.

Version History

Introduced: 6.0

IDLitManipulatorContainer::GetProperty

The IDLitManipulatorContainer::GetProperty procedure method retrieves the value of an IDLitManipulatorContainer property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclass.

Syntax

Obj->[IDLitManipulatorContainer:]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitManipulatorContainer Properties](#)” on page 3237 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitManipulatorContainer::Init

The IDLitManipulatorContainer::Init function method initializes the IDLitManipulatorContainer object, and should be called by the subclass' Init method. This method also calls the superclass' Init method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitManipulatorContainer'[, *PROPERTY=value*])

or

Result = *Obj*->[IDLitManipulatorContainer::]Init([*PROPERTY=value*])
(In a subclass' Init method only.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLitManipulatorContainer Properties](#)” on page 3237 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitManipulatorContainer::OnKeyboard

The IDLitManipulatorContainer::OnKeyboard procedure method is used when a keyboard event occurs on the target IDLitWindow object. If the current child manipulator supports keyboard events, the OnKeyboard method for that manipulator is called.

Syntax

Obj->[IDLitManipulator::]OnKeyboard, *Win*, *IsASCII*, *Character*, *Key Value*, *X*, *Y*, *Press*, *Release*, *KeyMods*

Arguments

Win

An object reference for the IDLitWindow object from which the event came.

IsASCII

A Boolean value to determine if the character is in the ASCII character set. This value is True if the character is in the ASCII character set.

KeyValue

The numeric value representing a non-ASCII key pressed. The possible values are:

- 1 = Shift
- 2 = Control
- 3 = Caps Lock
- 4 = Alt
- 5 = Left
- 6 = Right
- 7 = Up
- 8 = Down
- 9 = Page Up
- 10 = Page Down
- 11 = Home

- 12 = End

Character

A string containing the ASCII character of the key pressed

X

A floating-point value representing the *x*-coordinate in the window's coordinates.

Y

A floating-point value representing the *y*-coordinate in the window's coordinates.

Press

A Boolean value to determine if the current event is a key press. This value is True if the event is pressing the key

Release

A Boolean value to determine if the current event is a key release. This value is True if event is releasing the key

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values:

- 1 = SHIFT key
- 2 = CTRL key
- 4 = CAPS LOCK key
- 8 = ALT key

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer::OnMouseDown

The `IDLitManipulatorContainer::OnMouseDown` procedure method is used when a mouse down event occurs on the target `IDLitWindow` object. When applied and auto-switch mode is enabled in the container, the current child manipulator will be determined by matching the type of the visualization hit with the identifier of the sub-manipulators contained in this container. If no match is made, the default child manipulator is selected. Once complete, normal processing continues.

If the current sub-manipulator supports mouse button events, its `OnMouseDown` method is called.

Syntax

Obj->[IDLitManipulatorContainer:]OnMouseDown, *Win*, *X*, *Y*, *IButton*, *KeyMods*,
NClicks

Arguments

Win

An object reference for the `IDLitWindow` object from which the event came.

X

A floating-point value representing the *x*-coordinate in the window's coordinates.

Y

A floating-point value representing the *y*-coordinate in the window's coordinates.

IButton

An integer value representing the mask for the button pressed:

- 1 = Left
- 2 = Middle
- 4 = Right

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values:

- 1 = SHIFT key
- 2 = CTRL key
- 4 = CAPS LOCK key
- 8 = ALT key

NClicks

An integer value representing number of clicks

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer::OnMouseMotion

The IDLitManipulatorContainer::OnMouseMotion procedure method implements the OnMouseMotion method. If no mouse button is down and auto-switch mode is enabled, this method manages the setting of the cursor on the IDLitWindow object. Otherwise if the current sub-manipulator supports motion events, its OnMouseMotion method is called.

Syntax

Obj->[IDLitManipulatorContainer:]OnMouseMotion, *Win*, *X*, *Y*, *KeyMods*

Arguments

Win

An object reference for the IDLitWindow object from which the event came.

X

A floating-point value representing the *x*-coordinate in the window's coordinates.

Y

A floating-point value representing the *y*-coordinate in the window's coordinates.

KeyMods

An integer representing the keyboard modifiers for the button. These can have one of the following values:

- 1 = SHIFT key
- 2 = CTRL key
- 4 = CAPS LOCK key
- 8 = ALT key

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer::OnMouseUp

The IDLitManipulatorContainer::OnMouseUp procedure method is used when a mouse up event occurs on the target IDLitWindow object. When applied, the container will call the OnMouseUp method of the current child manipulator if that manipulator supports button events.

Syntax

Obj->[IDLitManipulatorContainer:]OnMouseUp, *Win*, *X*, *Y*, *IButton*

Arguments

Win

An object reference for the IDLitWindow object from which the event came.

X

A floating-point value representing the *x*-coordinate in the window's coordinates.

Y

A floating-point value representing the *y*-coordinate in the window's coordinates.

IButton

An integer value representing the mask for the button pressed:

- 1 = Left
- 2 = Middle
- 4 = Right

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer::SetCurrent

The IDLitManipulatorContainer::SetCurrent procedure method is used to set a manipulator within the container to be the current manipulator.

Syntax

Obj->[IDLitManipulatorContainer::]SetCurrent, *Manipulator*

Arguments

Manipulator

The object reference of the manipulator to be set as current in the manipulator container. This manipulator must be contained in the container or an error is thrown.

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer::SetCurrentManipulator

The IDLitManipulatorContainer::SetCurrentManipulator procedure method is used to set the current child manipulator in a manipulator hierarchy. When applied, the manipulator will use the provided identifier to call the SetCurrentManipulator on the target child manipulator. If the provided identifier is null, the default manipulator is set as current.

Syntax

Obj->[IDLitManipulatorContainer:]SetCurrentManipulator, *Identifier*

Arguments

Identifier

Either the identifier for the manipulator that is to be made current in the system or its object reference.

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorContainer:: SetProperty

The IDLitManipulatorContainer::SetProperty procedure method sets the value of an IDLitManipulatorContainer property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitManipulator::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLitManipulatorContainer Properties](#)” on page 3237 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitManipulatorManager

The `IDLitManipulatorManager` class is a specialization of a `IDLitManipulatorContainer` class that acts as the root of a manipulator hierarchy. In addition, this class provides the functionality that allows it to interface with external elements in the iTool system.

This class is written in the IDL language. Its source code can be found in the file `idlitmanipulatormanager__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitManipulatorContainer](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitManipulatorManager::Init](#)” on page 3260.

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLitManipulatorManager::Add](#)
- [IDLitManipulatorManager::AddManipulatorObserver](#)
- [IDLitManipulatorManager::GetDefaultManipulator](#)
- [IDLitManipulatorManager::Init](#)
- [IDLitManipulatorManager::RemoveManipulatorObserver](#)

In addition, this class inherits the methods of its superclass.

Version History

Introduced: 6.0

IDLitManipulatorManager Properties

Objects of this class have no properties of their own, but do have properties inherited from its superclass.

IDLitManipulatorManager::Add

The IDLitManipulatorManager::Add procedure method is used to add manipulators to the manipulator manager. This method uses the add functionality of its superclass to add the given manipulator to the hierarchy and checks the default keyword to determine if the given manipulator should be treated as the default for the system.

Syntax

Obj->[IDLitManipulatorManager::]Add, *Manipulator* [, /DEFAULT]

Arguments

Manipulator

A manipulator to add to the manipulator manager

Keywords

DEFAULT

If set, the added manipulator is made default for the system. This is primarily used with manipulators that implement transient behavior (mouse or keyboard). When these manipulators complete interaction, the current manipulator of the system is set to the default manipulator.

Version History

Introduced: 6.0

IDLitManipulatorManager::AddManipulatorObserver

The IDLitManipulatorManager::AddManipulatorObserver procedure method is used to add an observer to the manipulator system. An observer object is notified when the current manipulator of the manipulator hierarchy changes.

To perform this action, the observer object must implement the method OnManipulatorChange. This method has the following signature:

```
PRO ObjectClass::OnManipulatorChange, Subject
```

Where *Subject* is this class, the manipulator manager.

Syntax

```
Obj->[IDLitManipulatorManager::]AddManipulatorObserver, Observer
```

Arguments

Observer

A scalar or array of objects that implement the manipulator observer interface.

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorManager::GetDefaultManipulator

The IDLitManipulatorManager::GetDefaultManipulator function method returns a reference to the manipulator that was most recently added as the default manipulator. The return value is a reference to an IDLitManipulator object, or the NULL object reference if no manipulators have been added as a default.

Syntax

Result = *Obj*->[IDLitManipulatorManager::]->GetDefaultManipulator()

Keywords

None

Arguments

None

Version History

Introduced: 6.1

IDLitManipulatorManager::Init

The IDLitManipulatorManager::Init function method initializes the IDLitManipulatorManager object, and should be called by the subclass' Init method. This method also calls the superclass' Init method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitManipulatorManager' [, *PROPERTY=value*])

or

Result = *Obj*->[IDLitManipulatorManager::]Init([*PROPERTY=value*])

(In a subclass' Init method only.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLitManipulatorManager Properties](#)” on page 3256 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitManipulatorManager::RemoveManipulatorObserver

The IDLitManipulatorManager::RemoveManipulatorObserver procedure method is used to remove a manipulator observer from the manipulator object. If the value for the *Observer* argument is not a valid manipulator observer, this method does nothing and no error is reported.

Syntax

Obj->[IDLitManipulatorManager:]RemoveManipulatorObserver, *Observer*

Arguments

Observer

A scalar or array of objects that are to be removed from this container

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorVisual

The IDLitManipulatorVisual class implements methods that allow the iTool developer to create visual elements that are associated with an interactive manipulator. IDLitManipulatorVisual objects are not intended to be created as standalone entities; rather, this class should be included as the superclass of another iTool class.

Superclasses

[IDLgrModel](#)

[IDLitIMessaging](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitManipulatorVisual::Init](#)” on page 3268.

Properties

Objects of this class have the following properties. See “[IDLitManipulatorVisual Properties](#)” on page 3265 for details on individual properties.

- [UNIFORM_SCALE](#)
- [VISUAL_TYPE](#)

Methods

This class has the following methods:

- [IDLitManipulatorVisual::Cleanup](#)
- [IDLitManipulatorVisual::GetProperty](#)
- [IDLitManipulatorVisual::Init](#)
- [IDLitManipulatorVisual::SetProperty](#)

Version History

Introduced: 6.0

IDLitManipulatorVisual Properties

IDLitManipulatorVisual objects have the following properties. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitManipulatorVisual::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitManipulatorVisual::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitManipulatorVisual::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

UNIFORM_SCALE

Set this property to create a manipulator visual which should be scaled equally in all three dimensions when fitting the manipulator visual to the size of its associated visualization. If this property is not set, then the manipulator visual will use a different scale factor in the x , y , and z dimensions, in order to match its associated visualization.

Property Type	Boolean		
Name String	<i>not applicable</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

VISUAL_TYPE

Set this keyword to a string representing the manipulator visual type. The VISUAL_TYPE property should match the corresponding string in its associated IDLitManipulator object. When a new manipulator is selected in an iTool, the VISUAL_TYPE property on the manipulator object is used to search for a corresponding manipulator visual.

Property Type	String		
Name String	<i>not applicable</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLitManipulatorVisual::Cleanup

The IDLitManipulatorVisual::Cleanup procedure method performs all cleanup on the object, and should be called by the Cleanup method of a subclass.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitManipulatorVisual::]Cleanup (*only in subclass' Cleanup method*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitManipulatorVisual::GetProperty

The IDLitManipulatorVisual::GetProperty procedure method retrieves the value of an IDLitManipulatorVisual property.

Syntax

Obj->[IDLitManipulatorVisual::]GetProperty

Arguments

None

Keywords

Any property listed under [“IDLitManipulatorVisual Properties”](#) on page 3265 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitManipulatorVisual::Init

The IDLitManipulatorVisual::Init function method initializes the IDLitManipulatorVisual object, and should be called by the Init method of a subclass.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitManipulatorVisual'[, PROPERTY=value])
```

or

```
Result = Obj->[IDLitManipulatorVisual::]Init([, PROPERTY=value])  
(In a subclass' Init method only.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLitManipulatorVisual Properties](#)” on page 3265 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitManipulatorVisual:: SetProperty

The IDLitManipulatorVisual::SetProperty procedure method sets the value of an IDLitManipulatorVisual property.

Syntax

Obj->[IDLitManipulatorVisual::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLitManipulatorVisual Properties](#)” on page 3265 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitOperation

The IDLitOperation class is the basis for all iTool operations. It defines how an operation is executed and how information about the operation is recorded for the command transaction (undo-redo) system.

An operation encapsulates a specific set of tasks that perform some atomic action on a target. Between executions of the operation, no state values other than properties of the operation itself are maintained by this object. Two major areas of functionality are provided: automated operation execution and undo-redo capabilities.

Once defined, the operation object exposes a set of properties whose values can affect how the operation executes. The properties that an operation exposes are valid across multiple executions; properties can be set interactively via the Operation browser or programmatically using the operation object's GetProperty and SetProperty methods.

Note

For operations that act directly on data items that underlie a visualization, consider basing your operation class on the IDLitDataOperation class. IDLitDataOperation is a subclass of IDLitOperation that automates many of the target-selection and undo-redo mechanisms that must be provided by operation classes based directly on this class.

For a detailed discussion of both the IDLitOperation and IDLitDataOperation classes, see [Chapter 7, “Creating an Operation”](#) in the *iTool Developer's Guide* manual.

This class is written in the IDL language. Its source code can be found in the file `idlitoperation__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitComponent](#)

[IDLitIMessaging](#)

Note

See the [“iTools Object Model Diagram”](#) in Chapter 2 of the *iTool Developer's Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitOperation::Init](#)” on page 3281.

Properties

Objects of this class have the following properties. See “[IDLitOperation Properties](#)” on page 3275 for details on individual properties.

- [EXPENSIVE_COMPUTATION](#)
- [REVERSIBLE_OPERATION](#)
- [SHOW_EXECUTION_UI](#)
- [TYPES](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Note

By default, for `IDLitOperation` and its subclasses, the `NAME` property is hidden and the `DESCRIPTION` property is desensitized. This prevents a user from changing the name of an operation via the property sheet, which could change the name of a menu item. If you wish to expose these properties for your own subclass of `IDLitOperation`, you will need to unhide and sensitize the `NAME` and `DESCRIPTION` properties in your `Init` method.

Methods

This class has the following methods:

- [IDLitOperation::Cleanup](#)
- [IDLitOperation::DoAction](#)
- [IDLitOperation::GetProperty](#)
- [IDLitOperation::Init](#)
- [IDLitOperation::RecordFinalValues](#)
- [IDLitOperation::RecordInitialValues](#)
- [IDLitOperation::RedoOperation](#)
- [IDLitOperation::SetProperty](#)
- [IDLitOperation::UndoOperation](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

See Also

[IDLitDataOperation](#)

IDLitOperation Properties

IDLitOperation objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the properties table can be retrieved via [IDLitOperation::GetProperty](#). Properties with the word “Yes” in the “Init” column of the properties table can be initialized via [IDLitOperation::Init](#). Properties with the word “Yes” in the “Set” column of the properties table can be set via [IDLitOperation::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

EXPENSIVE_COMPUTATION

Set this property to 1 to indicate that a computation of the operation is *expensive*. Expensive computations are those that require significant memory or processing time to execute. Individual operations should use the value of this property to determine whether the results of the operation should be cached to avoid re-execution when undoing or redoing.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

REVERSIBLE_OPERATION

Set this property to 1 to indicate that the operation is *reversible*. When an operation is reversible, it can be undone by applying an operation rather than restoring a stored value. Rotation by a specified angle is an example of an operation that is reversible, since applying another rotation by the complementary angle returns the visualization to its original state. Individual operations should use the value of this property to determine how the operation should be undone.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

SHOW_EXECUTION_UI

Set this property to 1 to indicate that the operation should display a user interface element such as a dialog when the operation is executed. The flag is ignored by the IDLiteOperation itself, but can be used by subclasses (notably IDLiteDataOperation) to determine when to display user interface elements. This property is registered, but it is hidden by default.

Property Type	Boolean		
Name String	Show Dialog		
Get: Yes	Set: No	Init: Yes	Registered: Yes

TYPES

Set this property to a string or an array of strings indicating the types of data to which the operation can be applied. iTools data types are described in [Chapter 3, “Data Management”](#) in the *iTool Developer’s Guide* manual. Set this property to a null string (' ') to specify that the operation can be applied to all types of data.

Property Type	User Defined		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

IDLitOperation::Cleanup

The IDLitOperation::Cleanup procedure method performs all cleanup on the object.

Note

An operation based on the IDLitOperation class need not implement this method if the operation does not allocate any pointers or object references of its own.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitOperation::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitOperation::DoAction

The `IDLitOperation::DoAction` function method is called when an operation is requested by the iTools system, either as the result of a user action such as selection of a menu item or toolbar button, or by another operation.

Note

Every operation based directly on the `IDLitOperation` class must implement its own `DoAction` method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

A specific operation class’ implementation of this method must create an initialized `IDLitCommandSet` object, execute the operation, and return the command set object to the caller. `DoAction` can also call the [IDLitOperation::RecordInitialValues](#) and [IDLitOperation::RecordFinalValues](#) methods to ensure that the appropriate values are committed to the undo-redo transaction buffer.

If your operation changes the values of its own registered properties (as the result of user interaction with a dialog or other interface element called by `DoUIService`, for example), be sure to call the `RecordInitialValues` and `RecordFinalValues` methods. This ensures that changes made through the dialog are placed in the undo-redo transaction buffer.

Note

If you are creating an operation that acts directly on the data that underlies a visualization, consider using the [IDLitDataOperation](#) class rather than the `IDLitOperation` class. `IDLitDataOperation` provides additional automation of the Undo/Redo mechanism.

Syntax

Result = Obj->[IDLitOperation::]DoAction(Tool)

Return Value

If successful, this method returns an `IDLitCommandSet` object or array of `IDLitCommandSet` objects that will be placed in the iTool’s command buffer. The command set object will be passed to the `UndoOperation` and `RedoOperation` methods as necessary. If not successful, this method returns a null object reference.

Note

The OPERATION_IDENTIFIER property of the command set object returned by the IDLitOperation::DoAction method is set equal to the full identifier of the operation itself. This implies that UndoOperation and RedoOperation methods exist for the operation, and that they implement the necessary undo and redo mechanisms. If you are overriding this method, you must either ensure that the appropriate UndoOperation and RedoOperation methods exist, or manually create an IDLitCommandSet object with the OPERATION_IDENTIFIER set to another operation that is capable of undoing or redoing the changes made by your method.

Arguments

Tool

An object reference to the iTool object on which the operation executes.

Keywords

None

Version History

Introduced: 6.0

IDLitOperation::GetProperty

The IDLitOperation::GetProperty procedure method retrieves the value of a property or group of properties of an operation object.

Note

An operation based on the IDLitOperation class *must* implement this method if the operation defines one or more properties not inherited from the superclass. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Syntax

```
Obj->[IDLitOperation::]GetProperty [, PROPERTY=variable]
```

Arguments

None

Keywords

Any property listed under [“IDLitOperation Properties”](#) on page 3275 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Example

To retrieve the value of the REVERSIBLE_OPERATION property:

```
Obj->IDLitOperation::GetProperty, REVERSIBLE_OPERATION = rev
```

Version History

Introduced: 6.0

IDLitOperation::Init

The IDLitOperation::Init function method initializes the IDLitOperation object and sets properties that define the behavior the operation provides.

Note

An operation based on the IDLitOperation class *must* implement this method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitOperation' [, *PROPERTY=variable*])

or

Result = *Obj*->[IDLitOperation::]Init([, *PROPERTY=variable*]) (*Only in a subclass' Init method.*)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under [“IDLitOperation Properties”](#) on page 3275 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitOperation::RecordFinalValues

The `IDLitOperation::RecordFinalValues` function method records the information needed to redo an operation. Information about individual operations is recorded in `IDLitCommand` objects, which are in turn stored in an `IDLitCommandSet` object that is passed to the [IDLitOperation::UndoOperation](#) and [IDLitOperation::RedoOperation](#) methods.

Note

If you are creating an operation that acts directly on the data that underlies a visualization, consider using the [IDLitDataOperation](#) class rather than the `IDLitOperation` class. `IDLitDataOperation` provides additional automation of the Undo and Redo mechanisms.

If this method is implemented for an operation, it should do the following:

- Retrieve the appropriate `IDLitCommand` object from the `IDLitCommandSet` object specified as the first parameter,
- Record any information necessary to redo the operation in the `IDLitCommand` object,
- Return 1 for success.

Syntax

```
Result = Obj->[IDLitOperation::]RecordFinalValues(CommandSet, Targets  
[, IdProperty])
```

Return Value

The return value is 1 if the final values were recorded successfully, or zero otherwise.

Arguments

CommandSet

An `IDLitCommandSet` object that encapsulates all information recorded for this execution of the operation. The command set object is generally created by the [IDLitOperation::DoAction](#) method, which then calls the `RecordInitialValues` method to store the initial state of the target object.

Targets

An array of objects that are the targets of this operation. In most cases, the target objects are the currently selected objects, as returned by the [IDLitTool::GetSelectedItems](#) method.

IdProperty

A property identifier string specifying the target property or parameter of the operation.

Keywords

None

Version History

Introduced: 6.0

IDLitOperation::RecordInitialValues

The `IDLitOperation::RecordInitialValues` function method records the information needed to undo an operation. Information about individual operations is recorded in `IDLitCommand` objects, which are in turn stored in an `IDLitCommandSet` object that is passed to the [IDLitOperation::UndoOperation](#) and [IDLitOperation::RedoOperation](#) methods.

Note

If you are creating an operation that acts directly on the data that underlies a visualization, consider using the [IDLitDataOperation](#) class rather than the `IDLitOperation` class. `IDLitDataOperation` provides additional automation of the Undo and Redo mechanisms.

If this method is implemented for an operation, it should do the following:

- Create an `IDLitCommand` object, setting its `TARGET_IDENTIFIER` property to the identifier of the target object,
- Record any information necessary to redo the operation in the `IDLitCommand` object,
- Add the `IDLitCommand` object to the `IDLitCommandSet` object specified as the first parameter,
- Return 1 for success.

Syntax

```
Result = Obj->[IDLitOperation::]RecordInitialValues(CommandSet, Targets  
[, IdProperty])
```

Return Value

The return value is 1 if the initial values were recorded successfully, or zero otherwise.

Arguments

CommandSet

An `IDLitCommandSet` object that encapsulates all information recorded for this execution of the operation. The command set object is generally created by the `IDLitOperation::DoAction` method, which then calls the `RecordInitialValues` method to store the initial state of the target object.

Targets

An array of objects that are the targets of this operation. In most cases, the target objects are the currently selected objects, as returned by the `IDLitTool::GetSelectedItems` method.

IdProperty

A property identifier string specifying the target property or parameter of the operation.

Keywords

None

Version History

Introduced: 6.0

IDLitOperation::RedoOperation

The IDLitOperation::RedoOperation function method is called by the iTool system when the user requests the re-execution of an operation (usually by selecting Redo from the iTool Edit menu or toolbar).

Note

An operation based on the IDLitOperation class *must* implement this method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Note

If you are creating an operation that acts directly on the data that underlies a visualization, consider using the [IDLitDataOperation](#) class rather than the IDLitOperation class. IDLitDataOperation provides additional automation of the Undo and Redo mechanisms.

If this method is implemented for an operation, it should do the following:

- Retrieve the IDLitCommand objects from the IDLitCommandSet object passed as the first argument,
- For each IDLitCommand object, retrieve the target of the operation, using the TARGET_IDENTIFIER property,
- Retrieve the stored information and use it to redo the operation,
- Return 1 for success.

Syntax

```
Result = Obj->[IDLitOperation:]RedoOperation(CommandSet)
```

Return Value

The return value is 1 if the operation was redone successfully, or zero otherwise.

Arguments

CommandSet

An IDLitCommandSet object that contains all the IDLitCommand objects that were stored during the original execution of the operation.

Keywords

None

Version History

Introduced: 6.0

See Also

[IDLitOperation::RecordFinalValues](#), [IDLitOperation::RecordInitialValues](#),
[IDLitOperation::UndoOperation](#)

IDLitOperation:: SetProperty

The IDLitOperation::SetProperty procedure method sets the value of a property or group of properties for the operation.

Note

An operation based on the IDLitOperation class *must* implement this method if the operation defines one or more properties not inherited from the superclass. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Syntax

Obj->[IDLitOperation::]SetProperty [, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under [“IDLitOperation Properties”](#) on page 3275 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitOperation::UndoOperation

The IDLitOperation::UndoOperation function method is called by the iTool system when the user requests the un-execution of an operation (usually by selecting Undo from the iTool Edit menu or toolbar).

Note

An operation based on the IDLitOperation class *must* implement this method. See [Chapter 7, “Creating an Operation”](#) in the *iTool Developer’s Guide* manual for details.

Note

If you are creating an operation that acts directly on the data that underlies a visualization, consider using the [IDLitDataOperation](#) class rather than the IDLitOperation class. IDLitDataOperation provides additional automation of the Undo and Redo mechanisms.

If this method is implemented for an operation, it should do the following:

- Retrieve the IDLitCommand objects from the IDLitCommandSet object passed as the first argument,
- For each IDLitCommand object, retrieve the target of the operation, using the TARGET_IDENTIFIER property,
- Retrieve the stored information and use it to undo the operation,
- Return 1 for success.

Syntax

```
Result = Obj->[IDLitOperation::]UndoOperation(CommandSet)
```

Return Value

The return value is 1 if the operation was undone successfully, or zero otherwise.

Arguments

CommandSet

An IDLitCommandSet object that contains all the IDLitCommand objects that were stored during the original execution of the operation.

Keywords

None

Version History

Introduced: 6.0

See Also

[IDLitOperation::RecordFinalValues](#), [IDLitOperation::RecordInitialValues](#),
[IDLitOperation::RedoOperation](#)

IDLitParameter

The IDLitParameter class implements parameter management methods that allow parameter names to be associated with IDLitData objects. IDLitParameter objects are not intended to be created as standalone entities; rather, this class should be included as the superclass of another iTool class.

Note

In the iTools system, management of data parameters is handled by the IDLitVisualization class, of which the IDLitParameter class is a superclass. In practice, this means that if you need to override any methods of the IDLitParameter class, you will do so in the definition of your visualization class. In most cases, the only method you will need to implement is the OnDataChangeUpdate method.

When an object that inherits from the IDLitParameter class is created, the following are created as part of the new object's instance data:

- An IDL_Container object, which will contain parameter names for all parameters registered with the object. Parameter names are discussed in [“Parameters”](#) in Chapter 3 of the *iTool Developer's Guide* manual,
- A pointer to a string array containing the parameter names for all parameters registered with the object,
- An IDLitParameterSet object, which will contain the actual IDLitData objects associated with the object's parameters. Parameter sets are discussed in [“Parameter Sets”](#) in Chapter 3 of the *iTool Developer's Guide* manual, and in [“IDLitParameterSet”](#) on page 3315.

This class is written in the IDL language. Its source code can be found in the file `idlitparameter__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

This class has no superclasses.

Note

See the [“iTools Object Model Diagram”](#) in Chapter 2 of the *iTool Developer's Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

The objects of this class are not created as standalone objects.

Properties

Objects of this class have no properties of their own.

Methods

This class has the following methods:

- [IDLitParameter::Cleanup](#)
- [IDLitParameter::GetParameter](#)
- [IDLitParameter::GetParameterAttribute](#)
- [IDLitParameter::GetParameterSet](#)
- [IDLitParameter::Init](#)
- [IDLitParameter::OnDataChangeUpdate](#)
- [IDLitParameter::OnDataDisconnect](#)
- [IDLitParameter::QueryParameter](#)
- [IDLitParameter::RegisterParameter](#)
- [IDLitParameter::SetData](#)
- [IDLitParameter::SetParameterAttribute](#)
- [IDLitParameter::SetParameterSet](#)

Version History

Introduced: 6.0

IDLitParameter Properties

Objects of this class have no properties of their own.

IDLitParameter::Cleanup

The IDLitParameter::Cleanup procedure method removes the *data observer* from each data object in the visualization object's parameter set, and cleans up the objects and pointers defined to hold parameter data when the visualization object was created.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitParameter::]Cleanup() (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitParameter::GetParameter

The IDLitParameter::GetParameter function method retrieves the IDLitData object associated with a registered parameter.

Syntax

```
Result = Obj->[IDLitParameter::]GetParameter(Name [, /ALL]  
[, COUNT=variable] )
```

Return Value

Returns an object reference to the IDLitData object associated with the specified parameter name, or a null object reference if no data objects are associated with the specified parameter name.

Arguments

Name

A scalar string containing the parameter name associated with the IDLitData object to be retrieved.

Keywords

ALL

Set this keyword to retrieve all of the IDLitData objects contained in the visualization object's IDLitParameterSet object.

COUNT

Set this keyword equal to a named variable that will contain the number of items returned by this function.

Version History

Introduced: 6.0

IDLitParameter::GetParameterAttribute

The IDLitParameter::GetParameterAttribute procedure method retrieves one or more parameter attributes for a registered parameter.

Syntax

```
Obj->IDLitParameter::GetParameterAttribute(ParamName
    [, KEYWORD=variable] )
```

Arguments

ParamName

A scalar string or string array containing names of registered parameters.

Keywords

Any keyword listed under “IDLitParameter::RegisterParameter” on page 3307 can be retrieved using this method. To retrieve the value of an attribute, specify the attribute name as a keyword set equal to a named variable that will contain the value of the attribute.

Example

This example creates an iPlot tool and retrieves the description attribute of the registered parameter named “vertices”:

```
I PLOT, RANDOMU(seed, 15)
idTool = ITGETCURRENT(TOOL=otool)
idPlot = oTool->FindIdentifiers('*plot*', /VISUALIZATIONS)
oPlot = oTool->GetByIdentifier(idPlot)
oPlot->GetParameterAttribute, 'vertices', DESCRIPTION=desc
PRINT, desc
```

IDL prints:

```
Vertex Data
```

Version History

Introduced: 6.1

See Also

[IDLitParameter::QueryParameter](#), [IDLitParameter::RegisterParameter](#),
[IDLitParameter::SetParameterAttribute](#)

IDLitParameter::GetParameterSet

The IDLitParameter::GetParameterSet function method returns a reference to the IDLitParameterSet object associated with the visualization object.

Syntax

Result = *Obj*->[IDLitParameter::]GetParameterSet([, /DEEP_COPY)

Return Value

Returns an IDLitParameterSet object that contains the IDLitData objects and associated parameter names for the visualization object.

Arguments

None

Keywords

DEEP_COPY

Set this keyword to cause GetParameterSet to return a parameter set that contains *copies* of the IDLitData objects used by the visualization object's registered parameters. If this keyword is not set, the parameter set returned by this function will contain *references* to the IDLitData objects that are being used by the visualization object's registered parameters.

Version History

Introduced: 6.0

IDLitParameter::Init

The IDLitParameter::Init function method initializes object instance fields that contain parameter data. Since the IDLitParameter class is always used as a superclass of another class (IDLitVisualization, in the context of the iTool system), initializing this class simply initializes the objects and pointers used to store parameter information in the IDLitVisualization object.

Note

Init methods are special lifecycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Result = Obj->[IDLitParameter::]Init() (Only in subclass' Init method.)

Return Value

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitParameter::OnDataChangeUpdate

The IDLitParameter::OnDataChangeUpdate method is called when a data value has been updated or a new data object has been associated with the visualization object.

Note

A visualization class based on the IDLitVisualization class *must* implement this method in order for changes or additions to the data parameters to be updated automatically in the resulting visualizations.

Syntax

Obj->[IDLitParameter::]OnDataChangeUpdate, *Data*, *ParameterName*

Arguments

Data

An object reference to the IDLitData object or the IDLitParameterSet object containing the changed parameter data. If *Data* is an IDLitParameterSet object, the *ParameterName* argument must be set to the string '<PARAMETER SET>'.

ParameterName

An upper-case scalar string containing the name of the parameter that has changed. The parameter specified via this argument must be registered with the visualization object. If *oData* is an IDLitParameterSet object, this argument must be set to the string '<PARAMETER SET>'.

Keywords

None

Example

Normally, an implementation of this method will contain a CASE statement that switches based on the value of the *ParameterName* argument. For example, the following code handles a parameter named z:

```
PRO MyVis::OnChangeUpdate, oData, ParameterName

CASE ParameterName OF
  'Z': BEGIN
    ;; do something with the z data
  END
  '<PARAMETER SET>': BEGIN
    oZ = oData->GetByName('Z')
    ;; do something with the z data.
  END
ENDCASE

END
```

Note

The Z parameter is handled twice — once if supplied as an individual IDLitData object and once if supplied via the IDLitParameterSet object.

See [“Creating an OnDataChangeUpdate Method”](#) in the section [“Creating a New Visualization Type”](#) in Chapter 6 of the *iTool Developer’s Guide* manual for more discussion of the OnDataChangeUpdate method.

Version History

Introduced: 6.0

IDLitParameter::OnDataDisconnect

The IDLitParameter::OnDataDisconnect procedure method is called when a data value has been disconnected from a parameter. The general idea is that when a data item is disassociated from a visualization parameter, one or more properties of the visualization may need to be reset to reasonable default values.

Note

A visualization class based on the IDLitVisualization class *must* implement this method in order for changes or additions to the data parameters to be updated automatically in the resulting visualizations.

Syntax

Obj->[IDLitParameter:]OnDataDisconnect, *ParameterName*

Arguments

ParameterName

An upper-case string containing the name of the parameter that was disconnected. The parameter must have been previously registered with the visualization.

Keywords

None

Example

Normally, an implementation of this method will contain a CASE statement that switches based on the value of the *ParameterName* argument. For example, the following sample code performs some action when the parameter name is TEXTURE:

```
PRO MyVis::OnDataDisconnect, ParameterName

CASE ParameterName OF
  'TEXTURE': BEGIN
    ;; do something to remove the ref. of the texture
  END
ENDCASE

END
```

Version History

Introduced: 6.0

IDLitParameter::QueryParameter

The IDLitParameter::QueryParameter function method checks whether a parameter is registered, or retrieves a list of the names of all registered parameters.

Syntax

```
Result = Obj->IDLitParameter::QueryParameter([ParamName]
  [, COUNT=variable])
```

Return Value

If *ParamName* is a scalar string, this function returns a 1 if *ParamName* corresponds to a valid registered parameter, or a 0 otherwise. If *ParamName* is an array, the result is an array of 1s and 0s. If *ParamName* is not specified, *Result* is a string array containing the names of all registered parameters.

Arguments

ParamName

A scalar string or string array containing parameter names. If *ParamName* is not supplied, this function returns a list of all registered parameters.

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of parameters in *Result*. If *ParamName* is not supplied, then COUNT contains the number of registered parameters. If *ParamName* is supplied then COUNT contains the total number of matched parameters.

Example

This example creates an iPlot tool and tests whether the plot visualization has a registered parameter named “vertices”:

```
IPLOT, RANDOMU(seed, 15)
idTool = ITGETCURRENT(TOOL=otool)
idPlot = oTool->FindIdentifiers('*plot*', /VISUALIZATIONS)
oPlot = oTool->GetByIdentifier(idPlot)
PRINT, oPlot->QueryParameter('vertices')
```

IDL prints:

```
1
```

Continuing the same example, we now query for the names of all parameters of the plot visualization:

```
Params = oPlot->QueryParameter(COUNT=count)
FOR i=0,count-1 DO PRINT, Params[i]
```

IDL prints:

```
Y
X
VERTICES
Y ERROR
X ERROR
PALETTE
VERTEX_COLORS
```

Version History

Introduced: 6.1

See Also

[IDLitParameter::GetParameter](#), [IDLitParameter::GetParameterAttribute](#),
[IDLitParameter::RegisterParameter](#), [IDLitParameter::SetParameterAttribute](#)

IDLitParameter::RegisterParameter

The IDLitParameter::RegisterParameter procedure method registers a parameter with the visualization object. Registered parameters are displayed in the iTool parameter editor dialog, allowing users to interactively assign data values to individual visualization parameters.

Syntax

```
Obj->[IDLitParameter:]RegisterParameter, Name [, /BY_VALUE]
      [, DESCRIPTION=string] [, /INPUT] [, /OPTARGET] [, /OPTIONAL]
      [, /OUTPUT] [, TYPES=string]
```

Arguments

Name

Set this argument to a string specifying the name of the parameter.

Keywords

BY_VALUE

Set this keyword to indicate that this parameter is a by-value parameter. If a parameter is marked as by value, when data is associated with it a copy of the data is made and managed by this class.

DESCRIPTION

Set this keyword to a string giving the full name or description of this object.

INPUT

Set this keyword to indicate that this is an input parameter.

OPTARGET

Set this keyword to indicate that this parameter may be the target of any operations.

OPTIONAL

If this keyword is set, then this parameter is an optional parameter, and it is not required to be set. The default is `OPTIONAL=0`, indicating that it is required to be set.

OUTPUT

Set this keyword to indicate that this is an output parameter.

TYPES

Set this keyword to a scalar string or string array giving the data types that correspond to this parameter. See [“iTool Data Types”](#) in Chapter 3 of the *iTool Developer’s Guide* manual for more on parameter data types.

Version History

Introduced: 6.0

IDLitParameter::SetData

The IDLitParameter::SetData function method is used to set data in this interface, associating a data object with a given parameter. This method can automatically perform type matching to determine which parameter to associate with a particular parameter or the user can specify a parameter name.

If a parameter name is provided by the user via the PARAMETER_NAME keyword, this method will query the given data object for a data item that is of one of the types supported by this parameter and associate the found data item with the parameter. This could be the data object provided or if the data object contains other data objects, one of its children. If a type match doesn't occur, the method will return 0.

If no parameter name is provided, this method will start with the first available parameter (non-associated parameters, starting with op-targets first) and attempt to perform a data type match. When a data type match between the parameter and the provided data item (or one of its contained items) the association between the two are made.

When a data match has been made, this object is made an observer of the data object and if the NO_UPDATE keyword is not set, the onDataChangeUpdate() method is called on this object.

If a type match is found, but the given parameter already has a data object associated with it, the type match will continue.

If a match cannot be found, a value of 0 is returned.

Syntax

```
Result = Obj->[IDLitParameter::]SetData(Data [, /BY_VALUE] [, /NO_UPDATE]  
[, PARAMETER_NAME=string]
```

Return Value

Returns 1 if the IDLitData object was successfully matched with a parameter registered with the visualization object, or 0 otherwise.

Arguments

Data

An IDLitData object to be matched with a parameter registered with the visualization object.

Keywords

BY_VALUE

Set this keyword to cause *Data* to be copied and managed by the visualization's parameter interface. If this keyword is not set, the data object is managed by the iTool system, and can be altered by other iTools.

NO_UPDATE

Set this keyword to prevent the `OnDataChangeUpdate` method from being called when *Data* is associated with one of the visualization object's registered parameters. If this keyword is not set, the `OnDataChangeUpdate` method is called, providing the opportunity for subclasses of the visualization object to be notified that data associated with the parameter interface has changed.

PARAMETER_NAME

Set this keyword to a scalar string containing the name of the registered parameter with which *Data* should be associated. If the iTool data type of the data contained in *Data* does not match one of the data types supported by the specified registered parameter, *Data* will not be associated with the registered parameter and this function will return zero.

Version History

Introduced: 6.0

IDLitParameter::SetParameterAttribute

The IDLitParameter::SetParameterAttribute procedure method sets one or more parameter attributes for a registered parameter.

Syntax

```
Obj->IDLitParameter::SetParameterAttribute(ParamName
    [, KEYWORD=variable] )
```

Arguments

ParamName

A scalar string or string array containing names of registered parameters. If *ParamName* is an array, the specified attribute values are set on all parameters in the array.

Keywords

Any keyword listed under “IDLitParameter::RegisterParameter” on page 3307 can be retrieved using this method. To retrieve the value of an attribute, specify the attribute name as a keyword set equal to a named variable that will contain the value of the attribute.

Example

This example creates an iPlot tool sets the description attribute of the registered parameter named “vertices”:

```
IPLOT, RANDOMU(seed, 15)
idTool = ITGETCURRENT(TOOL=otool)
idPlot = oTool->FindIdentifiers('*plot*', /VISUALIZATIONS)
oPlot = oTool->GetByIdentifier(idPlot)
oPlot->SetParameterAttribute, 'vertices', $
    DESCRIPTION='My Vertex Data'
oPlot->GetParameterAttribute, 'vertices', DESCRIPTION=desc
PRINT, desc
```

IDL prints:

```
My Vertex Data
```

Version History

Introduced: 6.1

See Also

[IDLitParameter::GetParameterAttribute](#), [IDLitParameter::QueryParameter](#),
[IDLitParameter::RegisterParameter](#)

IDLitParameter::SetParameterSet

The IDLitParameter::SetParameterSet function method is used to associate an IDLitParameterSet object with the visualization object's parameter interface. IDLitData objects contained in the parameter set object will be associated with the registered parameters of the visualization object, either based on the parameter names included in the parameter set object or on the iTool data types of the data objects. When the association of the parameter set's data objects with the visualization object's registered parameters is complete, the onDataChangeUpdate method is called with the IDLitParameterSet object and the string '<PARAMETER SET>' as the arguments.

Note

When this method is called, the IDLitParameterSet object created when the visualization object was initialized will be disassociated from the object's parameter interface. This means that any individual data objects or parameter names previously added to the initial parameter set object will also be disassociated from the visualization object's parameter interface.

This method will not destroy or delete the information contained in the specified IDLitParameterSet object.

Syntax

Result = *Obj*->[IDLitParameter::]SetParameterSet(*ParamSet*)

Return Value

Returns 1 if the IDLitParameterSet object was successfully associated with the visualization object's parameter interface, or 0 otherwise.

Arguments

ParamSet

An IDLitParameterSet object to be associated with the visualization object's parameter interface.

Keywords

None

Version History

Introduced: 6.0

IDLitParameterSet

The IDLitParameterSet class is a specialized subclass of the IDLitDataContainer class that provides the ability to associate names with the contained IDLitData objects. This association allows the iTool developer to package a set of data parameters in a single container, which is then provided to the iTools system for processing a display.

In addition to providing a mechanism for associating IDLitData objects with parameter names, the IDLitParameterSet class allows you to incorporate an IDLitData object for which no parameter name is provided. These *auxiliary data* items are treated as parameters for which the parameter name is a null string (' '). The auxiliary data mechanism allows you to associate any amount of data with a parameter set without the need to define parameter names.

This class is written in the IDL language. Its source code can be found in the file `idlitparameterset__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitDataContainer](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitParameterSet::Init](#)” on page 3327.

Properties

Objects of this class have no properties of their own, but do have properties inherited from the superclass.

Methods

This class has the following methods:

- [IDLitParameterSet::Add](#)

- [IDLitParameterSet::Cleanup](#)
- [IDLitParameterSet::Copy](#)
- [IDLitParameterSet::Get](#)
- [IDLitParameterSet::GetByName](#)
- [IDLitParameterSet::GetParameterName](#)
- [IDLitParameterSet::Init](#)
- [IDLitParameterSet::Remove](#)

In addition, this class inherits the methods of its superclass.

Examples

See [Chapter 3, “Data Management”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitParameterSet Properties

Objects of this class have no properties of their own, but do have properties inherited from the superclass.

IDLitParameterSet::Add

The IDLitParameterSet::Add procedure method is used to add a data object to the parameter set.

Syntax

```
Obj->[IDLitParameterSet::]Add, Data [, PARAMETER_NAME=string]  
[, /PRESERVE_LOCATION]
```

Arguments

Data

An object reference or array of object references to IDLitData objects to be added to the parameter set.

Note

The objects to be added must be of class type IDLitData or a subclass of IDLitData. You cannot add the same Parameter Set object to itself."

Keywords

PARAMETER_NAME

Set this keyword to a string or array of strings containing the names of parameters to be associated with the data objects. If this value is not provided, a null string (' ') is associated with the data object. If the value of this keyword is an array with fewer elements than the array specified as the *Data* argument, null strings are associated with data objects for which no parameter name is specified.

PRESERVE_LOCATION

Set this keyword to leave the PARENT property (of all data items specified by the *Data* argument) unchanged. Normally, when an item is added to a parameter set (or any data container), the item's PARENT property is changed to be the parameter set or data container object. This, in turn, changes the data item's identifier string, since the item's location in the iTools hierarchy has changed. (See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer's Guide* manual for details.) In many cases, it is desirable to leave the PARENT property unchanged when adding items to a parameter set.

Version History

Introduced: 6.0

See Also

[IDLitParameterSet::Remove](#)

IDLitParameterSet::Cleanup

The IDLitParameterSet::Cleanup procedure method performs all cleanup on the parameter set object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitParameterSet::]Cleanup() (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitParameterSet::Copy

The IDLitParameterSet::Copy function method will return a copy of the parameter set and its contents. All items contained in the parameter set are copied when this method is called.

For individual data objects contained in the parameter set, the following values are *not* copied:

- Values of unregistered properties.
- Values of properties registered after the data object is created.
- Property attribute values.

Note

If an object has been added to a contained data object, only the object reference to the added object is copied.

Syntax

```
Result = Obj->[IDLitParameterSet:]Copy()
```

Return Value

Returns an IDLitParameterSet object that is a copy of the parameter set object on which the method was called.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitParameterSet::Get

The IDLitParameterSet::Get function method is used to retrieve one or more IDLitData objects from the parameter set.

Syntax

```
Result = Obj->[IDLitParameterSet::]Get( [, /ALL] [, COUNT=variable]  
[, NAME=variable] [, POSITION=integer]
```

Return Values

Returns an object reference or array of object references to the IDLitData objects specified, or returns -1 if the method fails.

Arguments

None

Keywords

ALL

Set this keyword to return an array containing object references to all IDLitData items in the parameter set.

COUNT

Set this keyword equal to a named variable that will contain the number of items returned by this function.

NAME

Set this keyword equal to a named variable that will contain a string or string array of the parameter names associated with the returned IDLitData objects. If a returned data object has no parameter name, a null string (' ') is returned.

POSITION

Set this keyword equal to a scalar or array containing the zero-based indices of the positions of the desired IDLitData objects within the parameter set.

Version History

Introduced: 6.0

IDLitParameterSet::GetByName

The IDLitParameterSet::GetByName function method returns the IDLitData object associated with a specified named parameter.

Syntax

```
Result = Obj->[IDLitParameterSet::]GetByName( Names [, COUNT=variable]
      [, NAME=variable] )
```

Return Value

An object reference or array of object references to the IDLitData objects specified by the *Names* argument. If no matching parameter names are found, a null object is returned.

Note

If multiple parameter names are specified, the IDLitData objects are returned in the order in which they are stored in the IDLitParameterSet object, which is not necessarily the same as the order specified in the *Names* argument. Use the value returned by the NAME keyword to associate the parameter names with IDLitData objects in the returned array.

Arguments

Names

A scalar string or string array that contains the parameter names of the IDLitData objects to be retrieved from the parameter set.

Note

Parameter names can be supplied in any case; the search is case insensitive.

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of items returned by this function.

NAME

Set this keyword equal to a named variable that will contain a scalar string or string array of the parameter names of the IDLitData objects returned by this method. When multiple parameter names are specified via the *Names* argument, the array returned in the NAME variable can be used to sort through the returned IDLitData objects by matching the indices of the two returned arrays.

Version History

Introduced: 6.0

IDLitParameterSet::GetParameterName

The IDLitParameterSet::GetParameterName function method retrieves the name of a specified parameter using a provided data object.

Syntax

Result = *Obj*->[IDLitParameterSet::]GetParameterName(*Data*, *Name*)

Return Value

Returns an integer 1 if a parameter name was found for the specified IDLitData object, or 0 if no parameter name was found.

Arguments

Data

An IDLitData object for which the parameter name is being requested.

Name

A named variable that will contain the parameter name associated with the object specified by the *oData* argument. If no parameter name is associated with the object specified by the *oData* argument, a null string (' ') is returned.

Keywords

None

Version History

Introduced: 6.0

IDLitParameterSet::Init

The IDLitParameterSet::Init function method initializes the IDLitParameter object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitParameterSet')
```

or

```
Result = Obj->[IDLitParameterSet:]Init() (Only in subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or 0 otherwise.

Arguments

None

Keywords

The IDLitParameterSet object has no properties of its own, but inherits the properties of its superclass, IDLitDataContainer. Any property listed under [“IDLitDataContainer Properties”](#) on page 3161 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitParameterSet::Remove

The IDLitParameterSet::Remove procedure method is used to remove a data item from the parameter set.

Syntax

```
Obj->[IDLitParameterSet::]Remove [, Items] [, /ALL] [, POSITION=integer]
```

Arguments

Items

An object reference or array of object references to IDLitData objects to be removed from the parameter set.

Keywords

ALL

Set this keyword to remove all items from the parameter set. If this keyword is set, the *Items* parameter is not required.

POSITION

Set this keyword equal to a scalar or array containing the zero-based indices within the parameter set of the specified IDLitData objects to be removed. If this keyword is set, the *Items* parameter is not required.

Version History

Introduced: 6.0

See Also

[IDLitParameterSet::Add](#)

IDLitReader

The IDLitReader class defines the interface used to construct file readers for the iTools framework. Objects of this class are not intended to be created as standalone entities; rather, this class should be included as the superclass of an iTool file reader class.

This class is written in the IDL language. Its source code can be found in the file `idlitreader__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitComponent](#)

[IDLitIMessaging](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitReader::Init](#)” on page 3338.

Properties

Objects of this class do not have any properties of their own, but do have properties inherited from any superclasses.

Methods

This class has the following methods:

- [IDLitReader::Cleanup](#)
- [IDLitReader::GetData](#)
- [IDLitReader::GetFileExtensions](#)
- [IDLitReader::GetFilename](#)
- [IDLitReader::GetProperty](#)

- [IDLitReader::Init](#)
- [IDLitReader::IsA](#)
- [IDLitReader::SetFilename](#)
- [IDLitReader::SetProperty](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [Chapter 9, “Creating a File Reader”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitReader Properties

Objects of this class do not have any properties of their own, but do have properties inherited from any superclasses.

IDLitReader::Cleanup

The IDLitReader::Cleanup procedure method performs all cleanup on the object, and should be called by the subclass' Cleanup method.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitReader::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitReader::GetData

The IDLitReader::GetData function method is called by the system to retrieve the data from the current file. When called, the reader should access the current filename and read the desired data. Once the information is loaded, the reader should place the obtained data in a data object, which is then passed back to the system.

This method will contain a majority of the implementation for a new reader class.

Syntax

Result = Obj->[IDLitReader::]GetData(Data)

Return Value

Returns a 1 if successful, or a 0 if the initialization failed.

Arguments

Data

A data object that contains the read information. The reader determines the exact type and format of this data object, which keys off the type of information being read and what the format supports.

Keywords

None

Version History

Introduced: 6.0

IDLitReader::GetFileExtensions

The IDLitReader::GetFileExtensions function method is called by the system to retrieve the file extensions supported by this particular reader.

Syntax

Result = *Obj*->[IDLitReader::]GetFileExtensions([**COUNT**=*variable*])

Return Value

Returns a scalar or string array that contains the file extensions associated with this reader.

Arguments

None

Keywords

COUNT

Set this keyword equal to a named variable that will contain the number of items returned by this function.

Version History

Introduced: 6.0

IDLitReader::GetFilename

The IDLitReader::GetFilename function method is called by the system to retrieve the current filename associated with this reader. Due to the automated nature of the file reader system, filenames can be associated with a file reader and then read at a later time. This method allows direct access to the file currently associated with the reader.

In addition, this methodology is helpful when multiple reads are performed from a given file.

Syntax

Result = *Obj*->[IDLitReader::]GetFilename()

Return Value

Returns a string containing the current file name associated with this reader, or an empty string if no filename has been associated.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitReader::GetProperty

The IDLitReader::GetProperty procedure method retrieves the value of an IDLitReader property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclasses.

Syntax

Obj->[IDLitReader::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitReader Properties](#)” on page 3332 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitReader::Init

The IDLitReader::Init function method initializes the IDLitReader object, and should be called by the subclass' Init method. This method also calls the superclass' Init method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitReader', *Extensions* [, *PROPERTY=value*])

or

Result = *Obj*->[IDLitReader::]Init(*Extensions* [, *PROPERTY=value*])

(Only in subclass' Init method.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Extensions

A scalar or string array containing the file extensions that are common for this file type. These values should not include the period that is often associated with file extensions (a correct value is “jpeg” not “.jpeg”).

Keywords

Any property listed under “[IDLitReader Properties](#)” on page 3332 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitReader::IsA

The IDLitReader::IsA function method is called by the system to determine if the given file is of the type supported by this file reader. Often this is used to determine what file reader to use when opening a new file.

The default behavior provided by this method is to check the file extension on the provided file name with the extensions provided to this object during initialization.

When a new reader is implemented, the default behavior can be used, or the developer can provide further logic to determine if the provided file is the correct type.

Syntax

Result = *Obj*->[IDLitReader::]IsA(*Filename*)

Return Value

Returns a 1 if the reader supports this type of file, or a 0 if the reader does not support this type of file.

Arguments

Filename

A string representing of the filename, which is used to check and determine if the reader supports its format.

Keywords

None

Version History

Introduced: 6.0

IDLitReader::SetFilename

The IDLitReader::SetFilename procedure method is called by the system to set the current filename associated with this reader. Due to the automated nature of the file reader system, filenames can be associated with a file reader and then the read at a later time.

Syntax

Obj->[IDLitReader::]SetFilename, *Filename*

Arguments

Filename

A string that contains the filename associated with this reader.

Keywords

None

Version History

Introduced: 6.0

IDLitReader:: SetProperty

The IDLitReader::SetProperty procedure method sets the value of an IDLitReader property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitReader::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under [“IDLitReader Properties”](#) on page 3332 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitTool

The IDLitTool class represents all the functionality provided by a particular instance of an IDL Intelligent Tool. It provides the management systems for the underlying iTool functionality and implements the desired interfaces for external parties to interact with this functionality in a known manner.

This class is written in the IDL language. Its source code can be found in the file `idlittool__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Note on the IDLitToolbase Class

All iTools included with the IDL release are based on a subclass of the IDLitTool class named IDLitToolbase. If your aim is to create a new iTool that shares the common features of the iTools included with IDL, you should base your new iTool on the IDLitToolbase class. Since IDLitToolbase is a subclass of IDLitTool, it inherits all of IDLitTool's methods and properties. Additional features included in the IDLitToolbase class are described in “[Subclassing from the IDLitToolbase Class](#)” in Chapter 5 of the *iTool Developer's Guide* manual.

Superclasses

[IDLitContainer](#)

[IDLitMessaging](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer's Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitTool::Init](#)” on page 3374.

Properties

Objects of this class have the following properties. See “[IDLitTool Properties](#)” on page 3347 for details on individual properties.

- [DESCRIPTION](#)

- [ICON](#)
- [NAME](#)
- [TYPE](#)
- [UPDATE_BYTYPE](#)
- [VERBOSE](#)
- [VERSION](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLitTool::ActivateManipulator](#)
- [IDLitTool::Add](#)
- [IDLitTool::AddService](#)
- [IDLitTool::Cleanup](#)
- [IDLitTool::CommitActions](#)
- [IDLitTool::DisableUpdates](#)
- [IDLitTool::DoAction](#)
- [IDLitTool::DoSetProperty](#)
- [IDLitTool::DoUIService](#)
- [IDLitTool::EnableUpdates](#)
- [IDLitTool::FindIdentifiers](#)
- [IDLitTool::GetCurrentManipulator](#)
- [IDLitTool::GetFileReader](#)
- [IDLitTool::GetFileWriter](#)
- [IDLitTool::GetManipulators](#)
- [IDLitTool::GetOperations](#)
- [IDLitTool::GetProperty](#)
- [IDLitTool::GetSelectedItems](#)
- [IDLitTool::GetService](#)

- `IDLitTool::GetVisualization`
- `IDLitTool::Init`
- `IDLitTool::RefreshCurrentWindow`
- `IDLitTool::Register`
- `IDLitTool::RegisterCustomization`
- `IDLitTool::RegisterFileReader`
- `IDLitTool::RegisterFileWriter`
- `IDLitTool::RegisterManipulator`
- `IDLitTool::RegisterOperation`
- `IDLitTool::RegisterStatusBarSegment`
- `IDLitTool::RegisterVisualization`
- `IDLitTool::SetProperty`
- `IDLitTool::UnRegister`
- `IDLitTool::UnRegisterCustomization`
- `IDLitTool::UnRegisterFileReader`
- `IDLitTool::UnRegisterFileWriter`
- `IDLitTool::UnRegisterManipulator`
- `IDLitTool::UnRegisterOperation`
- `IDLitTool::UnRegisterStatusBarSegment`
- `IDLitTool::UnRegisterVisualization`

In addition, this class inherits the methods of its superclasses.

Examples

See “[Example: Simple iTool](#)” in Chapter 5 of the *iTool Developer’s Guide* manual.

Version History

Introduced: 6.0

See Also

[Chapter 5, “Creating an iTool”](#) in the *iTool Developer’s Guide* manual

IDLitTool Properties

IDLitTool objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitTool::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitTool::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitTool::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

DESCRIPTION

This property contains a string representing the full name or description of this object.

Property Type	STRING		
Name String	Description		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ICON

A string identifying the icon that should be used when displaying the iTool object. See [“System Resources”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

NAME

A string representing the name of this object.

Property Type	STRING		
Name String	Name		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TYPE

A string or vector of strings that identify the type(s) that this iTool represents.

Property Type	String		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

UPDATE_BYTYPE

A Boolean value controlling whether the given iTool will add new functionality to it as new visualization types are added. By default, when a new visualization is added to the iTool, any functionality associated with that visualization type is added to the iTool. If this property is set to zero (False), this updating is disabled.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

VERBOSE

A named variable that will control the verbose logging of the iTool. If set to a non-zero value, the iTool will log more information than normal. If this keyword is set to 0, minimal logging (error conditions only) will be sent to the logging system.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

VERSION

A named variable that will be set to the version of the iTool system.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

IDLitTool::ActivateManipulator

Version History

The IDLitTool::ActivateManipulator procedure method activates a manipulator that has been registered with this tool.

Syntax

Obj->ActivateManipulator, Identifier [, /DEFAULT]

Arguments

Identifier

A string representing the identifier of the manipulator to activate.

Keywords

DEFAULT

Set this keyword to a non-zero value to indicate that the default manipulator registered with this tool should be activated. If this keyword is set, the Identifier argument is ignored.

Version History

Introduced: 6.1

IDLitTool::Add

The IDLitTool::Add procedure method adds any item to the iTool. If called with a Visualization, that visualization is added to the current IDLitWindow object. If called with a data object, that data object is placed in the systems data manager. All other items are added to the iTool container.

Syntax

Obj->[IDLitTool::]Add, *Item*

Arguments

Item

The item to add to the iTool.

Keywords

This method passes keywords to the Add method of the appropriate object. If the target destination is a data object, no keywords are passed. If the target destination is a visualization object, the keywords are passed to the IDLitWindow::Add method; otherwise, the keywords are passed to the IDLitContainer::Add method.

Version History

Introduced: 6.0

IDLitTool::AddService

The `IDLitTool::AddService` procedure method adds a service to the `iTool`. Unlike registration methodologies provided by the `iTool`, the method takes an existing object and adds it to the service repository in the `iTool`.

When a service is added to the system, a description of this service class is placed in the tools service folder. If it is desired to retrieve this descriptor at a later time, the `iTool` relative identifier would be `SERVICES/identifier`, where `identifier` is the identifier provided to this method.

Syntax

Obj->[IDLitTool::]AddService, *Service*

Arguments

Service

Set this variable to the service to add to the tools service repository. This service can be retrieved at a later point in time using its identifier (specified when creating the service object via the identifier keyword).

Keywords

None

Version History

Introduced: 6.0

IDLitTool::Cleanup

The IDLitTool::Cleanup procedure method performs all cleanup on the object, and should be called by the Cleanup method of a subclass.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitTool:]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitTool::CommitActions

The IDLitTool::CommitActions procedure method commits all pending transactions to the undo-redo buffer and causes a refresh of the current window.

This method is designed to be used in conjunction with the [IDLitTool::DoSetProperty](#) method.

Syntax

Obj->[IDLitTool::]CommitActions

Arguments

None

Keywords

None

Version History

Introduced: 6.0

See Also

[IDLitTool::DoSetProperty](#)

IDLitTool::DisableUpdates

The IDLitTool::DisableUpdates procedure method disables all drawing updates to the current window and UI updates (menu sensitivity) being passed to the user interface. After this method is called, updates are re-enabled by calling the EnableUpdates method.

Syntax

Obj->[IDLitTool::]DisableUpdates ([, PREVIOUSLY_DISABLED=*variable*])

Arguments

None

Keywords

PREVIOUSLY_DISABLED

Set this keyword to a named variable that upon return will be 1 if the tool had previously been disabled, or 0 otherwise.

Version History

Introduced: 6.0

PREVIOUSLY_DISABLED: Introduced 6.1

IDLitTool::DoAction

The IDLitTool::DoAction function method initiates an operation or action in the iTool object. The identifier that is passed in as the argument to this method specifies the action that executes.

Syntax

Result = *Obj*->[IDLitTool::]DoAction(*Identifier*)

Return Value

Returns 1 if the action succeeds, or 0 if the action fails.

Arguments

Identifier

This argument is set to a string identifier that specifies the object in the iTool that performs the operation that it implements.

Keywords

All keywords are passed to the DoAction function of the object in the iTool that performs the operation.

Version History

Introduced: 6.0

IDLitTool::DoSetProperty

The IDLitTool::DoSetProperty function method sets the value of a registered property on a target component object, and places the change in the undo/redo transaction buffer.

Note

In order to commit the change to the undo/redo transaction buffer, you must call the [IDLitTool::CommitActions](#) method. The property change is not actually undoable until it has been committed.

The separation of addition of the change to the buffer and commit of the change allows you to “package” several property changes into a single undo/redo transaction. The call to the CommitActions method will commit any changes that have been performed but not committed. For example, the following statements change two properties on an iTool component (oTarget) and commit them together:

```
targetID = oTarget->GetFullIdentifier()
status = oTool->DoSetProperty(targetID, 'Property1', 'Value 1')
status = oTool->DoSetProperty(targetID, 'Property2', 'Value 2')
oTool->CommitActions
```

If the user later undoes the action, both property value changes will be reversed in one step.

Syntax

Result = Obj->[IDLitTool::]DoSetProperty(TargetIdentifier, PropertyIdentifier, Value)

Return Value

Returns 1 if the SetProperty was successful, or 0 otherwise.

Note

If successful, you are responsible for calling IDLitTool::CommitActions to complete the transaction.

Arguments

TargetIdentifier

A scalar string or a string array containing the full object identifiers of the objects on which the property is being set.

PropertyIdentifier

A scalar string containing the identifier of the property whose value is being changed. *PropertyIdentifier* must be a registered property of the target objects.

Value

The new property value.

Keywords

None

Version History

Introduced: 6.0

See Also

[IDLitTool::CommitActions](#)

IDLitTool::DoUIService

The IDLitTool::DoUIService function method initiates a request for a UI service to execute. This method allows underlying iTool functionality to perform user-interface interactions without having a direct connection to the user interface portion of the iTool.

An example of an iTool UI service would be the **File** → **Open** command, which will prompt the user to select a file using the standard file open dialog. This command is identified by setting the *ServiceIdentifier* argument to FileOpen. See [Chapter 13, “Creating a User Interface Service”](#) in the *iTool Developer’s Guide* manual for more information about UI services.

Syntax

Result = *Obj*->[IDLitTool:]DoUIService(*ServiceIdentifier*, *Requestor*)

Return Value

Returns 1 if the action succeeds, or 0 if the action fails.

Arguments

ServiceIdentifier

A string identifier that specifies the UI service being requested. This identifier is passed to the tools user interface for control dispatching purposes.

Requestor

An object reference to the object that is requesting the UI service. The UI service uses this object to obtain any information particular to the interaction taking place.

Keywords

None

Version History

Introduced: 6.0

IDLitTool::EnableUpdates

The IDLitTool::EnableUpdates procedure method re-enables all drawing updates to the current window and UI updates (menu sensitivity) being passed to the user interface. After this method is called, updates can be disabled by calling the DisableUpdates method.

Syntax

Obj->[IDLitTool]::EnableUpdates

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitTool::FindIdentifiers

The IDLitTool::FindIdentifiers function method allows you to retrieve the full identifiers for items within the tool container.

Syntax

```
Result = Obj->IDLitTool::FindIdentifiers([Pattern] [, /ANNOTATIONS]  
[, /COUNT=variable] [, /DATA_MANAGER] [, /FILE_READERS]  
[, /FILE_WRITERS] [, /LEAF_NODES] [, /MANIPULATORS]  
[, /OPERATIONS] [, /VISUALIZATIONS] )
```

Return Value

Returns a scalar string or string array containing the full object identifiers for all items matching the *Pattern*. If no object identifiers are matched, the function returns a scalar null string.

Arguments

Pattern

A string specifying a search pattern to be used for filtering the returned object identifiers. Only object identifiers that match *Pattern* are returned. Matching is case insensitive. If *Pattern* is not supplied, identifiers for all objects within the object on which this method is called are returned. See [STRMATCH](#) for a description of the rules used when matching patterns. See “[Retrieving Component Identifiers](#)” in Appendix A of the *iTool Developer’s Guide* manual for additional information and examples.

Note

Objects for which the PRIVATE property contains a True value are not included in the *Result*. Only objects that subclass from IDLitComponent can be marked as private.

Keywords

ANNOTATIONS

Set this keyword to only return identifiers for items within the annotation layer of all views within the graphics window.

COUNT

Set this keyword equal to a named variable that will contain the number of identifiers in *Result*.

DATA_MANAGER

Set this keyword to only return identifiers for items within the Data Manager container.

FILE_READERS

Set this keyword to only return identifiers for items within the File Readers container.

FILE_WRITERS

Set this keyword to only return identifiers for items within the File Writers container.

LEAF_NODES

Set this keyword to return only leaf nodes. Leaf nodes are objects which are either not a container, or are an empty container.

Note

If all of the objects within a container are marked as PRIVATE, then that container is considered a leaf node. Only objects that subclass from IDLitComponent can be marked as private.

MANIPULATORS

Set this keyword to only return identifiers for items within the Manipulators container.

OPERATIONS

Set this keyword to only return identifiers for items within the Operations container.

VISUALIZATIONS

Set this keyword to only return identifiers for items within the Visualization layer of all views within the graphics window.

Example

This example finds the plot visualization and changes the color:

```
IPLOT, RANDOMU(s,100)
void = ITGETCURRENT(TOOL=oTool)

; Find the Plot visualization and change the color.
id = oTool->FindIdentifiers('*plot*', /VISUALIZATIONS)

void = oTool->DoSetProperty(id, 'COLOR', [255,0,0])
```

This example finds the file open operation and loads in an image:

```
IIMAGE
void = ITGETCURRENT(TOOL=oTool)

; Find the File Open operation and open a file.
id = oTool->FindIdentifiers('*File/Open*', /OPERATIONS)

oOp = oTool->GetByIdentifier(id)
oOp->SetProperty, SHOW_EXECUTION_UI=0, $
    FILENAMES=FILEPATH('rose.jpg', SUBDIR=['examples','data'])
void = oTool->DoAction(id)
oOp->SetProperty, SHOW_EXECUTION_UI=1
```

Version History

Introduced: 6.1

See Also

[“Retrieving Component Identifiers”](#) in Appendix A of the *iTool Developer’s Guide* manual

IDLitTool::GetCurrentManipulator

The IDLitTool::GetCurrentManipulator function method returns the current manipulator in the system.

Syntax

Result = *Obj*->[IDLitTool]::GetCurrentManipulator()

Return Value

Returns an object reference to the current manipulator.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitTool::GetFileReader

The IDLitTool::GetFileReader function method retrieves a file reader registered with the iTool object. If the requested item does not exist in the given iTool, the system will be queried to see if it supports an object of the given identifier.

Syntax

```
Result = Obj->[IDLitTool:]GetFileReaders(Identifier [, /ALL]  
[, COUNT=variable])
```

Return Value

Returns the object descriptors constructed for the registered file reader.

If the ALL keyword is set, the *Identifier* argument is ignored and the resulting list of object descriptors will be both those registered with the specific iTool and those file readers registered with the system.

Arguments

Identifier

A string representing the single level identifier of the item that is being requested. This identifier is the value provided during object registration and is either supplied using the IDENTIFIER keyword during registration, or constructed automatically by the name that is provided during the registration call.

Keywords

ALL

Set this keyword to return all file readers registered with the iTool and the system.

COUNT

Set this keyword to a named variable that will contain the number of object descriptors returned by this method.

Version History

Introduced: 6.0

IDLitTool::GetFileWriter

The IDLitTool::GetFileWriter function method retrieves a file writer registered with the iTool object. If the requested item does not exist in the given iTool, the system will be queried to see if it supports an object of the given identifier.

Syntax

```
Result = Obj->[IDLitTool:]GetFileWriters(Identifier [, /ALL]  
[, COUNT=variable])
```

Return Value

Returns the object descriptors constructed for the registered file writer.

If the ALL keyword is set, the *Identifier* argument is ignored and the resulting list of object descriptors will be both those registered with the specific iTool and those file writers registered with the system.

Arguments

Identifier

A string representing the identifier of the item that is being requested. This identifier is the value provided during object registration and is either supplied using the IDENTIFIER keyword during registration, or constructed automatically by the name that is provided during the registration call.

Keywords

ALL

Set this keyword to return all file writers registered with the iTool and the system.

COUNT

Set this keyword to a named variable that will contain the number of object descriptors returned by this method.

Version History

Introduced: 6.0

IDLitTool::GetManipulators

The IDLitTool::GetManipulators function method retrieves the manipulators registered with the iTool object. Unlike other items in the iTool and due to the active nature of the subsystem, manipulators are instantiated when registered.

Syntax

Result = Obj->[IDLitTool::]GetManipulators([**COUNT**=*variable*])

Return Value

Returns the object references to the manipulators that were instantiated when items were registered with the iTool.

Arguments

None

Keywords

COUNT

Set this keyword to a named variable that will contain the number of object descriptors returned by this method.

Version History

Introduced: 6.0

IDLitTool::GetOperations

The IDLitTool::GetOperations function method retrieves the operations registered with the iTool object.

Note

This routine returns a set of object descriptor classes that reference the items registered with the iTool. In addition, if the operation folder in the registry contains sub-folders, the returned list will contain IDLitContainer classes, each containing sub-operations.

Syntax

```
Result = Obj->[IDLitTool:]GetOperations([, COUNT=variable]  
[, IDENTIFIER=string])
```

Return Value

Returns the object descriptors constructed for the registered operations.

Arguments

None

Keywords

COUNT

Set this keyword to a named variable that will contain the number of object descriptors returned by this method.

IDENTIFIER

Set this keyword to an optional identifier that identifies the location in the operation hierarchy to retrieve the operation object descriptors from.

Version History

Introduced: 6.0

IDLitTool::GetProperty

The IDLitTool::GetProperty procedure method retrieves the value of an IDLitTool property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclasses.

Syntax

Obj->[IDLitTool::]GetProperty, [*PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitTool Properties](#)” on page 3347 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitTool::GetSelectedItems

The IDLitTool::GetSelectedItems function method returns a vector of references to the objects currently selected within the current window in the iTool. If no objects are currently selected, this function returns -1.

Syntax

Result = Obj->[IDLitTool:]GetSelectedItems([COUNT=variable])

Return Value

Returns a vector of references to the currently selected objects. If no objects are currently selected, this function returns -1.

Arguments

None

Keywords

COUNT

Set this keyword to a named variable that will contain the number of valid items returned by this function.

Version History

Introduced: 6.0

IDLitTool::GetService

The IDLitTool::GetService method retrieves a service that has been registered with the iTool.

Syntax

Result = *Obj*->[IDLitTool:]GetService(*IdService*)

Return Value

Returns an object reference to the service.

Arguments

IdService

The identifier of the desired service.

Note

IdService should specify only the relative identifier, not the fully-qualified identifier. See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for details.

Keywords

None

Version History

Introduced: 6.0

IDLitTool::GetVisualization

The IDLitTool::GetVisualization function method retrieves a visualization registered with the iTool object. If the requested item does not exist in the given iTool, the system will be queried to see if it supports an object of the given identifier.

Syntax

```
Result = Obj->[IDLitTool::]GetVisualizations(Identifier [, /ALL]  
[, COUNT=variable])
```

Return Value

Returns the object descriptors constructed for the registered visualizations. (See “iTool Object Identifiers” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object descriptors.)

If the ALL keyword is set, the *Identifier* argument is ignored and the resulting list of object descriptors will be both those registered with the specific iTool as well as those visualizations registered with the system.

Arguments

Identifier

A string containing the relative identifier of the item that is being requested. This identifier is the value provided during object registration and is either supplied using the IDENTIFIER keyword during registration, or constructed automatically by the name that is provided during the registration call. (See “iTool Object Identifiers” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of relative vs. fully-qualified identifiers.)

Keywords

ALL

Set this keyword to return all visualizations registered with the iTool and the system.

COUNT

Set this keyword to a variable that will contain the number of object descriptors returned by this method. (See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object descriptors.)

Version History

Introduced: 6.0

IDLitTool::Init

The IDLitTool::Init function method initializes the IDLitTool object, and should be called by the Init method of a subclass.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitTool'[, *PROPERTY=value*])

or

Result = *Obj*->[IDLitTool::]Init([*PROPERTY=value*])
(Only in a subclass' Init method.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under [“IDLitTool Properties”](#) on page 3347 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitTool::RefreshCurrentWindow

The IDLitTool::RefreshCurrentWindow procedure method redraws the current window of the iTool. The redraw will occur only if updates are enabled in the iTool.

Syntax

Obj->[IDLitTool]::RefreshCurrentWindow

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitTool::Register

The `IDLitTool::Register` procedure method registers a generic object subclassed from the `IDLitComponent` class with the `iTool`. Once registered, the `iTool` has the ability to instantiate this type of component when needed. Unlike the other provided methods to register components, this method is generic.

This method allows items to be registered in the standard registry folders of the `IDLitTool` object, and can be used as well for the creation of new folders. If the provided identifier specifies a folder that does not currently exist in the `iTool`, it is created.

Note

For information on registering functionality with the `iTool` system, making the registered functionality available to all `iTools`, see [ITREGISTER](#).

This method performs none of the validation performed by the specific registration functions. As such, it can cause issues with `iTool` operation, if used incorrectly. You should use the following related methods for registering certain, specific components to insure validation:

- File readers — “[IDLitTool::RegisterFileReader](#)” on page 3381.
- File writers — “[IDLitTool::RegisterFileWriter](#)” on page 3383.
- Manipulators — “[IDLitTool::RegisterManipulator](#)” on page 3386.
- Operations — “[IDLitTool::RegisterOperation](#)” on page 3388.
- Visualizations — “[IDLitTool::RegisterVisualization](#)” on page 3393

Syntax

```
Obj->[IDLitTool::]Register, Name, ClassName [, /DEFAULT]  
    [, DESCRIPTION=string] [, ICON=string] [, IDENTIFIER=string]  
    [, PROXY=string]
```

Arguments

Name

A string containing the human readable name of the component type being registered with the `iTool`.

ClassName

A string containing the class name of the component being registered. This class name is used by the iTool object to instantiate a component of this type when requested by the system.

Keywords

Note

Any keywords provided to this routine but not listed here are treated as property defaults for the registered item and applied to the registered object when it is created.

DEFAULT

Set this keyword to specify that the item being registered should be the *default item* of its type. Making an item the default places the item first in the list of items of its type. When the iTool system chooses an item, it will use the first item in the list that matches the data type of the selected data. When items are displayed in a list for the user to select from, the default item will appear at the top of the list.

For example, if you register a visualization type with the DEFAULT keyword set, your visualization type will be used by the iTool (assuming the selected data is of the proper type) unless you specifically choose a different visualization type.

Note

If you register more than one item with the DEFAULT keyword set, the *last* item registered will appear *first* in the list of items of that type.

DESCRIPTION

Set this keyword to a string that provides a brief description of the component being registered.

ICON

Set this keyword to the icon that should be used when displaying the iTool object. See [“System Resources”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

IDENTIFIER

Set this keyword to the identifier that should be used by the system for this visualization type. If not provided, an identifier is constructed from the provided *Name* value.

To use one of the iTool-provided registration locations, the value of this keyword should contain the destination for the new component that is being registered. Valid destinations in the iTool are:

- Visualizations
- Operations
- Manipulators
- File Writers
- File Readers

PROXY

Set this keyword to the identifier of the object that this registered item should utilize instead of providing a new object descriptor. If a proxy is registered, all requests and actions performed to this item are routed to the item specified by the identifier provide to this keyword. (See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object descriptors.)

Version History

Introduced: 6.0

DEFAULT keyword added: 6.1

IDLitTool::RegisterCustomization

Version History

The IDLitTool::RegisterCustomization procedure method registers an operation class that represents the graphics customization operation to be associated with this tool.

Syntax

Obj->[IDLitTool::]RegisterCustomization, *Name*, *ClassName*

Arguments

Name

A string representing the name of the customization operation.

ClassName

A string representing the classname of the operation being registered.

Keywords

Note

Any keywords provided to this routine are treated as property defaults for the registered operation and applied to the registered object when it is created.

Version History

Introduced: 6.1

IDLitTool::RegisterFileReader

The IDLitTool::RegisterFileReader procedure method registers a file reader component with the iTool. Once registered, the iTool has the ability to instantiate this type of component when needed.

When a reader class is registered with the iTool, an object descriptor for this operation class is placed in the iTool's file reader folder. To retrieve this descriptor at a later time, the iTool relative identifier would be `FILE_READERS/identifier`, where `identifier` is the identifier provided to this method.

Syntax

```
Obj->[IDLitTool::]RegisterFileReader, Name, ClassName [, /DEFAULT]
    [, DESCRIPTION=string] [, ICON=string] [, IDENTIFIER=string]
    [, PROXY=string]
```

Arguments

Name

The human readable name of the component type being registered with the iTool.

ClassName

A string containing the class name of the component being registered. This class name is used by the iTool object to instantiate a component of this type when requested by the system.

Keywords

Note

Any keywords provided to this routine but not listed here are treated as property defaults for the registered item and applied to the registered object when it is created.

DEFAULT

Set this keyword to specify that the file reader being registered should be the *default* file reader. When the iTool system picks a reader for a file of the specified type, it will attempt to use the specified file reader first. The file reader will also be displayed at the top of the list of file readers displayed for the user to select from.

Note

If you register more than one file reader with the DEFAULT keyword set, the *last* file reader registered will appear *first* in the list of items of file readers.

DESCRIPTION

Set this keyword to a string that provides a brief description of the component being registered.

ICON

Set this keyword to the icon that should be used when displaying the file reader in an iTool browser. See “[System Resources](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

IDENTIFIER

Set this keyword to the identifier that should be used by the system for this visualization type. If not provided, an identifier is constructed from the provided *Name* value.

While some items registered with an iTool can contain sub-folders that are specified by identifiers, the file reader registration cannot. As such, any nested identifiers that are registered are not visible to the iTool system.

PROXY

Set this keyword to the identifier of the object that this registered item should utilize instead of providing a new object descriptor. If a proxy is registered, all requests and actions performed on this item are routed to the item specified by the identifier provided for this keyword. (See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object descriptors.)

Version History

Introduced: 6.0

DEFAULT keyword added: 6.1

See Also

[ITREGISTER](#)

IDLitTool::RegisterFileWriter

The IDLitTool::RegisterFileWriter procedure method registers a file writer component with the iTool. Once registered, the iTool has the ability to instantiate this type of component when needed.

When a writer class is registered with the system, an object descriptor for this operation class is placed in the iTool's file writer folder. To retrieve this descriptor at a later time, the iTool relative identifier would be `FILE WRITERS/identifier`, where `identifier` is the identifier provided to this method.

Syntax

```
Obj->[IDLitTool::]RegisterFileWriter, Name, ClassName [, /DEFAULT]
    [, DESCRIPTION=string] [, ICON=string] [, IDENTIFIER=string]
    [, PROXY=string]
```

Arguments

Name

A string containing the human readable name of the component type being registered with the iTool.

ClassName

A string containing the class name of the component being registered. The iTool object uses this class name to instantiate a component of this type when requested by the system.

Keywords

Note

Any keywords provided to this routine but not listed here are treated as property defaults for the registered item and applied to the registered object when it is created.

DEFAULT

Set this keyword to specify that the file writer being registered should be the *default* file writer. When the iTool system picks a writer for a file of the specified type, it will

attempt to use the specified file writer first. The file writer will also be displayed at the top of the list of file writers displayed for the user to select from.

Note

If you register more than one file writer with the DEFAULT keyword set, the *last* file writer registered will appear *first* in the list of file writers.

DESCRIPTION

Set this keyword to a string that provides a brief description of the component being registered.

ICON

Set this keyword to the icon that should be used when displaying the file writer in an iTool browser. See “[System Resources](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

IDENTIFIER

Set this keyword to the identifier that should be used by the system for this visualization type. If not provided, an identifier is constructed from the provided *Name* value.

While some items registered with an iTool can contain sub-folders that are specified by identifiers, the file writer registration cannot. As such, any nested identifiers that are registered are not visible to the iTool system.

PROXY

Set this keyword to the identifier of the object that this registered item should utilize instead of providing a new object descriptor. If a proxy is registered, all requests and actions performed on this item are routed to the item specified by the identifier provided for this keyword. (See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object descriptors.)

Version History

Introduced: 6.0

DEFAULT keyword added: 6.1

See Also

[ITREGISTER](#)

IDLitTool::RegisterManipulator

The IDLitTool::RegisterManipulator procedure method registers a manipulator component with the iTool. Once registered, the iTool has the ability to instantiate this type of component when needed.

When a manipulator class is registered with the system, an object of the given class is created and added to the tools manipulator manager (an instance of IDLitManipulatorManager). To reference the new manipulator, the identifier that is provided should be prepended with the MANIPULATORS identifier level.

Syntax

```
Obj->[IDLitTool::]RegisterManipulator, Name, ClassName [, /DEFAULT]  
    [, DESCRIPTION=string] [, ICON=string] [, IDENTIFIER=string]
```

Arguments

Name

A string containing the human readable name of the component type being registered with the iTool.

ClassName

A string containing the class name of the component being registered. This class name is used by the iTool object to instantiate a component of this type when requested by the system.

Keywords

Note

Any keywords provided to this routine but not listed here are treated as property defaults for the registered item and applied to the registered object when it is created.

DEFAULT

If set, this registered item will be treated as the default manipulator for this iTool. Only one default manipulator can exist at anytime.

DESCRIPTION

Set this keyword to a string that provides a brief description of the component being registered.

ICON

Set this keyword to the icon that should be used when displaying the manipulator in an iTool menubar. See “[System Resources](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

IDENTIFIER

Set this keyword to the identifier that should be used by the system for this visualization type. If not provided, an identifier is constructed from the provided name value.

Version History

Introduced: 6.0

IDLitTool::RegisterOperation

The `IDLitTool::RegisterOperation` procedure method registers an operation component with the `iTool`. Once registered, the `iTool` has the ability to instantiate this type of operation when needed.

When an operation class is registered with the system, an object descriptor for this operation class is placed in the tools operations folder. To retrieve this descriptor at a later time, the `iTool` relative identifier would be `OPERATIONS/identifier`, where `identifier` is the identifier provided to this method.

Syntax

```
Obj->[IDLitTool::]RegisterOperation, Name, ClassName
    [, ACCELERATOR=string] [, /CHECKED] [, DESCRIPTION=string]
    [, /DISABLE] [, /DROPLIST_EDIT] [, DROPLIST_INDEX=value]
    [, DROPLIST_ITEMS=string array] [, ICON=string] [, IDENTIFIER=string]
    [, PROXY=string] [, /SEPARATOR]
```

Arguments

Name

A string containing the human readable name of the operation type being registered with the `iTool`.

ClassName

A string containing the class name of the operation component being registered. This class name is used by the `iTool` object to instantiate an operation component of this type when requested.

Keywords

Note

Any keywords provided to this routine but not listed here are treated as property defaults for the registered item and applied to the registered object when it is created.

ACCELERATOR

Set this keyword to a string giving the keyboard accelerator to be used for the menu item associated with this operation. For valid values see the ACCELERATOR keyword to WIDGET_BUTTON.

Note

This keyword is only a hint to the Tool, and may be ignored if a non-standard user interface is being used.

CHECKED

Set this keyword to indicate that a "checked" menu item should be used for this operation.

Note

This keyword is only a hint to the Tool, and may be ignored if a non-standard user interface is being used.

DESCRIPTION

Set this keyword to a string that provides a brief description of the component being registered.

DISABLE

Set this keyword to indicate that the menu item associated with this operation should appear disabled (insensitive) when initially created.

Note

This keyword is only a hint to the Tool, and may be ignored if a non-standard user interface is being used.

DROPLIST_EDIT

Set this keyword to indicate that a droplist associated with this operation should be editable (that is, the droplist should be a combobox widget). The default is to create a non-editable droplist widget. The DROPLIST_ITEMS keyword must be set for this keyword to have any effect.

Note

This keyword is only a hint to the *iTool*, and may be ignored if a non-standard user interface is being used.

DROPLIST_INDEX

Set this keyword equal to the starting index of the droplist associated with this operation. The default is 0 (zero). The `DROPLIST_ITEMS` keyword must be set for this keyword to have any effect.

Note

This keyword is only a hint to the *iTool*, and may be ignored if a non-standard user interface is being used.

DROPLIST_ITEMS

Set this keyword equal to a scalar string or string array containing values to be used for the droplist. If this keyword is set, the `DoAction` method for the associated operation must accept a keyword called `OPTION`. When an item from the droplist is selected, the `DoAction` method will be called with the `OPTION` keyword set equal to the selected string value.

Note

This keyword is only a hint to the *iTool*, and may be ignored if a non-standard user interface is being used.

ICON

Set this keyword to the icon that should be used when displaying the operation in an *iTool* browser. See “[System Resources](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

IDENTIFIER

Set this keyword to the identifier for the *iTool* for this component type. If not provided, an identifier is constructed from the *Name*.

Operations are placed in the `operation` folder of the *iTool* component hierarchy. If a multiple level identifier is provided (indicated by the presence of one or more “/” characters), the item is placed in the corresponding subfolder of the tool’s `operation` folder. If the specified subfolder is not present, it is created. This

functionality allows operations to be classified in logical groupings and results in sub-menu items in the menus of the iTool interface.

See “[iTool Object Hierarchy](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object identifiers and the iTool component hierarchy.

PROXY

Set this keyword to the identifier of the object that this registered item should utilize instead of providing a new object descriptor. If a proxy is registered, all requests and actions performed on this item are routed to the item specified by the identifier provided for this keyword. (See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of object descriptors.)

SEPARATOR

Set this keyword to indicate that a menu separator should be placed before this operation, if this operation appears in a menu.

Note

This keyword is only a hint to the Tool, and may be ignored if a non-standard user interface is being used.

Version History

Introduced: 6.0

ACCELERATOR, CHECKED, DISABLE, DROPLIST_EDIT, DROPLIST_INDEX, DROPLIST_ITEMS, SEPARATOR keywords added: 6.1

IDLitTool::RegisterStatusBarSegment

The IDLitTool::RegisterStatusBarSegment procedure method registers a status message bar segment with the tool.

Syntax

```
Obj->[IDLitTool:]RegisterStatusBarSegment, Name [, IDENTIFIER=string]  
[, NORMALIZED_WIDTH=scalar]
```

Arguments

Name

A scalar string containing the name of the status bar segment.

Keywords

IDENTIFIER

Set this keyword to a string containing the relative identifier for the status bar segment being registered. If this keyword is not specified, the *Name* argument is used as the identifier. See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

NORMALIZED_WIDTH

Set this keyword to a floating-point integer between 0 and 1.0 indicating the normalized width (relative to the overall width of the status bar) that this segment should occupy.

Version History

Introduced: 6.1

See Also

[IDLitTool::UnRegisterStatusBarSegment](#)

IDLitTool::RegisterVisualization

The IDLitTool::RegisterVisualization procedure method registers a visualization component with the iTool. Once registered, the iTool has the ability to instantiate this type of visualization when needed.

When a visualization class is registered with the system, an object descriptor for this visualization class is placed in the iTool's visualization folder. To retrieve this descriptor at a later time, the iTool relative identifier would be:

VISUALIZATIONS/*identifier* where *identifier* is the identifier provided to this method.

Syntax

```
Obj->[IDLitTool::]RegisterVisualization, Name, ClassName [, /DEFAULT]
    [, DESCRIPTION=string] [, ICON=string] [, IDENTIFIER=string]
    [, PROXY=string]
```

Arguments

Name

A string containing the human readable name of the visualization type being registered with the iTool.

ClassName

A string containing the class name of the visualization component being registered. The iTool object uses this class name to instantiate a visualization component of this type when requested.

Keywords

Note

Any keywords provided to this routine but not listed here are treated as property defaults for the registered item and applied to the registered object when it is created.

DEFAULT

If set, this registered item will be treated as the default visualization for this iTool. Only one default visualization can exist at a time.

DESCRIPTION

Set this keyword to a string that provides a brief description of the component being registered.

ICON

Set this keyword to the icon that should be used when displaying the visualization in an iTool browser. See “[System Resources](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for additional details.

IDENTIFIER

Set this keyword to the identifier that should be used by the system for this visualization type. If not provided, an identifier is constructed from the provided name value.

While some items registered with an iTool can contain sub-folders that are specified by identifiers, the visualization registration cannot. As such, any nested identifiers that are registered are not visible to the iTool system.

PROXY

Set this keyword to the identifier of the object that this registered item should utilize instead of providing a new object descriptor. If a proxy is registered, all requests and actions performed on this item are routed to the item specified by the identifier provided for this keyword.

Version History

Introduced: 6.0

See Also

[ITREGISTER](#)

IDLitTool:: SetProperty

The IDLitTool::SetProperty procedure method sets the value of an IDLitTool property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitTool::]SetProperty[, *PROPERTY=**value*]

Arguments

None

Keywords

Any property listed under “[IDLitTool Properties](#)” on page 3347 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

See Also

[IDLitTool::DoSetProperty](#)

IDLitTool::UnRegister

The IDLitTool::UnRegister procedure method unregisters a component with the iTool. When a component is unregistered, it is no longer available to perform whatever function it performs, and it is removed from iTool menus and toolbars.

Syntax

Obj->[IDLitTool::]UnRegister, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the component to be removed from the iTool. To remove a component from the standard folders in the iTool's content registry, this identifier should contain the location of the component as well as the component identifier itself. See “[iTool Object Identifiers](#)” in Chapter 2 of the *iTool Developer's Guide* manual for a discussion of iTool identifiers.

Valid destinations in the iTool are:

- Visualizations
- Operations
- Manipulators
- File Writers
- File Readers

Keywords

None

Version History

Introduced: 6.0

IDLitTool::UnRegisterCustomization

The IDLitTool::UnRegisterCustomization procedure method unregistered an operation class (that was previously registered as the graphics customization operation to be associated with this tool).

Syntax

Obj->[IDLitTool::]UnRegisterCustomization

Keywords

None

Arguments

None

Version History

Introduced: 6.1

IDLitTool::UnRegisterFileReader

The IDLitTool::UnRegisterFileReader procedure method unregisters a file reader component with the iTool. When a file reader is unregistered, it is no longer available to import files into the iTool, and it is removed from iTool menus and toolbars.

Syntax

Obj->[IDLitTool::]UnRegisterFileReader, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the file reader to be removed from the iTool. See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

Keywords

None

Version History

Introduced: 6.0

IDLitTool::UnRegisterFileWriter

The IDLitTool::UnRegisterFileWriter procedure method unregisters a component with the iTool. When a file writer is unregistered, it is no longer available to export files from the iTool, and it is removed from iTool menus and toolbars.

Syntax

Obj->[IDLitTool::]UnRegisterFileWriter, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the file writer to be removed from the iTool. See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

Keywords

None

Version History

Introduced: 6.0

IDLitTool::UnRegisterManipulator

The IDLitTool::UnRegisterManipulator procedure method unregisters a manipulator component with the iTool. When a manipulator is unregistered, it is removed from iTool menus and toolbars.

Syntax

Obj->[IDLitTool::]UnRegisterManipulator, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the manipulator to be removed from the iTool. See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

Keywords

None

Version History

Introduced: 6.0

IDLitTool::UnRegisterOperation

The IDLitTool::UnRegisterOperation procedure method unregisters an operation component with the iTool. When an operation is unregistered, it is no longer available to perform actions within the iTool, and it is removed from iTool menus and toolbars.

Syntax

Obj->[IDLitTool::]UnRegisterOperation, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the component to be removed from the iTool. See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

Keywords

None

Version History

Introduced: 6.0

IDLitTool::UnRegisterStatusBarSegment

The IDLitTool::UnRegisterStatusBarSegment procedure method unregisters a status message bar segment.

Syntax

Obj->[IDLitTool:]UnRegisterStatusBarSegment, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the status bar segment to be unregistered. See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

Keywords

None

Version History

Introduced: 6.1

See Also

[IDLitTool::RegisterStatusBarSegment](#)

IDLitTool::UnRegisterVisualization

The IDLitTool::UnRegisterVisualization procedure method unregisters a visualization component with the iTool. When a visualization is unregistered, visualizations of its type can no longer be created by the iTool, and it is removed from iTool menus and toolbars.

Syntax

Obj->[IDLitTool:]UnRegisterVisualization, *Identifier*

Arguments

Identifier

Set this to the relative identifier of the visualization to remove from the iTool. See [“iTool Object Identifiers”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a discussion of iTool identifiers.

Keywords

None

Version History

Introduced: 6.0

IDLitToolbase

The IDLitToolbase class is a subclass of the IDLitTool class that *registers* all of the features (menu items, operations, manipulators, *etc.*) of a standard iTool provided by RSI. If your aim is to create a new iTool that shares the common features of the iTools included with IDL, you should base your new iTool on the IDLitToolbase class. Since IDLitToolbase is a subclass of IDLitTool, it inherits all of IDLitTool's methods and properties. Additional features included in the IDLitToolbase class are described in “[Subclassing from the IDLitToolbase Class](#)” in Chapter 5 of the *iTool Developer's Guide* manual.

Warning

RSI may change the list of items registered by the IDLitToolbase class from release to release. If your aim is to have your tool present the same features as the standard iTools provided by RSI, subclass your tool from IDLitToolbase. If your aim is to create your own interface that stays static between releases of IDL, use IDLitToolbase as a template for your own tool, but do not subclass it directly.

This class is written in the IDL language. Its source code can be found in the file `idlittoolbase__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitTool](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer's Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitTool::Init](#)” on page 3374.

Properties

Objects of this class have no properties of their own, but do have properties inherited from any superclasses.

Methods

Objects of this class have no methods of their own, but do have methods inherited from any superclasses.

Examples

See [“Example: Simple iTool”](#) in Chapter 5 of the *iTool Developer’s Guide* manual.

Version History

Introduced: 6.0

See Also

[Chapter 5, “Creating an iTool”](#) in the *iTool Developer’s Guide* manual

IDLitUI

The IDLitUI class serves as the link between the underlying functionality of an iTool and the IDL Widget interface that is displayed to the user. The user interface object provides the following functionality:

- Access to and communication with the underlying iTool object.
- Registration of various user interface elements that are part of the iTool.
- Registration and management of dialog sub-elements that are used by the iTool to perform specific tasks.

For additional information on the IDLitUI class, and user interface issues in general, see [Chapter 11, “iTool User Interface Architecture”](#) in the *iTool Developer’s Guide* manual.

This class is written in the IDL language. Its source code can be found in the file `idlitui__define.pro` in the `lib/itools/ui_widgets` subdirectory of the IDL distribution.

Superclasses

[IDLitContainer](#)

Note

See the [“iTools Object Model Diagram”](#) in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See [“IDLitUI::Init”](#) on page 3416.

Properties

Objects of this class have the following properties. See [“IDLitUI Properties”](#) on page 3408 for details on individual properties.

- [GROUP_LEADER](#)

Methods

This class has the following methods:

- [IDLitUI::AddOnNotifyObserver](#)
- [IDLitUI::Cleanup](#)
- [IDLitUI::DoAction](#)
- [IDLitUI::GetProperty](#)
- [IDLitUI::GetTool](#)
- [IDLitUI::GetWidgetByName](#)
- [IDLitUI::Init](#)
- [IDLitUI::RegisterUIService](#)
- [IDLitUI::RegisterWidget](#)
- [IDLitUI::RemoveOnNotifyObserver](#)
- [IDLitUI::SetProperty](#)
- [IDLitUI::UnRegisterUIService](#)
- [IDLitUI::UnRegisterWidget](#)

In addition, this class inherits the methods of its superclasses (if any).

Examples

See [“Example: A Simple UI Panel”](#) in Chapter 14 of the *iTool Developer’s Guide* manual.

Version History

Introduced: 6.0

See Also

[Chapter 11, “iTool User Interface Architecture”](#) in the *iTool Developer’s Guide* manual

IDLitUI Properties

IDLitUI objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLitUI::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitUI::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitUI::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

GROUP_LEADER

A long integer containing the IDL Widget ID of the group leader widget for the user interface.

Property Type	Long Integer (Widget ID)		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLitUI::AddOnNotifyObserver

The IDLitUI::AddOnNotifyObserver procedure method is used to register a specified iTool component object as wishing to receive messages generated by the DoOnNotify method of another specified iTool component object.

Syntax

```
Obj->[IDLitUI:]AddOnNotifyObserver, IdObserver, IdSubject
```

Arguments

IdObserver

The object identifier of an iTool component object that is the *observer* expressing interest in the subject specified by *IdSubject*. Often, *IdObserver* is the object identifier of the object on which this method is being called.

The object specified by *IdObserver* must implement the OnNotify callback method, which is called when a notification message is dispatched to *IdObserver* by the DoOnNotify method of another iTool component object (usually the object specified by *IdSubject*). The OnNotify method has the following signature:

```
PRO ::OnNotify, idObject, idMessage, message
```

where

- *idObject* is the object identifier of the iTool component that is the source of the message.
- *idMessage* is a string that identifies the type of message being sent.
- *message* is the message data itself.

In general, the *idMessage* string is used by the OnNotify method to determine what type of action to take. See “[IDLitMessaging::DoOnNotify](#)” on page 3192 for additional details.

IdSubject

A string identifying the item that *IdObserver* is interested in. This is normally the object identifier of a particular iTool component object, but it can be any string value. When a message sent via [IDLitMessaging::DoOnNotify](#) specifies *IdSubject* as the originator, the *IdObserver* object’s OnNotify method is called.

Keywords

None

Version History

Introduced: 6.0

IDLitUI::Cleanup

The IDLitUI::Cleanup procedure method performs all cleanup on the object, and should be called by the Cleanup method of a subclass.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitUI::]Cleanup (only in subclass' *Cleanup* method)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitUI::DoAction

The IDLitUI::DoAction function method initiates an operation or action in the iTool object associated with this user interface. The identifier that is passed in as the argument to this method specifies the action that executes.

Syntax

Result = Obj->[IDLitUI::]doAction(Identifier)

Return Value

Returns 1 if the action succeeds, or 0 if the action fails.

Arguments

Identifier

This argument is set to a string identifier that specifies the object in the iTool that performs the operation or action that it implements.

Keywords

None.

Version History

Introduced: 6.0

IDLitUI::GetProperty

The IDLitUI::GetProperty procedure method retrieves the value of an IDLitUI property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclass.

Syntax

Obj->[IDLitUI:]GetProperty, [*PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitUI Properties](#)” on page 3408 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitUI::GetTool

The IDLitUI::GetTool function method returns an object reference to the iTool with which the user interface is associated.

Syntax

Result = *Obj*->[IDLitUI::]GetTool()

Return Value

Returns an object reference to the iTool associated with the current user interface, or a null object if there is no associated iTool.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitUI::GetWidgetByName

The IDLitUI::GetWidgetByName function method returns the IDL Widget ID of a widget that has been registered with the user interface object via a call to the [IDLitUI::RegisterWidget](#) method.

Syntax

Result = *Obj*->[IDLitUI::]GetWidgetByName(*Name*)

Return Value

Returns the widget identifier (a long integer) of the widget specified by the *Name* argument, or 0 if the widget was not registered with the iTool system.

Arguments

Name

Set this argument to the name that the desired widget was registered with.

Keywords

None

Example

This method can be used to determine whether a given widget already exists and act accordingly. For example, the following code checks to determine whether a widget registered as `myWidget` exists; if it does, that widget is mapped and un-iconified:

```
wID = oUI->GetWidgetByName(myWidget)
IF(wID NE 0) THEN BEGIN
    WIDGET_CONTROL, wID, /MAP, ICONIFY=0 ;; show the widget
    RETURN
ENDIF
```

Similar code is used in many of the widget interface elements used by the standard iTools included with IDL.

Version History

Introduced: 6.0

IDLitUI::Init

The IDLitUI::Init function method initializes the IDLitUI object, and should be called by the Init method of a subclass.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitUI', *oTool*)

or

Result = *Obj*->[IDLitUI::]Init(*oTool*) (*Only in subclass' Init method.*)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

oTool

An object reference to the IDLitTool object with which the user interface object is associated.

Keywords

None

Version History

Introduced: 6.0

IDLitUI::RegisterUIService

The IDLitUI::RegisterUIService function method registers a *user interface service* with the user interface. Once the user interface service has been registered, the iTool with which the user interface is associated can call the UI service to display dialogs or other UI elements.

Note

Every UI service must implement a callback function that displays the requested user interface. See [Chapter 13, “Creating a User Interface Service”](#) in the *iTool Developer’s Guide* manual for details.

Syntax

Result = Obj->[IDLitUI:]RegisterUIService(*Name*, *Callback*)

Return Value

Returns an iTool object identifier for the UI service specified by *Name*.

Arguments

Name

A string containing the human readable name of the user interface being registered.

Callback

A string containing the name of the callback function for this UI service. The callback function is called when execution of the UI service is requested. The callback should have the following calling signature:

```
FUNCTION NAME, oUI, oTarget
```

where:

- *NAME* is the name of the callback function,
- *oUI* is an object reference to the user interface object of which the UI service is a part
- *oTarget* is an object reference to the object that is the target of the operation or action with which the UI service is associated.

Keywords

None

Version History

Introduced: 6.0

IDLitUI::RegisterWidget

The IDLitUI::RegisterWidget function method registers an IDL widget hierarchy with the user interface object. Once a widget has been registered with the UI object, the UI can route messages to that widget and manage visibility if the widget is registered with the FLOATING property set.

Syntax

```
Result = Obj->[IDLitUI::]RegisterWidget(wID, Name, Callback[, /FLOATING])
```

Return Value

The iTool object identifier of the user interface adaptor that links the widget specified by *wID* with the iTool.

Arguments

wID

A long integer containing the widget ID of the widget being registered. This value is provided to the routine specified by the *Callback* argument when it is called.

Name

A string containing the human readable name for the widget being registered.

Callback

A string containing the name of a *callback procedure* for the widget. Any notification messages sent by the iTool to this widget are sent to this callback procedure. The callback procedure should have the following calling signature:

```
PRO NAME, wID, strID, message, messageData
```

where:

- *NAME* is the name of the callback procedure,
- *wID* is the widget ID provided to the IDLitUI::RegisterWidget method,
- *strID* is a string containing the object identifier of the iTool object that triggered the message,

- *message* is the message being sent to the callback procedure,
- *messageData* is any data associated with the message.

Note

If this argument is set to an empty string, no callbacks will be made to this widget.

Keywords

FLOATING

Set this keyword to treat the widget as a floating element of the specified iTool user interface, to be managed by the user interface object. If this keyword is set, the widget will only be visible if the associated iTool is “current,” and the widget will be destroyed when the iTool is destroyed.

Version History

Introduced: 6.0

IDLitUI::RemoveOnNotifyObserver

The IDLitUI::RemoveOnNotifyObserver procedure method is used to un-register a specified iTool component object as wishing to receive messages generated by the DoOnNotify method of another specified iTool component object. This method reverses the action of calling the [IDLitUI::AddOnNotifyObserver](#) method.

Syntax

Obj->[IDLitUI::]RemoveOnNotifyObserver, *IdObserver*, *IdSubject*

Arguments

IdObserver

The object identifier of an iTool component object that is currently registered as an observer of the component specified by *IdSubject*. Often, this is the object identifier of the object on which method is being called.

IdSubject

The object identifier of the iTool component object that *IdObserver* is currently registered as observing. This is normally the object identifier of a particular iTool component object, but it can also be a string that references a global system service.

Keywords

None

Version History

Introduced: 6.0

IDLitUI:: SetProperty

The IDLitUI:: SetProperty procedure method sets the value of an IDLitUI property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitUI::]SetProperty[, *PROPERTY=*value]

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitUI::UnRegisterUIService

The IDLitUI::UnRegisterUIService procedure method unregisters a user interface service with the user interface object.

Syntax

Obj->[IDLitUI::]UnRegisterUIService, *Name*

Arguments

Name

A string containing the name of the UI service to be unregistered.

Keywords

None

Version History

Introduced: 6.0

IDLitUI::UnRegisterWidget

The IDLitUI::UnRegisterWidget procedure method unregisters a widget with the user interface object.

Syntax

Obj->[IDLitUI::]UnRegisterWidget, *Name*

Arguments

Name

A string containing the name of the widget to be unregistered.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization

The IDLitVisualization class represents the base visualization component, from which all iTool visualization classes should subclass. IDLitVisualization acts as a container and provides methods for adding, removing, and grouping visualization components.

This class is written in the IDL language. Its source code can be found in the file `idlitvisualization__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLgrModel](#)

[IDLitMessaging](#)

[IDLitParameter](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitVisualization::Init](#)” on page 3453.

Properties

Objects of this class have the following properties. See “[IDLitVisualization Properties](#)” on page 3429 for details on individual properties.

- [CENTER_OF_ROTATION](#)
- [IMPACTS_RANGE](#)
- [ISOTROPIC](#)
- [MANIPULATOR_TARGET](#)
- [PROPERTY_INTERSECTION](#)
- [TYPE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has all the methods associated with the superclasses, plus the following methods:

- [IDLitVisualization::Add](#)
- [IDLitVisualization::Aggregate](#)
- [IDLitVisualization::BeginManipulation](#)
- [IDLitVisualization::Cleanup](#)
- [IDLitVisualization::EndManipulation](#)
- [IDLitVisualization::Get](#)
- [IDLitVisualization::GetCenterRotation](#)
- [IDLitVisualization::GetCurrentSelectionVisual](#)
- [IDLitVisualization::GetDataSpace](#)
- [IDLitVisualization::GetDataString](#)
- [IDLitVisualization::GetDefaultSelectionVisual](#)
- [IDLitVisualization::GetManipulatorTarget](#)
- [IDLitVisualization::GetProperty](#)
- [IDLitVisualization::GetRequestedAxesStyle](#)
- [IDLitVisualization::GetSelectionVisual](#)
- [IDLitVisualization::GetTypes](#)
- [IDLitVisualization::GetXYZRange](#)
- [IDLitVisualization::Init](#)
- [IDLitVisualization::Is3D](#)
- [IDLitVisualization::IsIsotropic](#)
- [IDLitVisualization::IsManipulatorTarget](#)
- [IDLitVisualization::IsSelected](#)
- [IDLitVisualization::Move](#)
- [IDLitVisualization::On2DRotate](#)
- [IDLitVisualization::OnAxesRequestChange](#)

- [IDLitVisualization::OnAxesStyleRequestChange](#)
- [IDLitVisualization::OnDataChange](#)
- [IDLitVisualization::OnDataComplete](#)
- [IDLitVisualization::OnDataRangeChange](#)
- [IDLitVisualization::OnDimensionChange](#)
- [IDLitVisualization::OnWorldDimensionChange](#)
- [IDLitVisualization::Remove](#)
- [IDLitVisualization::RequestsAxes](#)
- [IDLitVisualization::Restore](#)
- [IDLitVisualization::Rotate](#)
- [IDLitVisualization::Scale](#)
- [IDLitVisualization::Select](#)
- [IDLitVisualization::Set3D](#)
- [IDLitVisualization::SetAxesRequest](#)
- [IDLitVisualization::SetAxesStyleRequest](#)
- [IDLitVisualization::SetCurrentSelectionVisual](#)
- [IDLitVisualization::SetData](#)
- [IDLitVisualization::SetDefaultSelectionVisual](#)
- [IDLitVisualization::SetParameterSet](#)
- [IDLitVisualization::SetProperty](#)
- [IDLitVisualization::UpdateSelectionVisual](#)
- [IDLitVisualization::VisToWindow](#)
- [IDLitVisualization::WindowToVis](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [Chapter 6, “Creating a Visualization”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitVisualization Properties

IDLitVisualization objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via `IDLitVisualization::GetProperty`. Properties with the word “Yes” in the “Init” column of the property table can be initialized via `IDLitVisualization::Init`. Properties with the word “Yes” in the “Set” column of the property table can be set via `IDLitVisualization::SetProperty`.

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

CENTER_OF_ROTATION

A two- or three-element vector, $[x, y]$ or $[x, y, z]$ indicating the center (in data coordinates) of rotation for this visualization. If a two-element vector is supplied, the z value remains unchanged. When retrieving the value of this property, the center of rotation is always returned as a three-element vector.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IMPACTS_RANGE

A boolean that, when set to 0, indicates that the x , y , and z range of the contents of this visualization should not impact the x , y , and z range of any visualization that contains it. By default, IMPACTS_RANGE is 1.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ISOTROPIC

A boolean that, when set to a non-zero value, indicates that this object should have isotropic scaling applied to it. By default, ISOTROPIC is 0 (isotropic scaling is not enforced).

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MANIPULATOR_TARGET

A boolean that, when set to a non-zero value, indicates that this object should be treated as a target for manipulations. By default, MANIPULATOR_TARGET is 0 (this object is not a target for manipulations).

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PROPERTY_INTERSECTION

A boolean value that indicates whether the visualization should display the intersection or the union of the properties of any aggregated objects. Visualizations that display the union of their aggregated objects' properties appear in iTool browsers as a single visualization with one set of properties. Visualizations that display the intersection of their aggregated objects' properties appear in iTool browsers as a container for the aggregated objects, allowing access to the contained objects' properties as well as to those of the container. See [“Working with Aggregated Properties”](#) in Chapter 4 of the *iTool Developer's Guide* manual for additional discussion of aggregated properties.

Property Type	Boolean		
Name String	<i>no default value</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

TYPE

A string or vector of strings that identify the type(s) that this visualization represents. See “[Predefined iTool Visualization Classes](#)” in Chapter 6 of the *iTool Developer’s Guide* manual for a list of predefined types.

Property Type	String		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

IDLitVisualization::Add

The IDLitVisualization::Add procedure method adds objects to the visualization container.

This method overrides the superclass [IDLgrModel::Add](#) method. This specialized implementation performs the following actions:

- Stores visualizations in the correct order based on type so that they are correctly displayed
- Adds the objects to appropriate positions
- Handles property aggregation (if the AGGREGATE keyword is set)
- Updates this visualization's state flags (indicating whether this visualization is 3-D, isotropic, etc.) as appropriate
- Sends notification that the scene needs to be redrawn (unless the NO_UPDATE keyword is set)

Syntax

```
Obj->[IDLitVisualization::]Add, Objects [, /AGGREGATE] [, /NO_UPDATE]  
[, POSITION=index]
```

Arguments

Objects

An object reference (or array of object references), each referring to an atomic graphic object, an IDLgrModel, or another IDLitVisualization object to be added to the visualization container.

Keywords

AGGREGATE

Set this keyword to indicate that the object(s) being added should become part of this visualization's property aggregate. The properties of all aggregated objects are exposed as properties of this visualization (accessible via the GetProperty and SetProperty methods).

NO_UPDATE

Set this keyword to indicate that the overall scene should not be updated after the addition of the object(s). By default, the overall scene is updated.

POSITION

Set this keyword equal to a scalar or vector of zero-based index values. The number of elements specified must be equal to the number of object references specified by the *Objects* argument. Each index value specifies the position within this visualization container at which the corresponding object should be placed. The default is to add new objects at the end of the list of contained items.

Obsolete Keywords

The following keyword is obsolete:

- GROUP

For information on obsolete keywords, see [Appendix I, “Obsolete Features”](#).

Version History

Introduced: 6.0

Obsoleted GROUP keyword: 6.1

IDLitVisualization::Aggregate

The IDLitVisualization::Aggregate procedure method adds the given object(s) to this visualization's property aggregate. The properties of all aggregated objects are exposed as properties of this visualization (via ::GetProperty and ::SetProperty).

Syntax

Obj->[IDLitVisualization::]Aggregate, *Objects*

Arguments

Objects

An object reference (or array of object references), each of which refers to an atomic graphic object, an IDLgrModel, or another IDLitVisualization object to be aggregated within this visualization container.

Keywords

None.

Version History

Introduced: 6.0

IDLitVisualization::BeginManipulation

The IDLitVisualization::BeginManipulation procedure method handles notifications that an [IDLitManipulator](#) object is about to manipulate this visualization.

Note

Calling this method on this visualization object simply returns without action. This method is designed to be implemented by a sub-class (such as those listed in [“Predefined iTool Visualization Classes”](#) in Chapter 6 of the *iTool Developer’s Guide* manual).

Syntax

Obj->[IDLitVisualization::]BeginManipulation, *Manipulator*

Arguments

Manipulator

An object reference to the manipulator object sending notification.

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::Cleanup

The IDLitVisualization::Cleanup procedure method performs all cleanup on the object, and should be called by the subclass' Cleanup method.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitVisualization::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::EndManipulation

The IDLitVisualization::EndManipulation procedure method handles notifications that an [IDLitManipulator](#) object has finished manipulating this visualization.

Note

Calling this method on this visualization object simply returns without action. This method is designed to be implemented by a sub-class (such as those listed in [“Predefined iTool Visualization Classes”](#) in Chapter 6 of the *iTool Developer’s Guide* manual).

Syntax

Obj->[IDLitVisualization::]EndManipulation, *Manipulator*

Arguments

Manipulator

An object reference to the manipulator object sending notification.

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::Get

The IDLitVisualization::Get function method retrieves object(s) from the visualization.

Syntax

```
Result = Obj->[IDLitVisualization::]Get([, /ALL] [, COUNT=variable]  
    [, ISA=string or array of strings] [, POSITION=index or array of indices]  
    [, /SKIP_PRIVATE])
```

Return Value

Returns the reference of the object retrieved from the visualization. If the object is not contained by this visualization, a NULL object reference is returned.

Arguments

None

Keywords

ALL

Set this keyword to return an array of object references to all of the objects in the container.

COUNT

Set this keyword equal to a named variable that will contain the number of objects selected by the function.

ISA

Set this keyword equal to a class name or vector of class names. This keyword is used in conjunction with the ALL keyword. The ISA keyword filters the array returned by the ALL keyword, returning only the objects that inherit from the class or classes specified by the ISA keyword.

Note

This keyword is ignored if the ALL keyword is not provided.

POSITION

Set this keyword equal to a scalar or array containing the zero-based indices of the positions of the objects to return.

SKIP_PRIVATE

Set this keyword to indicate that this method should return only those objects that do not have the `PRIVATE` property set.

Version History

Introduced: 6.0

IDLitVisualization::GetCenterRotation

The IDLitVisualization::GetCenterRotation function method returns the center of rotation for this visualization.

Syntax

```
Result = Obj->[IDLitVisualization::]GetCenterRotation([, /NO_TRANSFORM]  
[, XRANGE=[xmin, xmax]] [, YRANGE=[ymin, ymax]] [, ZRANGE=[zmin,  
zmax]])
```

Return Value

Returns a two- or three-element vector, [x, y] or [x, y, z] indicating the center (in data coordinates) of the bounding box for this visualization if the center of rotation was not explicitly specified with the CENTER_OF_ROTATION property of this visualization. See the [CENTER_OF_ROTATION](#) property for more details.

Arguments

None

Keywords

NO_TRANSFORM

Set this keyword to indicate that this visualization's transform should not be applied when computing the x, y, and z ranges. By default, the transform is applied.

XRANGE

Set this keyword to a named variable that will contain a 2-element vector, [xmin, xmax], representing the x range of the bounding box for this visualization.

YRANGE

Set this keyword to a named variable that will contain a 2-element vector, [ymin, ymax], representing the y range of the bounding box for this visualization.

ZRANGE

Set this keyword to a named variable that will contain a 2-element vector, [zmin, zmax], representing the z range of the bounding box for this visualization.

Obsolete Keywords

The following keyword is obsolete:

- DATA

For information on obsolete keywords, see [Appendix I, “Obsolete Features”](#).

Version History

Introduced: 6.0

Obsoleted DATA keyword: 6.1

IDLitVisualization::GetCurrentSelectionVisual

The IDLitVisualization::GetCurrentSelectionVisual function method returns the currently active selection visual object for this visualization.

Syntax

Result = *Obj*->[IDLitVisualization::]GetCurrentSelectionVisual()

Return Value

Returns a reference to an IDLitManipulatorVisual object. If no selection visual is currently active, the NULL object reference is returned.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::GetDataSpace

The IDLitVisualization::GetDataSpace function method returns a reference to the parent data space object within the graphics hierarchy that contains the visualization.

Syntax

Result = Obj->[IDLitVisualization::]GetDataSpace([, /UNNORMALIZED])

Return Value

Returns a reference to the nearest data space object in the graphics hierarchy. If no data space objects are found by walking up the hierarchy, then a null object reference is returned.

Arguments

None

Keywords

UNNORMALIZED

Set this keyword to indicate that the returned data space should be the parent data space that subclasses from the IDLitVisDataSpace class (which is not normalized) rather than the IDLitVisNormDataSpace class (which is normalized).

Version History

Introduced: 6.0

IDLitVisualization::GetDataString

The IDLitVisualization::GetDataString function method retrieves a description of this visualization's data at the given x , y , and z location.

Note

This method is designed to be implemented as needed by subclasses. The IDLitVisualization::GetDataString method simply always returns an empty string

Syntax

Result = Obj->[IDLitVisualization::]GetDataString(XYZLocation)

Return Value

Returns a string describing the visualization's data at the given location.

Arguments

XYZLocation

A three-element vector representing the x , y , and z location of the data for which a string description is to be returned.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::GetDefaultSelectionVisual

The IDLitVisualization::GetDefaultSelectionVisual function method returns an object that serves as the default selection visual for this visualization.

Note

If the visualization on which this method is called is not a manipulator target (see “[MANIPULATOR_TARGET](#)” on page 3430), the default selection visual returned by this method will be used as the current selection visual whenever the visualization is selected.

Syntax

Result = *Obj*->[IDLitVisualization:]GetDefaultSelectionVisual()

Return Value

Returns a reference to an IDLitManipulatorVisual object.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::GetManipulatorTarget

The IDLitVisualization::GetManipulatorTarget function method retrieves the manipulator target associated with this visualization. The manipulator target may be this visualization itself.

Syntax

Result = *Obj*->[IDLitVisualization::]GetManipulatorTarget()

Return Value

Returns an object reference to the manipulator target, or a NULL object reference if none is found.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::GetProperty

The IDLitVisualization::GetProperty procedure method retrieves the value of a property or group of properties for the object.

Syntax

Obj->[IDLitVisualization::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitVisualization Properties](#)” on page 3429 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Obsolete Keywords

The following keyword is obsolete:

- GROUP_PARENT

For information on obsolete keywords, see [Appendix I, “Obsolete Features”](#).

Version History

Introduced: 6.0

Obsoleted GROUP_PARENT keyword: 6.1

IDLitVisualization::GetRequestedAxesStyle

The IDLitVisualization::GetRequestedAxesStyle function method returns a scalar value indicating the axes style requested by this visualization. This method returns the axes style request made by the first visualization within the hierarchy (beginning with the root visualization) to make a request. If none of the visualizations in the hierarchy requests an axes style then -1 is returned.

Syntax

Result = *Obj*->[IDLitVisualization::]GetRequestedAxesStyle()

Return Value

Returns a scalar value indicating the requested axes style. Possible return values are:

Value	Requested Axes Style
0	None.
1	At Dataspace Minimum.
2	Box Axes.
3	Crosshairs.
-1	No visualization in the hierarchy requested axes.

Table 12: Axes Style Values

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::GetSelectionVisual

The IDLitVisualization::GetSelectionVisual function method retrieves the selection visual for this visualization that corresponds to the given manipulator.

Syntax

Result = *Obj*->[IDLitVisualization:]GetSelectionVisual(*Manipulator*)

Return Value

Returns a reference to an IDLitManipulatorVisual object that corresponds to the given manipulator. If this visualization already has a selection visual for the given manipulator, a reference to that selection visual is returned. Otherwise, a default selection visual will be requested from the manipulator, added to the visualization, and returned.

Arguments

Manipulator

Set this argument to a reference to an IDLitManipulator object that identifies the manipulator for which a selection visual is to be retrieved.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::GetTypes

The IDLitVisualization::GetTypes function method identifies the types that this visualization represents, including base types and any specializations.

Note

This method is designed to be implemented as needed by subclasses of the IDLitVisualization class. Calling this method on the IDLitVisualization class itself returns the following string array: ['VISUALIZATION', '_VISUALIZATION', *AnySpecifiedTypes*] where *AnySpecifiedTypes* are provided by any value set for the TYPE property of the IDLitVisualization class.

Syntax

Result = *Obj*->[IDLitVisualization::]GetTypes()

Return Value

Returns a vector of strings identifying the types that this visualization represents. See [“Predefined iTool Visualization Classes”](#) in Chapter 6 of the *iTool Developer’s Guide* manual for a list of predefined types

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::GetXYZRange

The IDLitVisualization::GetXYZRange function method computes the x , y , and z ranges of the overall contents of the visualization, taking into account the IMPACTS_RANGE property setting for itself and its contents.

Syntax

```
Result = Obj->[IDLitVisualization::]GetXYZRange(XRange,YRange, ZRange  
[, /DATA] [, /NO_TRANSFORM])
```

Return Value

Returns a 1 if the x , y , and z ranges were successfully computed, or 0 otherwise.

Arguments

XRange

A named variable that, upon return, will contain a two-element vector, [$xmin$, $xmax$], representing the x -range of the contents of this visualization that impact ranges.

YRange

A named variable that, upon return, will contain a two-element vector, [$ymin$, $ymax$], representing the y -range of the contents of this visualization that impact ranges.

ZRange

A named variable that, upon return, will contain a two-element vector, [$zmin$, $zmax$], representing the z -range of the contents of this visualization that impact ranges.

Keywords

DATA

Set this keyword to indicate that the ranges should be computed for the full data sets of the contents of this visualization. By default, if the keyword is not set, the ranges are computed for the displayed portions of the data sets.

NO_TRANSFORM

Set this keyword to indicate that this visualization's transform matrix should not be applied when computing the x , y , and z ranges. By default, the transform matrix is applied.

Version History

Introduced: 6.0

IDLitVisualization::Init

The IDLitVisualization::Init function method initializes the visualization object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitVisualization' [, PROPERTY=value] )
```

or

```
Result = Obj->[IDLitVisualization::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLitVisualization Properties](#)” on page 3429 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitVisualization::Is3D

The IDLitVisualization::Is3D function method determines whether or not this visualization (or any of its contents) is marked as being three-dimensional.

Syntax

Result = *Obj*->[IDLitVisualization::]Is3D()

Return Value

Returns 1 if this visualization is marked as being three-dimensional, or 0 if it is not three-dimensional.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::IsIsotropic

The IDLitVisualization::IsIsotropic function method indicates whether or not this visualization (or any of its contents) is marked as being isotropic. See the [ISOTROPIC](#) property for more information.

Syntax

Result = *Obj*->[IDLitVisualization::]IsIsotropic()

Return Value

Returns 1 if the visualization or any of the items it contains has the ISOTROPIC property set, or 0 otherwise.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::IsManipulatorTarget

The IDLitVisualization::IsManipulatorTarget function method determines whether or not this visualization is a manipulator target.

Syntax

Result = *Obj*->[IDLitVisualization::]IsManipulatorTarget()

Return Value

Returns 1 if this visualization is a manipulator target, or 0 if it is not.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::IsSelected

The IDLitVisualization::IsSelected function method determines if this visualization is currently selected or not.

Syntax

Result = *Obj*->[IDLitVisualization::]IsSelected()

Return Value

Returns 1 if this visualization is currently selected, or 0 if it is not selected.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::Move

The IDLitVisualization::Move procedure method moves an object from one position in the visualization container to a new position. Use the IDLitVisualization::Get method to return the objects within the visualization.

The positions of the objects within a container controls how they are rendered. The object whose location has the lowest index value is rendered first. If several objects are located at the same point in three-dimensional space, the object rendered first will occlude objects rendered later.

Note

This method overrides the IDL_Container::Move method, which IDLitVisualization inherits since IDL_Container is a superclass of IDLgrModel. This specialized implementation of the Move method ensures visualizations are stored in the correct order based on type when they are moved. Those items that must appear on top of other objects are stored as the last items in the list of objects contained by this visualization.

Syntax

Obj->[IDLitVisualization::]Move, *Source*, *Destination*

Arguments

Source

The zero-based index of the current location of the object to be moved.

Destination

The zero-based index of the new location where the object will reside after being moved.

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::On2DRotate

The IDLitVisualization::On2DRotate procedure method notifies objects that use the visualization's data when the rotation of the parent dataspace changes.

Syntax

Obj->[IDLitVisualization::]On2DRotate, *Notifier*, *IsRotated*

Arguments

Notifier

An object reference to the parent object that is sending notification when the dataspace rotation changes.

IsRotated

A scalar value that is non-zero if the dataspace is now rotated, or zero otherwise.

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::OnAxesRequestChange

The IDLitVisualization::OnAxesRequestChange procedure method notifies objects that use the visualization's data when the axes of a contained object change.

Syntax

Obj->[IDLitVisualization::]OnAxesRequestChange, *Notifier*, *AxesRequest*

Arguments

Notifier

An object reference to the object that is sending notification when axes change.

AxesRequest

A scalar value representing the new axes request. Allowable values are:

Value	Requested Axes
0	An x-axis request.
1	A y-axis request.
2	A z-axis request.

Table 13: Allowable AxesRequest Values

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::OnAxesStyleRequestChange

The IDLitVisualization::OnAxesStyleRequestChange procedure method notifies objects that use the visualization's data when the axes style of a contained object changes.

Syntax

Obj->[IDLitVisualization::]OnAxesStyleRequestChange, *Notifier*, *StyleRequest*

Arguments

Notifier

An object reference to the object that is sending notification when the axes style changes.

StyleRequest

A scalar value representing the new axes style request. Allowable values are:

Value	Requested Axes Style
0	None.
1	At Dataspace Minimum.
2	Box Axes.
3	Crosshairs.
-1	No visualization in the hierarchy requested axes.

Table 14: Axes Style Values

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::OnDataChange

The IDLitVisualization::OnDataChange procedure method notifies objects that use the visualization's data when the visualization's data changes.

Note

This method is generally not called explicitly.

Syntax

Obj->[IDLitVisualization::]OnDataChange, *Notifier*

Arguments

Notifier

A reference to the object that is sending notification when the data changes.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::OnDataComplete

The IDLitVisualization::OnDataComplete procedure method notifies objects that use the visualization's data when changes to the visualization's data are complete.

Note

This method is generally not called explicitly.

Syntax

Obj->[IDLitVisualization::]OnDataComplete, *Notifier*

Arguments

Notifier

A reference to the object that is sending notification when the data changes are complete.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::OnDataRangeChange

The IDLitVisualization::OnDataRangeChange procedure method notifies objects that use the visualization's data when the range of the visualization's data changes.

Note

This method is generally not called explicitly.

Syntax

Obj->[IDLitVisualization::]OnDataRangeChange, *Notifier*, *XRange*, *YRange*, *Zrange*

Arguments

Notifier

A reference to the object that is sending notification when the data range changes.

XRange

A two-element vector, [*xmin*, *xmax*], representing the new *x*-range.

YRange

A two-element vector, [*ymin*, *ymax*], representing the new *y*-range.

ZRange

A two-element vector, [*zmin*, *zmax*], representing the new *z*-range.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::OnDimensionChange

The IDLitVisualization::OnDimensionChange procedure method notifies objects that use the visualization's data when the dimensionality of the visualization's data changes.

Syntax

Obj->[IDLitVisualization::]OnDimensionChange, Notifier, Is3D

Arguments

Notifier

An object reference to the object that is sending notification when the dimensionality changes.

Is3D

A scalar value that is non-zero if the object is now three-dimensional, or zero otherwise.

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::OnWorldDimensionChange

The IDLitVisualization::OnWorldDimensionChange procedure method notifies objects that use the visualization's data when the dimensionality of the visualization's parent dataspace changes.

Syntax

Obj->[IDLitVisualization::]OnWorldDimensionChange, *Notifier*, *Is3D*

Arguments

Notifier

An object reference to the parent object that is sending notification when the dimensionality changes.

Is3D

A scalar value that is non-zero if the object is now three-dimensional, or zero otherwise.

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::Remove

The IDLitVisualization::Remove procedure method removes the given object(s) from the visualization.

This method overrides the [IDL_Container::Remove](#) method, which IDLitVisualization inherits since IDL_Container is a superclass of IDLgrModel. This specialized implementation of the Remove method performs the following actions:

- Handles property aggregation
- Updates this visualization's state flags (indicating whether this visualization is 3-D, isotropic, etc.) as appropriate
- Sends notification that the scene needs to be redrawn (unless the NO_UPDATE keyword is set)

Syntax

```
Obj->[IDLitVisualization::]Remove, Object [, /NO_UPDATE]
```

Arguments

Object

A reference (or vector of references) to the object(s) to be removed from this visualization.

Keywords

NO_UPDATE

Set this keyword to indicate the scene should not be updated after objects are removed. By default, the overall scene is updated.

Version History

Introduced: 6.0

Added NO_UPDATE keyword:6.1

IDLitVisualization::RequestsAxes

The IDLitVisualization::RequestsAxes function method indicates whether or not the visualization requests axes. Most visualization classes do request axes. However, axes, images, ROIs, map grids, profile lines, and backgrounds do not request axes.

Syntax

Result = *Obj*->[IDLitVisualization::]RequestsAxes()

Return Value

Returns a 1 if the visualization does request axes, or 0 if does not request axes.

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLitVisualization::Restore

The IDLitVisualization::Restore procedure method performs any transitional work required after an object of this class has been restored from a SAVE file, to ensure that its state is appropriate for the current version.

Note

This method overrides the implementation in the IDLitComponent class. This specialized implementation ensures that all axes requests and axis style requests are set properly.

Syntax

Obj->[IDLitVisualization::]Restore

Arguments

None

Keywords

None

Version History

Introduced: 6.1

See Also

[IDL_Savefile](#), [SAVE](#), [RESTORE](#)

IDLitVisualization::Rotate

The IDLitVisualization::Rotate procedure method rotates this visualization about the given axis by the given angle.

This method overrides the superclass's [IDLgrModel::Rotate](#) method. This specialized implementation rotates about a center of rotation rather than about the origin, [0,0,0].

Syntax

```
Obj->[IDLitVisualization::]Rotate, Axis, Angle [, CENTER_OF_ROTATION=[x, y, z]] [, /PREMULTIPLY]
```

Arguments

Axis

A three-element vector of the form [x, y, z] describing the axis about which the visualization is to be rotated.

Angle

A scalar value indicating the angle of rotation (measured in degrees).

Keywords

CENTER_OF_ROTATION

Set this keyword to a three-element vector, [x, y, z], representing the center of the rotation. By default, the visualization's own center of rotation (accessible through [“IDLitVisualization::GetCenterRotation”](#) on page 3440) will be used.

PREMULTIPLY

Set this keyword to cause the rotation matrix specified by *Axis* and *Angle* to be pre-multiplied to this visualization's current transformation matrix. By default, the rotation matrix is post-multiplied.

Note

In pre-multiplication, the current visualization's transform and scaling are applied before the specified rotation. In post-multiplication, the specified rotation is applied before the visualization's transformation and scaling.

Version History

Introduced: 6.1

IDLitVisualization::Scale

The IDLitVisualization::Scale procedure method scales the visualization by the given scale factors.

Syntax

```
Obj->[IDLitVisualization::]Scale, SX, SY, SZ [, CENTER_OF_ROTATION=[x, y] |  
[x, y, z]] [, /PREMULTIPLY]
```

Arguments

SX

The factor in the x -dimension by which the visualization is to be scaled.

SY

The factor in the y -dimension by which the visualization is to be scaled.

SZ

The factor in the z -dimension by which the visualization is to be scaled.

Keywords

CENTER_OF_ROTATION

Set this keyword to a 2- or 3-element vector, $[x, y]$ or $[x,y,z]$, representing the center of rotation (in data coordinates) to be used as the center for scaling. By default, if `PREMULTIPLY` is set, then this visualization's own center of rotation will be used as the center for scaling; if `PREMULTIPLY` is not set, the result of transforming this visualization's center of rotation by its own transformation matrix will be used as the center for scaling.

PREMULTIPLY

Set this keyword to cause the scaling matrix specified by S_x , S_y , and S_z to be pre-multiplied to this visualization's transformation matrix. By default, the scaling matrix is post-multiplied.

Version History

Introduced: 6.0

IDLitVisualization::Select

The IDLitVisualization::Select procedure method handles notification of mechanisms that key off the current selection (such as the visualization browser) when this visualization has been selected.

Syntax

```
Obj->[IDLitVisualization::]Select[, Mode]  
    [, /ADDITIVE | /SELECT | /TOGGLE | /UNSELECT] [, /NO_NOTIFY]
```

Arguments

Mode

A scalar that indicates the selection mode. Valid values include:

- 0 - Unselect (deselect this visualization).
- 1 - Select (set this visualization as the current selection). This mode is the default.
- 2 - Toggle (toggle the selection of this visualization). Use the **CTRL** key to toggle.
- 3 - Additive (add this visualization to the current selection list). Use the **SHIFT** key to add.

Keywords

ADDITIVE

Set this keyword to indicate that the selection mode is additive. This is equivalent to setting *Mode* to 3.

NO_NOTIFY

Set this keyword to indicate that this visualization's parent should not be notified of this selection. By default, the parent is notified.

SELECT

Set this keyword to indicate that the selection mode is select. This is equivalent to setting *Mode* to 1.

TOGGLE

Set this keyword to indicate that the selection mode is toggle. This is equivalent to setting *Mode* to 2.

UNSELECT

Set this keyword to indicate that the selection mode is unselect. This is equivalent to setting *Mode* to 0.

Version History

Introduced: 6.0

IDLitVisualization::Set3D

The IDLitVisualization::Set3D procedure method sets a flag indicating this visualization is three-dimensional.

Syntax

```
Obj->[IDLitVisualization::]Set3D, Is3D [, /ALWAYS] [, /AUTO_COMPUTE]
```

Arguments

Is3D

A non-zero value indicates that the visualization is three-dimensional; 0 indicates that it is two-dimensional.

Keywords

ALWAYS

Set this keyword to indicate that the given 3-D setting always applies.

AUTO_COMPUTE

Set this keyword to indicate that the 3-D setting for this visualization should be auto-computed based upon the dimensionality of its contents. This keyword is mutually exclusive of the ALWAYS keyword, and the *Is3D* argument is ignored if AUTO_COMPUTE is set.

Version History

Introduced: 6.0

IDLitVisualization::SetAxesRequest

The IDLitVisualization::SetAxesRequest procedure method sets the current axes request for this visualization.

Syntax

```
Obj->[IDLitVisualization::]SetAxesRequest [, AxesRequest]  
[, /ALWAYS | /AUTO_COMPUTE] [, /NO_NOTIFY]
```

Arguments

AxesRequest

A scalar value indicating whether this visualization requests axes:

- 0 – Axes are not requested by the visualization
- 1 – Axes are requested by the visualization.

Note

If this argument is not present, the visualization is marked as requesting axes.

Keywords

ALWAYS

Set this keyword to always request or not request axes as specified by the *AxesRequest* argument. This setting will override the requests made by children of the visualization.

AUTO_COMPUTE

Set this keyword to compute the axes for this visualization based on the axes requested by objects contained in the visualization.

Note

This keyword is mutually exclusive of the ALWAYS keyword. If this keyword is set, the *AxesRequest* argument is ignored.

NO_NOTIFY

Set this keyword to indicate that this visualization's parent should not be notified of a change in axes request. By default, the parent is notified.

Version History

Introduced: 6.1

IDLitVisualization::SetAxesStyleRequest

The IDLitVisualization::SetAxesStyleRequest procedure method sets the current axes style request for this visualization. The axes container is automatically created whenever a data space is instantiated, but the default axes style property is 0 (None) until visualizations are added.

Whenever a visualization is added to the visualization container, it is queried for axes requirements. Most visualization classes do request axes (with the exception of axes, images, ROIs, map grids, profile lines, and backgrounds). When visualizations are added, and the current axes style value is 0 (None), then the axes style will be changed as follows:

- The axes style is set according to the request of the first visualization in the hierarchy to request a *particular* axes style. By default, most visualizations do not request a particular style. The exceptions are plots, which request style 2 (Box Axes) if non-polar, or style 3 (Crosshairs) if polar.
- The default style 1 (At Dataspace Minimum) is used if none of the visualizations in the hierarchy request a particular axes style.

Syntax

Obj->[IDLitVisualization::]SetAxesStyleRequest, *StyleRequest* [, /NO_NOTIFY]

Arguments

StyleRequest

A scalar value representing the requested axes style. The value should correspond to one of the values listed in the following table:

Value	Requested Axes Style
0	None.
1	At Dataspace Minimum.
2	Box Axes.
3	Crosshairs.
-1	No visualization in the hierarchy requested a particular axes style.

Table 15: Axes Style Values

Keywords

NO_NOTIFY

Set this keyword to indicate that this visualization's parent should not be notified of a change in the axes style request. By default, the parent is notified.

Version History

Introduced: 6.1

IDLitVisualization::SetCurrentSelectionVisual

The IDLitVisualization::SetCurrentSelectionVisual procedure method sets the current selection visual for the given manipulator.

Syntax

Obj->[IDLitVisualization::]SetCurrentSelectionVisual, *Manipulator*

Arguments

Manipulator

An object reference (that subclasses from IDLitManipulator) identifying the manipulator that corresponds to the selection visual to be set as current.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::SetData

The IDLitVisualization::SetData function method sets the data parameter of the visualization.

Note

If the SetData method returns successfully, notification of a data change is issued to all observers of the data object.

Syntax

Result = Obj->[IDLitVisualization::]SetData(Data)

Return Value

Returns a 1 if successful, or 0 if it fails.

Arguments

Data

A reference to the data object to be associated with the visualization parameter.

Keywords

This method accepts all keywords accepted by the IDLitParameter::SetData method.

Version History

Introduced: 6.0

IDLitVisualization::SetDefaultSelectionVisual

The IDLitVisualization::SetDefaultSelectionVisual procedure method sets the default selection visual to be associated with this visualization.

Note

If the visualization on which this method is called is not a manipulator target (see “[MANIPULATOR_TARGET](#)” on page 3430) the default selection visual set by this method will be used as the current selection visual whenever the visualization is selected.

Syntax

```
Obj->[IDLitVisualization::]SetDefaultSelectionVisual, SelectionVisual  
[, POSITION=value]
```

Arguments

SelectionVisual

A reference to the IDLitManipulatorVisual object that is added to the visualization object and serves as the default selection visual for this visualization.

Keywords

POSITION

Set this keyword equal to the zero-based position at which the selection visual should be inserted within the container. The default is to add the object to the end of the container (the visualization object).

Version History

Introduced: 6.0

IDLitVisualization::SetParameterSet

The IDLitVisualization::SetParameterSet function method associates a parameter set with this visualization.

Note

If the SetParameterSet method returns successfully, notification of a data change is issued to all observers of the visualization's data.

Syntax

Result = Obj->[IDLitVisualization::]SetParameterSet(ParameterSet)

Return Value

Returns a 1 if successful, or a 0 otherwise.

Arguments

ParameterSet

A reference to the IDLitParameterSet object to be associated with the visualization.

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization:: SetProperty

The IDLitVisualization::SetProperty procedure method sets the value of a property or group of properties for the object.

Syntax

Obj->[IDLitVisualization::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLitVisualization Properties](#)” on page 3429 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitVisualization::UpdateSelectionVisual

The IDLitVisualization::UpdateSelectionVisual procedure method transforms this visualization's selection visual to match the visualization's geometry.

Syntax

Obj->[IDLitVisualization::]UpdateSelectionVisual

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitVisualization::VisToWindow

The IDLitVisualization::VisToWindow procedure method transforms given points from visualization data space to window device coordinates.

Syntax

```
Obj->[IDLitVisualization::]VisToWindow, InX, InY, InZ, OutX, OutY, OutZ  
    [, /NO_TRANSFORM]
```

or

```
Obj->[IDLitVisualization::]VisToWindow, InX, InY, OutX, OutY  
    [, /NO_TRANSFORM]
```

or

```
Obj->[IDLitVisualization::]VisToWindow, InVerts, OutVerts [, /NO_TRANSFORM]
```

Arguments

InX

A vector of input X values (in visualization data space).

InY

A vector of input Y values (in visualization data space). The number of elements of this vector must match the number of elements in *InX*.

InZ

A vector of input Z values (in visualization data space). The number of elements of this vector must match the number of elements in *InX*.

InVerts

A [2, *n*] or [3, *n*] array of input vertices (in visualization data space).

OutX

A named variable that will contain a vector of output X values (in window device coordinates).

OutY

A named variable that will contain a vector of output Y values (in window device coordinates).

OutZ

A named variable that will contain a vector of output Z values (in window device coordinates).

OutVerts

A named variable that will contain a vector of output vertices (in window device coordinates).

Keywords

NO_TRANSFORM

If this keyword is set, the current transformation matrix is not included in the computation. Setting this keyword is useful when computing selection visual scaling.

Version History

Introduced: 6.0

IDLitVisualization::WindowToVis

The IDLitVisualization::WindowToVis procedure method transforms given points from window device coordinates to visualization data space.

Syntax

Obj->[IDLitVisualization::]WindowToVis, *InX*, *InY*, *InZ*, *OutX*, *OutY*, *OutZ*

or

Obj->[IDLitVisualization::]WindowToVis, *InX*, *InY*, *OutX*, *OutY*

or

Obj->[IDLitVisualization::]WindowToVis, *InVerts*, *OutVerts*

Arguments

InX

A vector representing the input x values (in window device coordinates).

InY

A vector representing the input y values (in window device coordinates). The number of elements of this vector must match the number of elements in *InX*.

InZ

A vector representing the input z values (in window device coordinates). The number of elements of this vector must match the number of elements in *InX*.

InVerts

A $[2, n]$ or $[3, n]$ array representing the input vertices (in window device coordinates).

OutX

A named variable that upon return will contain a vector representing the output x values (in visualization data space).

OutY

A named variable that upon return will contain a vector representing the output y values (in visualization data space).

OutZ

A named variable that upon return will contain a vector representing the output z values (in visualization data space).

OutVerts

A named variable that upon return will contain a vector representing the output vertices (in visualization data space).

Keywords

None

Version History

Introduced: 6.0

IDLitWindow

The IDLitWindow class serves as the base class for all windows within the iTools framework.

Superclasses

[IDLgrWindow](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitWindow::Init](#)” on page 3504.

Properties

Objects of this class inherit properties from the IDLgrWindow superclass. See “[IDLgrWindow Properties](#)” on page 4138 for descriptions of these properties.

Methods

This class has the following methods:

- [IDLitWindow::Add](#)
- [IDLitWindow::AddWindowEventObserver](#)
- [IDLitWindow::Cleanup](#)
- [IDLitWindow::ClearSelections](#)
- [IDLitWindow::DoHitTest](#)
- [IDLitWindow::GetEventMask](#)
- [IDLitWindow::GetProperty](#)
- [IDLitWindow::GetSelectedItems](#)
- [IDLitWindow::Init](#)
- [IDLitWindow::OnKeyboard](#)

- [IDLitWindow::OnMouseDown](#)
- [IDLitWindow::OnMouseMove](#)
- [IDLitWindow::OnMouseUp](#)
- [IDLitWindow::OnScroll](#)
- [IDLitWindow::Remove](#)
- [IDLitWindow::RemoveWindowEventObserver](#)
- [IDLitWindow::SetCurrentZoom](#)
- [IDLitWindow::SetEventMask](#)
- [IDLitWindow::SetManipulatorManager](#)
- [IDLitWindow::SetProperty](#)
- [IDLitWindow::ZoomIn](#)
- [IDLitWindow::ZoomOut](#)

In addition, this class inherits the methods of its superclass.

Version History

Introduced: 6.0

IDLitWindow Properties

IDLitWindow objects inherit properties from the IDLgrWindow superclass. See [“IDLgrWindow Properties”](#) on page 4138 for descriptions of these properties.

Properties with the word “Yes” in the “Get” column of the table can be retrieved via [IDLitWindow::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLitWindow::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLitWindow::SetProperty](#).

IDLitWindow::Add

The IDLitWindow::Add procedure method adds the given object(s) to the window.

Syntax

Obj->[IDLitWindow::]Add, *Objects* [, POSITION=*value*]

Arguments

Objects

A reference (or array of references) to the object(s) to be added to the window. Any of the objects that inherit from IDLgrViewGroup are added directly to the window's scene, which represents a container for all of the views (and their corresponding visualization hierarchies) that appear within a window. The remaining objects are added to the current view within the window's scene.

Keywords

POSITION

Set this keyword equal to a scalar or array containing the zero-based indices of the position within the container at which the new object should be placed.

Version History

Introduced: 6.0

IDLitWindow::AddWindowEventObserver

The IDLitWindow::AddWindowEventObserver procedure method adds the given object(s) to the list of observers that are to be notified of events that occur within this window. Each observer must support the following methods:

- OnKeyboard
- OnMouseDown
- OnMouseMove
- OnMouseUp

When an event occurs within this window, the corresponding method (from the list above) will be called for each observer in the window's list.

Syntax

Obj->[IDLitWindow::]AddWindowEventObserver, *Objects*

Arguments

Objects

A reference (or vector of references) to the object(s) to be added as window event observers.

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::Cleanup

The IDLitWindow::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitWindow::]Cleanup (only in subclass' *Cleanup* method)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::ClearSelections

The IDLitWindow::ClearSelections procedure method clears the window's list of currently selected items (within its current view). The items within the current view that had been selected are deselected.

Syntax

Obj->[IDLitWindow:]ClearSelections

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::DoHitTest

The IDLitWindow::DoHitTest function method performs a hit test to determine which visualizations within the destination are displayed at a given pixel location.

Note

The objects returned in *Result* are the “top-level” objects (usually IDLgrModel or IDLitVisualization objects). Objects *contained* by the returned objects can be returned using the SUB_HIT keyword.

Syntax

```
Result = Obj->DoHitTest(X, Y [, DIMENSIONS=[width, height]][, /ORDER]
  [, SUB_HIT=variable] [, UNITS={0 | 1 | 2 | 3}])
```

Return Value

Returns a vector of references to the objects (usually IDLgrModel or IDLitVisualization objects) that appear at the given location. If no visualizations are displayed at that location, a null reference is returned.

Arguments

X

A floating-point value representing the *x*-location at which the hit test is to be performed

Y

A floating-point value representing the *y*-location at which the hit test is to be performed.

Keywords

DIMENSIONS

Set this keyword to a two-element integer vector, [*w*, *h*], to specify the dimensions (width and height) of the hit test box. The hit test box is centered at the location specified by the *X* and *Y* arguments. Any object that falls within this hit test box will be included in the return vector. By default, the hit test box is 3 pixels by 3 pixels.

ORDER

Set this keyword to control the ordering of objects in the returned hit test list when the objects are at the same distance from the viewer. Set ORDER=0 (the default) to return the objects in the order in which they were drawn. Set ORDER=1 to return objects in the reversed order from which they were drawn.

Note

If the visualization objects use DEPTH_TEST_FUNCTION=4 (less than or equal), then it is recommended that the ORDER keyword be set to 1 to ensure that objects at the same depth are returned in the same order in which they appear visually.

SUB_HIT

Set this keyword to a named variable that will contain references to all *contained* visualization objects that satisfy all of the following conditions:

- The objects are contained by the nearest hit visualization (the first object in the return value)
- The objects are SELECT_TARGETs (see “IDLgrModel” on page 3756)
- The objects are actively displayed at the given X and Y coordinates

The results of SUB_HIT are returned in the order in which the sub-hit visualizations appear in the graphics hierarchy. Sub-hits are only returned for the first visualization.

UNITS

Set this keyword to a scalar value to indicate the units of measure for the X, Y arguments and the DIMENSIONS keyword. Valid values include:

- 0 = Device (default)
- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to the dimensions of this window

Version History

Introduced: 6.0

Added ORDER keyword: 6.1

IDLitWindow::GetEventMask

The IDLitWindow::GetEventMask function method returns a bitwise mask representing the events that are enabled for this window.

Syntax

```
Result = Obj->[IDLitWindow::]GetEventMask([, BUTTON_EVENTS=variable]
[, KEYBOARD_EVENTS=variable] [, MOTION_EVENTS=variable]
[, TRACKING_EVENTS=variable])
```

Return Value

Returns the bitwise mask as an unsigned long integer. The bits in the mask are as follows:

Bit	Value	Event
0	1	Button Events
1	2	Motion Events
2	4	Keyboard Events
3	8	Tracking Events

Table 7-1: Bits of the Event Mask

Arguments

None

Keywords

BUTTON_EVENTS

Set this keyword to a named variable that upon return will contain a 1 if mouse button events are currently enabled for this window, or a 0 otherwise.

KEYBOARD_EVENTS

Set this keyword to a named variable that upon return will contain a 1 if keyboard events are currently enabled for this window, or a 0 otherwise.

MOTION_EVENTS

Set this keyword to a named variable that upon return will contain a 1 if mouse motion events are currently enabled for this window, or a 0 otherwise.

TRACKING_EVENTS

Set this keyword to a named variable that upon return will contain a 1 if tracking events are currently enabled for this window, or a 0 otherwise.

Version History

Introduced: 6.0

IDLitWindow::GetProperty

The IDLitWindow::GetProperty procedure method retrieves the value of an IDLitWindow property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclass.

Syntax

Obj->[IDLitData::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitWindow Properties](#)” on page 3493 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitWindow::GetSelectedItems

The IDLitWindow::GetSelectedItems function method returns the currently selected objects within this window's scene.

Syntax

```
Result = Obj->[IDLitWindow::]GetSelectedItems([, /ALL] [, COUNT=named variable])
```

Return Value

Returns a vector of references to the objects currently selected within this window's scene. If no objects are currently selected, this function returns a -1.

Arguments

None

Keywords

ALL

Set this keyword to return all classes of selected items. By default, only visualizations are returned.

COUNT

Set this keyword to a named variable that will contain the number of items returned by this method.

Version History

Introduced: 6.0

Added ALL keyword: 6.1

IDLitWindow::Init

The IDLitWindow::Init function method initializes the window object.

This function method initializes the IDLitWindow object, and should be called by the subclass' Init method. This method also calls the superclass' Init method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLitWindow'[, PROPERTY=value])
```

or

```
Result = Obj->[IDLitWindow::]Init([, PROPERTY=value])  
(Only in subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or 0 otherwise.

Arguments

None

Keywords

Any property listed under [“IDLitWindow Properties”](#) on page 3493 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitWindow::OnKeyboard

The `IDLitWindow::OnKeyboard` procedure method handles notification (from the native window device) that a keyboard event has occurred, and passes along that notification to all observers in the list of window event observers. With proper event handling, these values are passed into the procedure arguments. Alternatively, keyboard events can be simulated by calling this method directly.

Syntax

Obj->[IDLitWindow::]OnKeyboard, *IsASCII*, *Character*, *KeySymbol*, *X*, *Y*, *Press*,
Release, *Modifiers*

Arguments

IsASCII

A scalar value that indicates whether the keyboard event corresponds to a character in the ASCII character set. If *IsASCII* is non-zero, the *Character* argument will be set to that ASCII character. If *IsASCII* is zero, the *KeySymbol* argument will be set to a numeric value indicating the key that was pressed.

Character

If *IsASCII* is non-zero, this argument is set to the ASCII character that corresponds to the key that was pressed. Otherwise, this argument is set to zero.

KeySymbol

If *IsASCII* is zero, this argument is set to an unsigned long integer that indicates the key that was pressed. Otherwise, this argument is set to zero. Valid values for key symbols include:

- 1 = Shift
- 2 = Control
- 3 = Caps Lock
- 4 = Alt
- 5 = Left
- 6 = Right
- 7 = Up

- 8 = Down
- 9 = Page Up
- 10 = Page Down
- 11 = Home
- 12 = End

X,Y

The *x*-coordinate and *y*-coordinate of the mouse cursor (in device coordinates) at the time of the event.

Press

A scalar value indicating whether this event represents a key press. This argument is non-zero if the event is the result of pressing the key.

Release

A scalar value indicating whether this event represents a key release. This argument is non-zero if the event is the result of releasing the key.

Modifiers

A bitwise mask indicating which modifier keys are active at the time of the keyboard event. If a bit is zero, the key is up; if the bit is set, the key is pressed. The following table describes the bits in this bit mask:

Bit	Value	Modifier Key
0	1	Shift
1	2	Control
2	4	Caps Lock
3	8	Alt

Table 16: Modifier Values

Keywords

None

Version History

Introduced: 6.0

Corrected syntax to read *IsASCII* instead of *IsAlphaNumeric*, corrected *KeySymbol* values, added *X*, *Y*, *Press*, *Release*, and *Modifiers* arguments: 6.1

IDLitWindow::OnMouseDown

The `IDLitWindow::OnMouseDown` procedure method handles notification (from the native window device) that a mouse down event has occurred, and passes along that notification to all observers in the list of window event observers.

Syntax

Obj->[IDLitWindow::]OnMouseDown, X, Y, ButtonMask, Modifiers, NumClicks

Arguments

X

A floating-point value representing the *x*-location (in device coordinates) of the mouse event.

Y

A floating-point value representing the *y*-location (in device coordinates) of the mouse event.

ButtonMask

An integer value of a bitwise mask indicating which of the left, center, or right mouse button was pressed:

Bitmask	Mouse Button
1	Left
2	Middle
4	Right

Table 7-2: Bitmask for Button Events

Modifiers

An integer value of a bitwise mask indicating which modifier keys are active at the time the mouse button is pressed. If a bit is zero, the key is up; if the bit is set, the key is pressed. The following table describes the bits in this bitmask:

Bit	Value	Modifier Key
0	1	Shift
1	2	Control
2	4	Caps Lock
3	8	Alt

Table 7-3: Bits of the Modifier Key Mask

NumClicks

An integer value indicating the number of times the mouse button was clicked.

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::OnMouseMotion

The IDLitWindow::OnMouseMotion procedure method handles notification (from the native window device) that a mouse motion event has occurred, and passes along that notification to all observers in the list of window event observers.

Syntax

Obj->[IDLitWindow::]OnMouseMotion, *X*, *Y*, *Modifiers*

Arguments

X

A floating-point value representing the *x*-location (in device coordinates) of the mouse event.

Y

A floating-point value representing the *y*-location (in device coordinates) of the mouse event.

Modifiers

An integer value of a bitwise mask indicating which modifier keys are active at the time the mouse button is pressed. If a bit is zero, the key is up; if the bit is set, the key is pressed. The following table describes the bits in this bitmask:

Bit	Value	Modifier Key
0	1	Shift
1	2	Control
2	4	Caps Lock
3	8	Alt

Table 7-4: Bits of the Modifier Key Mask

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::OnMouseUp

The IDLitWindow::OnMouseUp procedure method handles notification (from the native window device) that a mouse up event has occurred, and passes along that notification to all observers in the list of window event observers.

Syntax

Obj->[IDLitWindow::]OnMouseUp, *X*, *Y*, *ButtonMask*

Arguments

X

A floating-point value representing the *x*-location (in device coordinates) of the mouse event.

Y

A floating-point value representing the *y*-location (in device coordinates) of the mouse event.

ButtonMask

An integer value of a bitwise mask indicating which of the left, center, or right mouse button was pressed:

Bitmask	Mouse Button
1	Left
2	Middle
4	Right

Table 7-5: Bitmask for Button Events

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::OnScroll

The IDLitWindow::OnScroll procedure method handles notification (from the native window device) that a scrolling event has occurred. This notification is passed on to the window's scene, which handles the cropping of each of its contained views. A scene represents a container for all of the views (and their corresponding visualization hierarchies) that appear within a window.

Syntax

Obj->[IDLitWindow::]OnScroll, *X*, *Y*

Arguments

X

A floating-point value representing the *x*-coordinate of the lower left corner of the visible portion of the canvas (in device coordinates) after the scroll.

Y

A floating-point value representing the *y*-coordinate of the lower left corner of the visible portion of the canvas (in device coordinates) after the scroll.

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::Remove

The IDLitWindow::Remove procedure method removes the given object(s) from the window.

Syntax

Obj->[IDLitWindow::]Remove, *Object* [, /ALL] [, POSITION=index]

Arguments

Object

A reference (or vector of references) to the object(s) to be removed from this window.

Keywords

ALL

Set this keyword to remove all objects from the container. If this keyword is set, the *Object* argument is not required.

POSITION

Set this keyword equal to the zero-based index of the object to be removed. If the *Object* argument is supplied, this keyword is ignored.

Version History

Introduced: 6.0

IDLitWindow::RemoveWindowEventObserver

The IDLitWindow::RemoveWindowEventObserver procedure method removes the given object(s) from the list of observers that are notified of events that occur within this window.

Syntax

Obj->[IDLitWindow::]RemoveWindowEventObserver, *Objects*

Arguments

Objects

A reference (or vector of references) to the object(s) to be removed from the list of window event observers.

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::SetCurrentZoom

The `IDLitWindow::SetCurrentZoom` procedure method sets the current zoom factor for this window by changing its virtual dimensions. Setting the zoom factor on this window overrides any zoom factor inherited from the `IDLgrWindow`. The contents of the window are updated to reflect the new zoom factor.

Syntax

Obj->[IDLitWindow::]SetCurrentZoom, *ZoomFactor* [, /RESET]

Arguments

ZoomFactor

A positive floating point value indicating the zoom factor to be applied to the window. Values less than 1.0 result in a zooming out; values greater than 1.0 result in a zooming in on the contents of the window.

Keywords

RESET

Set this keyword to indicate that the zoom factor should be reset to 1.0. If this keyword is present, the *ZoomFactor* argument is ignored.

Version History

Introduced: 6.0

IDLitWindow::SetEventMask

The IDLitWindow::SetEventMask procedure method enables the given events within this window.

When an event occurs within this window, if the corresponding event type is enabled, then the list of its window event observers will be notified of the event. See “IDLitWindow::AddWindowEventObserver” on page 3495 for more details.

Syntax

```
Obj->[IDLitWindow::]SetEventMask([EventMask] [, /BUTTON_EVENTS]
    [, /KEYBOARD_EVENTS] [, /MOTION_EVENTS] [, /TRACKING_EVENTS])
```

Arguments

EventMask

An unsigned long integer representing the bitwise mask for the events that are to be enabled for this window. The bits in the mask are as follows:

Bit	Value	Event
0	1	Button Events
1	2	Motion Events
2	4	Keyboard Events
3	8	Tracking Events

Table 7-6: Bits of the Event Mask

This argument is optional; the keywords described below may be used instead.

Keywords

BUTTON_EVENTS

Set this keyword to indicate that mouse button events are to be enabled for this window.

KEYBOARD_EVENTS

Set this keyword to indicate the keyboard events are to be enabled for this window.

MOTION_EVENTS

Set this keyword to indicate that mouse motion events are to be enabled for this window.

TRACKING_EVENTS

Set this keyword to indicate that tracking events are to be enabled for this window.

Version History

Introduced: 6.0

IDLitWindow::SetManipulatorManager

The IDLitWindow::SetManipulatorManager procedure method sets the given IDLitManipulatorManager object as the current manager of this window's manipulators.

Syntax

Obj->[IDLitWindow::]SetManipulatorManager, *Manager*

Arguments

Manager

A reference to the IDLitManipulatorManager object that is to serve as the manager for this window's manipulators.

Keywords

None

Version History

Introduced: 6.0

IDLitWindow:: SetProperty

The IDLitWindow::SetProperty procedure method sets the value of an IDLitWindow property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitWindow::]SetProperty[, *PROPERTY=**value*]

Arguments

None

Keywords

Any property listed under [“IDLitWindow Properties”](#) on page 3493 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitWindow::ZoomIn

The IDLitWindow::ZoomIn procedure method causes the current zoom factor for this window to be increased (that is, multiplied by the factor given by the inherited IDLgrWindow `ZOOM_BASE` property). The contents of the window are updated to reflect the new zoom factor. See “IDLitWindow::SetCurrentZoom” on page 3518 for additional details.

Syntax

Obj->[IDLitWindow::]ZoomIn

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitWindow::ZoomOut

The IDLitWindow::ZoomOut procedure method causes the current zoom factor for this window to be decreased (that is, divided by the factor given by the inherited IDLgrWindow `ZOOM_BASE` property). The current contents of the window are updated to reflect the new zoom factor. See “IDLitWindow::SetCurrentZoom” on page 3518 for additional details.

Syntax

Obj->[IDLitWindow::]ZoomOut

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitWriter

The IDLitWriter class defines the interface used to construct file writers for the iTools framework. Objects of this class are not intended to be created as standalone entities; rather, this class should be included as the superclass of an iTool file writer class.

This class is written in the IDL language. Its source code can be found in the file `idlitwriter__define.pro` in the `lib/itools/framework` subdirectory of the IDL distribution.

Superclasses

[IDLitComponent](#)

[IDLitMessaging](#)

Note

See the “[iTools Object Model Diagram](#)” in Chapter 2 of the *iTool Developer’s Guide* manual for a visual representation of the iTools object model hierarchy.

Creation

See “[IDLitWriter::Init](#)” on page 3532.

Properties

Objects of this class do not have any properties of their own, but do have properties inherited from any superclasses.

Objects of this class have the following properties. See “[IDLitWriter Properties](#)” on page 3527 for details on individual properties.

- [TYPES](#)

Methods

This class has the following methods:

- [IDLitWriter::Cleanup](#)
- [IDLitWriter::GetFileExtensions](#)
- [IDLitWriter::GetFilename](#)

- [IDLitWriter::GetProperty](#)
- [IDLitWriter::Init](#)
- [IDLitWriter::IsA](#)
- [IDLitWriter::SetData](#)
- [IDLitWriter::SetFilename](#)
- [IDLitWriter::SetProperty](#)

In addition, this class inherits the methods of its superclasses.

Examples

See [Chapter 10, “Creating a File Writer”](#) in the *iTool Developer’s Guide* manual for examples using this class and its methods.

Version History

Introduced: 6.0

IDLitWriter Properties

IDLitWriter objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the properties table can be retrieved via [IDLitWriter::GetProperty](#). Properties with the word “Yes” in the “Init” column of the properties table can be initialized via [IDLitWriter::Init](#). Properties with the word “Yes” in the “Set” column of the properties table can be set via [IDLitWriter::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

TYPES

Set this property to a string or an array of strings indicating the types of data that this file writer accepts. iTools data types are described in [Chapter 3, “Data Management”](#) in the *iTool Developer’s Guide* manual. Set this property to a null string (' ') to specify that the file reader can accept all types of data.

Property Type	User Defined		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: Yes	Registered: No

IDLitWriter::Cleanup

The IDLitWriter::Cleanup procedure method performs all cleanup on the object, and should be called by the subclass' Cleanup method.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLitWriter::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitWriter::GetFileExtensions

The IDLitWriter::GetFileExtensions method is called by the system to retrieve the file extensions supported by this particular writer.

Syntax

Result = *Obj*->[IDLitWriter:]GetFileExtensions([**COUNT**=*count*])

Return Value

Returns a scalar or string array that contains the file extensions associated with this writer.

Arguments

None

Keywords

COUNT

Set this keyword to a named variable that will contain the number of file extensions returned by this method.

Version History

Introduced: 6.0

IDLitWriter::GetFilename

The IDLitWriter::GetFilename function method is called by the system to retrieve the current filename associated with this writer. Due to the automated nature of the file writer system, filenames can be associated with a file writer and then the file can be written at a later time. This method allows direct access to the file currently associated with the writer.

In addition, this methodology is helpful when multiple writes are performed from a given file.

Syntax

Result = *Obj*->[IDLitWriter:]GetFilename()

Return Value

Returns a string containing the current file name associated with this writer, or an empty string if no filename has been associated.

Arguments

None

Keywords

None

Version History

Introduced: 6.0

IDLitWriter::GetProperty

The IDLitWriter::GetProperty procedure method retrieves the value of an IDLitWriter property, and should be called by the subclass' GetProperty method. This method also retrieves properties defined in the superclasses.

Syntax

Obj->[IDLitWriter::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLitWriter Properties](#)” on page 3527 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 6.0

IDLitWriter::Init

The IDLitWriter::Init function method initializes the IDLitWriter object, and should be called by the subclass' Init method. This method also calls the superclass' Init method.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: if you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLitWriter', *Extensions* [, *PROPERTY=value*])

or

Result = *Obj*->[IDLitWriter::]Init(*Extensions* [, *PROPERTY=value*])
(Only in subclass' Init method.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or 0 otherwise.

Arguments

Extensions

A scalar or string array that contains the file extensions that are common for this file type. These values should not include the period that is often associated with file extensions (a correct value is “jpeg” not “.jpeg”).

Keywords

Any property listed under [“IDLitWriter Properties”](#) on page 3527 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0

IDLitWriter::IsA

The IDLitWriter::IsA function method is called by the system to determine if the given file is of the type supported by this file writer. Often this is used to determine what file writer to use when opening a new file.

The default behavior provided by this method is to check the file extension on the provided file name with the extensions provided to this object during initialization.

When a new writer is implemented, the default behavior can be used, or the developer can provide further logic to determine if the provided file is the correct type.

Syntax

Result = *Obj*->[IDLitWriter::]IsA(*Filename*)

Return Value

Returns a 1 if the writer supports this type of file, or a 0 if the writer does not support this type of file.

Arguments

Filename

A string scalar representing of the filename, which is used to check and determine if the writer supports its format.

Keywords

None

Version History

Introduced: 6.0

IDLitWriter::SetData

The IDLitWriter::SetData function method is called by the system to set the data for the current file. When called, the writer should access the current filename and write the desired data.

This method will contain a majority of the implementation for a new writer class.

Syntax

Result = *Obj*->[IDLitWriter::]SetData(*Data*)

Return Value

Returns a 1 if successful, or a 0 if the initialization failed.

Arguments

Data

A data object that contains the information to be written.

Keywords

None

Version History

Introduced: 6.0

IDLitWriter::SetFilename

The IDLitWriter::SetFilename procedure method is called by the system to set the current filename associated with this writer. Due to the automated nature of the file writer system, filenames can be associated with a file writer and then the file can be written at a later time.

Syntax

Obj->[IDLitWriter::]SetFilename, *Filename*

Arguments

Filename

A string that contains the filename associated with this writer.

Keywords

None

Version History

Introduced: 6.0

IDLitWriter:: SetProperty

The IDLitWriter::SetProperty procedure method sets the value of an IDLitWriter property, and should be called by the subclass' SetProperty method. This method also calls the superclass' SetProperty method.

Syntax

Obj->[IDLitWriter::]SetProperty[, *PROPERTY*=*value*]

Arguments

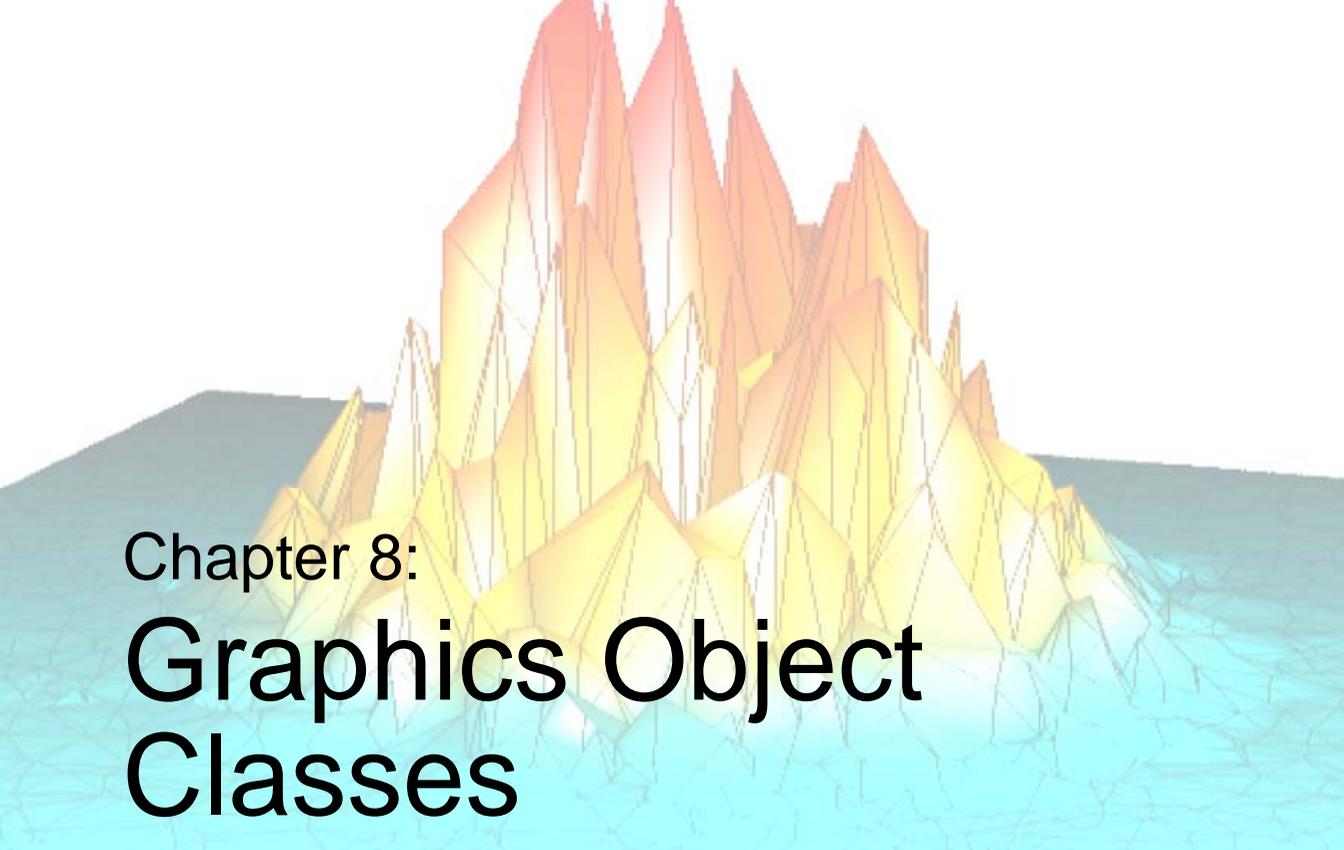
None

Keywords

Any property listed under “[IDLitWriter Properties](#)” on page 3527 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 6.0



Chapter 8: Graphics Object Classes

This chapter describes IDL's built-in graphics class library.

IDLgrAxis	3540	IDLgrPolyline	3875
IDLgrBuffer	3572	IDLgrPrinter	3902
IDLgrClipboard	3600	IDLgrROI	3927
IDLgrColorbar	3627	IDLgrROIGroup	3946
IDLgrContour	3647	IDLgrScene	3963
IDLgrFont	3687	IDLgrSurface	3976
IDLgrImage	3695	IDLgrSymbol	4010
IDLgrLegend	3719	IDLgrTessellator	4021
IDLgrLight	3740	IDLgrText	4032
IDLgrModel	3756	IDLgrView	4056
IDLgrMPEG	3779	IDLgrViewgroup	4073
IDLgrPalette	3795	IDLgrVolume	4085
IDLgrPattern	3809	IDLgrVRML	4114
IDLgrPlot	3819	IDLgrWindow	4135
IDLgrPolygon	3844		

IDLgrAxis

An axis object represents a single vector that may include a set of tick marks, tick labels, and a title.

An IDLgrAxis object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrAxis::Init](#)” on page 3569.

Properties

Objects of this class have the following properties. See “[IDLgrAxis Properties](#)” on page 3542 for details on individual properties.

- [ALL](#)
- [AM_PM](#)
- [COLOR](#)
- [DAYS_OF_WEEK](#)
- [DEPTH_TEST_FUNCTION](#)
- [DIRECTION](#)
- [EXTEND](#)
- [HIDE](#)
- [LOG](#)
- [MINOR](#)
- [NOTEXT](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [ALPHA_CHANNEL](#)
- [CLIP_PLANES](#)
- [CRANGE](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_WRITE_DISABLE](#)
- [EXACT](#)
- [GRIDSTYLE](#)
- [LOCATION](#)
- [MAJOR](#)
- [MONTHS](#)
- [PALETTE](#)
- [RANGE](#)
- [SUBTICKLEN](#)

- [TEXTALIGNMENTS](#)
- [TEXTPOS](#)
- [THICK](#)
- [TICKFORMAT](#)
- [TICKINTERVAL](#)
- [TICKLEN](#)
- [TICKUNITS](#)
- [TITLE](#)
- [XCOORD_CONV](#)
- [YCOORD_CONV](#)
- [ZCOORD_CONV](#)
- [TEXTBASELINE](#)
- [TEXTUPDIR](#)
- [TICKDIR](#)
- [TICKFRMTDATA](#)
- [TICKLAYOUT](#)
- [TICKTEXT](#)
- [TICKVALUES](#)
- [USE_TEXT_COLOR](#)
- [XRANGE](#)
- [YRANGE](#)
- [ZRANGE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrAxis::Cleanup](#)
- [IDLgrAxis::GetCTM](#)
- [IDLgrAxis::GetProperty](#)
- [IDLgrAxis::Init](#)
- [IDLgrAxis::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrAxis Properties

IDLgrAxis objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via `IDLgrAxis::GetProperty`. Properties with the word “Yes” in the “Init” column of the property table can be set via `IDLgrAxis::Init`. Properties with the word “Yes” in the “Set” column in the property table can be set via `IDLgrAxis::SetProperty`.

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

A floating-point value in the range [0.0, 1.0] to specify the opacity of the entire axis, including tick text and title text. The default value of 1.0 causes IDL to draw the axis completely opaque. If the value of this property is less than 1.0, then the pixels of the axis are blended with the pixels already on the screen, where the color of the axis is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha. Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results. This property has no effect on devices using indexed color mode. When the `USE_TEXT_COLOR` property is set to a non-zero value, the tick and title text is drawn using the value of the `ALPHA_CHANNEL` property from the tick and/or title text objects. In other words, `USE_TEXT_COLOR` also implies using the text objects' alpha as well as its color.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see “[Alpha Channel and Objects](#)” in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

AM_PM

A string array of 2 names to be used for the names of the AM and PM string when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the TICKFORMAT property.

Property Type	String array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

CLIP_PLANES

A 4 by n floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any x , y , or z coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the `MAX_NUM_CLIP_PLANES` property of the `GetDeviceInfo` method for the `IDLgrBuffer`, `IDLgrClipboard`, `IDLgrWindow`, and `IDLgrVRML` objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

The color to be used as the foreground color for this axis. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default is [0, 0, 0].

Property Type	Color		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CRANGE

A double-precision floating-point vector of the form [*minval, maxval*] that will contain the actual full range of the axis. This computed range may not exactly match the requested range provided via the `RANGE` property due to tick mark rounding and effects of the `EXTEND` property.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DAYS_OF_WEEK

A string array of 7 names to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the TICKFORMAT property.

Property Type	String array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DIRECTION

An integer value that specifies which axis is being created. Specify 0 (zero) to create an X axis, 1 (one) to create a Y axis, or 2 to create a Z axis. Specifying this property is the same as specifying the optional *Direction* argument.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Direction		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

EXACT

A Boolean value that determines whether to force the axis range to be exactly as specified. If this property is not set, the range may be lengthened or shortened slightly to allow for evenly spaced tick marks.

Property Type	Boolean		
Name String	Use exact axis range		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

EXTEND

A Boolean value that determines whether to extend the axis slightly beyond the range of data to be plotted, which is a value specified by the RANGE property. This can be useful when you specify the axis range based on the minimum and maximum data values, but do not want the graphic to extend all the way to the end of the axis. The

amount by which the axis is extended is determined by the system using a number of factors, including the MAJOR property of the axis.

Property Type	Boolean		
Name String	Extend axis		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

GRIDSTYLE

An integer or two-element vector that indicates the line style that should be used to draw the axis' tick marks. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the GRIDSTYLE property equal to one of the following integer values:

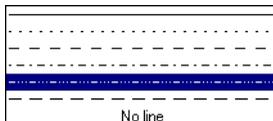
- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq repeat \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `GRIDSTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

In a property sheet, this property appears as a Linestyle drop-down list, with the following options:



Property Type	Linestyle		
Name String	Line style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this object should be drawn. Options:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In property sheets, this property appears as an enumerated list, with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LOCATION

A two- or three-element floating-point vector of the form $[x, y]$ or $[x, y, z]$ to specify the coordinate through which the axis should pass. The default is $[0, 0, 0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LOG

A Boolean value that indicates whether the axis is logarithmic.

Property Type	Boolean		
Name String	Use logarithmic axis		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MAJOR

An integer representing the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

Property Type	Integer		
Name String	Number of major ticks		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MINOR

An integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

Property Type	Integer		
Name String	Number of minor ticks		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MONTHS

A string array of 12 names to be used for the names of the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the TICKFORMAT property.

Property Type	String array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

NOTEXT

A Boolean value indicating whether the tick labels and the axis title should be drawn.

- 0 = Draw tick labels and axis title (the default).
- 1 = Do not draw tick labels and axis title.

In a property sheet, this property appears as an enumerated list. Options:

- True = Draw tick labels and axis title (the default).
- False = Do not draw the tick labels and axis title.

Property Type	ENUMLIST		
Name String	Text show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

The object reference of a palette object (an instance of the IDLgrPalette object class) that defines the color palette of this object. This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the PALETTE property on this object is not set, the destination object PALETTE property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: No	Init: No	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

RANGE

A two-element floating-point vector containing the minimum and maximum data values covered by the axis. The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

SUBTICKLEN

A floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

Property Type	FLOAT		
Name String	Minor tick length		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTALIGNMENTS

A two-element floating-point vector, [*horizontal*, *vertical*], specifying the horizontal and vertical alignments for the tick text. Each alignment value should be a value between 0.0 and 1.0. For horizontal alignment, 0.0 left-justifies the text; 1.0 right-justifies the text.

For vertical alignment, 0.0 bottom-justifies the text, 1.0 top-justifies the text. The defaults are as follows:

- X-Axis: [0.5, 1.0] (centered horizontally, top-justified vertically)
- Y-Axis: [1.0, 0.5] (right-justified horizontally, centered vertically)
- Z-Axis: [1.0, 0.5] (right-justified horizontally, centered vertically)

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TEXTBASELINE

A two- or three-element floating-point vector describing the direction in which the baseline of the tick text is to be oriented. Use this property in conjunction with the

TEXTUPDIR property to specify the plane on which the tick text lies. The default is [1,0,0].

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TEXTPOS

An integer that indicates on which side of the axis to draw the tick text labels. The table below describes the placement of the tick text with each setting.

Axis	TEXTPOS = 0	TEXTPOS = 1
X	Tick text will be drawn <i>below</i> the X axis, where <i>below</i> is defined as being toward the direction of the negative Y axis (this is the default).	Tick text will be drawn <i>above</i> the X axis, where <i>above</i> is described as being toward the direction of the positive Y axis.
Y	Tick text will be drawn to the <i>left</i> of the Y Axis, where <i>left</i> is defined as being toward the direction of the negative X axis (this is the default).	Tick text will be drawn to the <i>right</i> of the Y axis, where <i>right</i> is defined as being toward the direction of the positive X axis.
Z	Tick text will be drawn to the <i>left</i> of the Z axis, where <i>left</i> is defined as being toward the direction of the negative X axis (this is the default).	Tick text will be drawn to the <i>right</i> of the Z axis, where <i>right</i> is defined as being toward the direction of the positive X axis.

Table 8-1: Values for the TEXTPOS property

In a property sheet, this property appears as an enumerated list. Options:

- Below/left: Text will be drawn below the axis and to the left.
- Above/right: Text will be drawn above the axis and to the right.

Property Type	ENUMLIST		
Name String	Text position		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTUPDIR

A two- or three-element floating-point vector describing the direction in which the up-vector of the tick text is to be oriented. Use this property in conjunction with the TEXTBASELINE property to specify the plane on which the tick text lies.

TEXTUPDIR should be orthogonal to TEXTBASELINE. The default is as follows:

- X-Axis: [0, 1, 0]
- Y-Axis: [0, 1, 0]
- Z-Axis: [0, 0, 1]

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value in points between 1.0 and 10.0 that specifies the line thickness used to draw the axis. The default is 1.0 points.

In a property sheet, this property appears as a Thickness drop-down list with the following options:



Property Type	Thickness		
Name String	Line thickness		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKDIR

An integer value of zero or one to indicate the tick mark direction.

Axis	TICKDIR = 0	TICKDIR = 1
X	Tick marks will be drawn above the X axis, in the direction of the positive Y axis (this is the default).	Tick marks will be drawn below the X axis.
Y	Tick marks will be drawn to the right of the Y axis, in the direction of the positive X axis (this is the default).	Tick marks will be drawn to the left of the Y axis.
Z	Tick marks will be drawn to the right the Z axis, in the direction of the positive X axis (this is the default).	Tick marks will be drawn to the left of the Z axis.

Table 8-2: Tick Mark Direction Values

In a property sheet, this property appears as an enumerated list, with the following options:

- Right/above: Tick marks will be drawn above the axis and to the right.
- Left/below: Tick marks will be drawn below the axis and to the left.

Property Type	ENUMLIST		
Name String	Tick direction		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKFORMAT

A string, or an array of strings, in which each string represents a format string or the name of a function to be used to format the tick mark labels. If an array is provided, each string corresponds to a level of the axis. (The [TICKUNITS](#) property determines the number of levels for an axis.) If the number of strings stored in this property is one, `GetProperty` returns a scalar string, otherwise `GetProperty` returns an array of strings.

If the string begins with an open parenthesis, it is treated as a standard format string. (See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual.)

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether [TICKUNITS](#) is specified.

If [TICKUNITS](#) are not specified:

- The callback function is called with three parameters: *Axis*, *Index*, and *Value*, where:
- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis
- *Index* is the tick mark index (indices start at 0)
- *Value* is the data value at the tick mark (a double-precision floating point value)

If [TICKUNITS](#) are specified:

The callback function is called with four parameters: *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled. (Level indices start at 0.)

Whether or not [TICKUNITS](#) is specified, the callback function can also define a `DATA` keyword. If the callback function accepts the `DATA` keyword, it will be set equal to the value (if any) of the [TICKFRMTDATA](#) property.

Note

For more information, see “[XYZ]TICKFORMAT” on page 4345.

Used with the LABEL_DATE function, this property can easily create axes with date/time labels.

Property Type	String or string array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKFRMTDATA

A value of any type that will be passed to the user-supplied formatting callback function (specified via the [TICKFORMAT](#) property) as the value of the DATA keyword. By default, this value is undefined, indicating that the value of the DATA keyword will be undefined.

Note

TICKFRMTDATA will not be included in the structure returned via the ALL property to the IDLgrAxis::GetProperty method.

Property Type	Scalar of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKINTERVAL

A floating-point scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis RANGE and the number of major tick marks (MAJOR). This property takes precedence over MAJOR.

For example, if TICKUNITS=['S','H','D'], and TICKINTERVAL=30, then the interval between major ticks for the first axis level will be 30 seconds.

Property Type	FLOAT		
Name String	Tick interval		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKLAYOUT

An integer scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

In a property sheet, this property appears as an enumerated list with the following options:

- Axis plus labels: Same as the 0 value above.
- Labels only: Same as the 1 value above.
- Box style: Same as the 2 value above.

Property Type	ENUMLIST		
Name String	Tick layout		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKLEN

A floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Major tick length		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKTEXT

An object reference to a single instance of the [IDLgrText](#) object class (with multiple strings) or a vector of instances of the [IDLgrText](#) object class (one per major tick) to specify the annotations to be assigned to the tickmarks. By default, with **TICKTEXT** set equal to a null object, IDL computes the tick labels based on major tick values. The positions of the provided text objects may be overwritten; position is determined according to tick mark location. The tickmark text will have the same color as the [IDLgrAxis](#) object, regardless of the color specified by the **COLOR** property of the [IDLgrText](#) object or objects, unless the **USE_TEXT_COLOR** property is specified.

Note

If IDL computes the tick labels, the text object it creates will be destroyed automatically when the axis object is destroyed, even if you have altered the properties of the text object. If you create your own text object containing tickmark text, however, it will *not* be destroyed automatically.

Property Type	Object reference or string vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKUNITS

A string (or a vector of strings) indicating the units to be used for axis tick labeling. If the number of levels in the axis is one, **GetProperty** returns a scalar string, otherwise **GetProperty** returns an array of strings.

If more than one unit is provided, the axis will be drawn in multiple levels, one level per unit.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- "" - Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the "Numeric" unit. This is the default setting.

If any of the time units are utilized, then the tick values are interpreted as Julian date/time values. Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS='Day' is equivalent to TICKUNITS='Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000 A.D., respectively.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	STRING		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKVALUES

A floating-point vector of data values representing the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis

range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TITLE

An object reference to an instance of the [IDLgrText](#) object class that specifies the title for the axis. The default is the null object, specifying that no title is drawn. The title will be centered along the axis, even if the text object itself has an associated location. The title will have the same color as the IDLgrAxis object, regardless of the color specified by the COLOR property of the IDLgrText object, unless the USE_TEXT_COLOR property is specified.

The IDLgrText object becomes a child of the IDLgrAxis object; therefore, to use the same IDLgrText object for multiple axes, you must use multiple IDLgrAxis objects.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

USE_TEXT_COLOR

A Boolean value that indicates whether, for the tick text and/or title of the axis, the color property values set for the given [IDLgrText](#) objects are to be used to draw those text items. By default, this value is zero, indicating that the color properties of the IDLgrText objects will be ignored, and that the COLOR property for the axis object will be used for these text items instead.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors that convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element double-precision floating-point vector of the form [*xmin*, *xmax*] that specifies the range of *x* data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors that convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element double-precision floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is $[0.0, 1.0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

ALPHA_CHANNEL added: 6.1

IDLgrAxis::Cleanup

The IDLgrAxis::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrAxis::]Cleanup(*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrAxis::GetCTM

The IDLgrAxis::GetCTM function method returns the graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrAxis::]GetCTM([, DESTINATION=objref] [, PATH=objref(s)]  
[, TOP=objref])
```

Return Value

Returns the 4 x 4 double-precision floating-point graphics transform matrix.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the axis object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrAxis::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrAxis::GetProperty

The IDLgrAxis::GetProperty procedure method retrieves the value of a property or group of properties for the axis.

Syntax

Obj->[IDLgrAxis::]GetProperty [, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrAxis Properties](#)” on page 3542 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 5.0

IDLgrAxis::Init

The IDLgrAxis::Init function method initializes an axis object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrAxis' [, Direction] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrAxis::]Init([Direction] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Direction

An integer value specifying which axis is being created. Specify 0 (zero) to create an X axis, 1 (one) to create a Y axis, or 2 to create a Z axis.

Keywords

Any property listed under “[IDLgrAxis Properties](#)” on page 3542 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

CLIP_PLANES keyword added: 5.6

IDLgrAxis:: SetProperty

The IDLgrAxis::SetProperty procedure method sets the value of a property or group of properties for the axis.

Syntax

Obj->[IDLgrAxis::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrAxis Properties](#)” on page 3542 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrBuffer

An IDLgrBuffer object is an in-memory, off-screen destination object. Object trees can be drawn to instances of the IDLgrBuffer object and the resulting image can be retrieved from the buffer using the Read() method. The off-screen representation avoids dithering artifacts by providing a full-resolution buffer for objects using either the RGB or Color Index color models.

Note

Objects or subclasses of this type can not be saved or restored.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrBuffer::Init](#)” on page 3590.

Properties

Objects of this class have the following properties. See “[IDLgrBuffer Properties](#)” on page 3574 for details on individual properties.

- [ALL](#)
- [COLOR_MODEL](#)
- [DIMENSIONS](#)
- [GRAPHICS_TREE](#)
- [IMAGE_DATA](#)
- [N_COLORS](#)
- [PALETTE](#)
- [QUALITY](#)
- [REGISTER_PROPERTIES](#)
- [RESOLUTION](#)
- [SCREEN_DIMENSIONS](#)

- [UNITS](#)
- [ZBUFFER_DATA](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrBuffer::Cleanup](#)
- [IDLgrBuffer::Draw](#)
- [IDLgrBuffer::Erase](#)
- [IDLgrBuffer::GetContiguousPixels](#)
- [IDLgrBuffer::GetDeviceInfo](#)
- [IDLgrBuffer::GetFontnames](#)
- [IDLgrBuffer::GetProperty](#)
- [IDLgrBuffer::GetTextDimensions](#)
- [IDLgrBuffer::Init](#)
- [IDLgrBuffer::PickData](#)
- [IDLgrBuffer::Read](#)
- [IDLgrBuffer::Select](#)
- [IDLgrBuffer::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrBuffer Properties

IDLgrBuffer objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrBuffer::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrBuffer::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrBuffer::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the retrievable properties associated with this object (except IMAGE_DATA and ZBUFFER_DATA).

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR_MODEL

A Boolean value that determines the color model to be used for the buffer:

- 0 = RGB (default)
- 1 = Color Index

In a property sheet, this property appears as an enumerated list with the following options:

- RGB
- Indexed

Property Type	ENUMLIST		
Name String	Color model		
Get: Yes	Set:	Init: Yes	Registered: Yes

DIMENSIONS

A two-element integer vector of the form [*width, height*] that specifies the dimensions of the buffer in units specified by the UNITS property. The default is [640,480].

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GRAPHICS_TREE

An object reference of type IDLgrScene, IDLgrViewgroup, or IDLgrView that specifies the graphics tree of this object. If this property is set to a valid object reference, calling the Draw method on the destination object with no arguments will cause the object reference associated with this property to be drawn. If this object is valid and the destination object is destroyed, this object reference will be destroyed as well. By default the GRAPHICS_TREE property is set equal to the null-object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IMAGE_DATA

A byte array representing the image that is currently rendered within the buffer. If the buffer uses an RGB color model, the returned array will have dimensions (3, *xdim*, *ydim*). If the window object uses an indexed color model, the returned array will have dimensions (*xdim*, *ydim*).

Property Type	Byte array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

N_COLORS

An integer that determines the number of colors (between 2 and 256) to be used if COLOR_MODEL is set to Color Index.

Property Type	INTEGER		
Name String	Number of colors		
Get: Yes	Set: No	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class) that specifies the red, green, and blue values to be loaded into the buffer's color lookup table.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

QUALITY

An integer that indicates the rendering quality at which graphics are to be drawn to the buffer. Valid values are:

- 0 = Low
- 1 = Medium
- 2 = High (default)

In a property sheet, this property appears as an enumerated list with the following options:

- Low
- Medium
- High

Property Type	ENUMLIST		
Name String	Draw quality		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESOLUTION

A two-element floating-point vector of the form [*xres*, *yres*] that specifies the device resolution in centimeters per pixel. This value is stored in double precision. The default value is: [0.035277778, 0.035277778] (72 DPI).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Resolution		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SCREEN_DIMENSIONS

A two-element integer vector of the form [*width*, *height*] that specifies the maximum allowed dimensions (measured in device units) for the buffer object.

Note

The maximum buffer dimension size is always 2048 by 2048.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

UNITS

An integer value that indicates the units of measure for the DIMENSIONS property. Valid values are:

- 0 = Device (default)
- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to 1600 x 1200

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZBUFFER_DATA

A floating-point array representing the zbuffer that is currently within the buffer. The returned array will have dimensions (*xdim*, *ydim*).

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLgrBuffer::Cleanup

The IDLgrBuffer::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrBuffer:]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrBuffer::Draw

The IDLgrBuffer::Draw procedure method draws the given picture to this graphics destination.

Note

Objects are drawn to the destination device in the order that they are added to the model, view, viewgroup, or scene object that contains them.

Syntax

```
Obj->[IDLgrBuffer::]Draw [, Picture] [, CREATE_INSTANCE={1 | 2}]  
[, /DRAW_INSTANCE]
```

Arguments

Picture

The view (an instance of an [IDLgrView](#) object), viewgroup (an instance of an [IDLgrViewgroup](#) object) or scene (an instance of an [IDLgrScene](#) object) to be drawn.

Keywords

CREATE_INSTANCE

Set this keyword equal to one to specify that this scene or view is the unchanging part of a drawing. Some destinations can make an instance from the current window contents without having to perform a complete redraw. If the view or scene to be drawn is identical to the previously drawn view or scene, this keyword can be set equal to 2 to hint the destination to create the instance from the current window contents if it can.

DRAW_INSTANCE

Set this keyword to specify that this scene, viewgroup, or view is the changing part of the drawing. It is overlaid on the result of the most recent CREATE_INSTANCE draw.

Version History

Introduced: 5.0

IDLgrBuffer::Erase

The IDLgrBuffer::Erase procedure method erases this graphics destination.

Syntax

Obj->[IDLgrBuffer:]Erase [, **COLOR**=*index or RGB vector*]

Arguments

None

Keywords

COLOR

Set this keyword to the color to be used for the erase. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*].

Version History

Introduced: 5.0

IDLgrBuffer::GetContiguousPixels

The IDLgrBuffer::GetContiguousPixels function method returns an array of long integers whose length is equal to the number of colors available in the index color mode (that is, the value of the N_COLORS property).

The returned array marks contiguous pixels with the ranking of the range's size. This means that within the array, the elements in the largest available range are set to zero, the elements in the second-largest range are set to one, etc. Use this range to set an appropriate colormap for use with the SHADE_RANGE property of the [IDLgrSurface](#) and [IDLgrPolygon](#) object classes.

To get the largest contiguous range, you could use the following IDL command:

```
result = obj->GetContiguousPixels()  
Range0 = WHERE(result EQ 0)
```

A contiguous region in the colormap can be increasing or decreasing in values. The following would be considered contiguous:

```
[ 0, 1, 2, 3, 4 ]  
[ 4, 3, 2, 1, 0 ]
```

Syntax

Result = *Obj*->[IDLgrBuffer::]GetContiguousPixels()

Return Value

Returns an array of long integers whose length is equal to the number of colors available in the index color mode.

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrBuffer::GetDeviceInfo

The IDLgrBuffer::GetDeviceInfo procedure method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=1 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Obj->[IDLgrBuffer:]GetDeviceInfo [, ALL=variable]
    [, MAX_NUM_CLIP_PLANES=variable]
    [, MAX_TEXTURE_DIMENSIONS=variable]
    [, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]
    [, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable]
```

Arguments

None

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_NUM_CLIP_PLANES

Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_TEXTURE_DIMENSIONS contains a two-element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_VIEWPORT_DIMENSIONS contains a two-element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, NAME contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, NUM_CPUS contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The NUM_CPUS keyword accurately returns the number of CPUs for the SGI IRIX, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, VENDOR contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, VERSION contains the version of the rendering device driver as a string.

Version History

Introduced: 5.0

MAX_NUM_CLIP_PLANES keyword added: 5.6

IDLgrBuffer::GetFontnames

The `IDLgrBuffer::GetFontnames` function method returns the list of available fonts that can be used in `IDLgrFont` objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned. See [Appendix H, “Fonts”](#) for more information.

Syntax

```
Result = Obj->[IDLgrBuffer::]GetFontnames( FamilyName[, IDL_FONTS={0 | 1 | 2  
}] [, STYLES=string] )
```

Return Value

Returns the list of available fonts that can be used in `IDLgrFont` objects.

Arguments

FamilyName

A string representing the name of the font family to which all of the returned fonts must belong. The string may be a fully specified family name—such as “Helvetica”. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. To return all available family names, use “*”.

Keywords

IDL_FONTS

Set this keyword to specify where to search for fonts that IDL may use. Set `IDL_FONT` to 1 to select only fonts installed by IDL and to 2 to select only fonts detected in the host operating system. The default value is 0, specifying that both IDL and operating system fonts should be returned.

STYLES

Set this keyword to a string specifying the styles that are to be matched by the returned font names. You can set `STYLES` to a fully specified style string, such as “Bold Italic”. If you set `STYLES` to the null string, ‘’, only fontnames without style modifiers will be returned. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. The default value is

the string, “*”, which returns all fontnames containing the *FamilyName* argument, with or without style modifiers.

Version History

Introduced: 5.0

IDLgrBuffer::GetProperty

The IDLgrBuffer::GetProperty procedure method retrieves the value of a property or group of properties for the buffer.

Syntax

Obj->[IDLgrBuffer:]GetProperty[, *PROPERTY=variable*]

Keywords

Any property listed under “[IDLgrBuffer Properties](#)” on page 3574 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s super-class.

Version History

Introduced: 5.0

IDLgrBuffer::GetTextDimensions

The `IDLgrBuffer::GetTextDimensions` function method retrieves the dimensions of a text or axis object that will be rendered in a window. The result is a 3-element double-precision floating-point vector [*xDim*, *yDim*, *zDim*] representing the dimensions of the text or axis object, measured in data units. If the object specified is an axis object, the result encompasses the tick labels and the title of the axis (if any).

Syntax

```
Result = Obj->[IDLgrBuffer::]GetTextDimensions( TextObj [, DESCENT=variable]  
[, PATH=objref(s)] )
```

Return Value

Returns the dimensions of a text or axis object that will be rendered in a window.

Arguments

TextObj

The object reference to a text or axis object for which text dimensions are requested.

Keywords

DESCENT

Set this keyword equal to a named variable that will contain an array of double-precision floating-point values (one for each string in the `IDLgrText` object). The values represent the distance to travel (parallel to the `UPDIR` vector) from the text baseline to reach the bottom of the lowest descender in the string. All values will be negative numbers, or zero. This keyword is valid only if *TextObj* is an `IDLgrText` object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the text dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this keyword is not set, the `PARENT` properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If

IDLgrBuffer::GetTextDimensions is called from within a Draw method and the PATH keyword is not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in IDLgrModel::Add.

Version History

Introduced: 5.0

IDLgrBuffer::Init

The IDLgrBuffer::Init function method initializes the buffer object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrBuffer' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrBuffer::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrBuffer Properties](#)” on page 3574 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrBuffer::PickData

The IDLgrBuffer::PickData function method maps a point in the two-dimensional device space of the buffer to a point in the three-dimensional data space of an object tree. The resulting 3-D data space coordinates are returned in a user-specified variable. The PickData function returns one if the specified location in the buffer's device space "hits" a graphic object, or zero otherwise.

Note

The PickData method relies on the contents of the depth buffer at the time it is called to compute and return its results. Be sure that the depth buffer contents are appropriate for getting the expected results from PickData. For example, you may use the DEPTH_WRITE_DISABLE property of an object to prevent an object from modifying the depth buffer as it is drawn. This prevents the object from affecting the result returned by the PickData method.

Syntax

```
Result = Obj->[IDLgrBuffer::]PickData( View, Object, Location, XYZLocation  
    [, DIMENSIONS=[width,height]][, PATH=objref(s)]  
    [, PICK_STATUS=variable])
```

Return Value

Returns one if the specified location in the buffer's device space "hits" a graphic object, or zero otherwise.

Arguments

View

The object reference of an IDLgrView object that contains the object being picked.

Object

The object reference of a model or atomic graphic object from which the data space coordinates are being requested.

Note

Even though you can specify any object for this argument, the location that is returned may not lie on this object. If another object is in front of the specified

object then the returned location lies on the object in front. However, the coordinate system used is that of the object specified by this argument.

Location

A two-element vector $[x, y]$ specifying the location in the buffer's device space of the point to pick data from.

XYZLocation

A named variable that will contain the three-dimensional double-precision floating-point data space coordinates of the picked point. Note that the value returned in this variable is a location, not a data value.

Note

If the atomic graphic object specified as the target has been transformed using either the `LOCATION` or `DIMENSIONS` properties (this is only possible with `IDLgrAxis`, `IDLgrImage`, and `IDLgrText` objects), these transformations will *not* be included in the data coordinates returned by the `PickData` function. This means that you may need to re-apply the transformation accomplished by specifying `LOCATION` or `DIMENSIONS` once you have retrieved the data coordinates with `PickData`. This situation does not occur if you transform the axis, text, or image object using the `[XYZ]COORD_CONV` properties.

Keywords

DIMENSIONS

Set this keyword to a two-element array $[w, h]$ to specify data picking should occur for all device locations that fall within a *pick box* of these dimensions. The pick box will be centered about the coordinates $[x, y]$ specified in the *Location* argument, and will occupy the rectangle defined by:

$$(x-(w/2), y-(h/2)) - (x+(w/2), y+(h/2))$$

By default, the pick box covers a single pixel. The array returned via the *XYZLocation* argument will have dimensions $[3, w, h]$.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a data space coordinate. Each path object reference specified with this keyword must

contain an alias. The data space coordinate is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in IDLgrModel::Add.

PICK_STATUS

Set this keyword to a named variable that will contain “hit” information for each pixel in the pick box. If the DIMENSIONS keyword is not set, the PICK_STATUS will be a scalar value exactly matching the *Result* of the method call. If the DIMENSIONS keyword is set, the PICK_STATUS variable will be an array matching the dimensions of the pick box. Each value in the PICK_STATUS array corresponds to a pixel in the pick box, and will be set to one of the following values:

Value	Description
-1	The pixel falls outside of the window’s viewport.
0	No graphic object is “hit” at that pixel location.
1	A graphic object is “hit” at that pixel location.

Table 8-3: PICK_STATUS Keyword Values

Version History

Introduced: 5.0

PICK_STATUS keyword added: 5.6

IDLgrBuffer::Read

The IDLgrBuffer::Read function method reads an image from a buffer. The returned value is an instance of the [IDLgrImage](#) object class.

Syntax

Result = *Obj*->[IDLgrBuffer::]Read()

Return Value

Returns an instance of the [IDLgrImage](#) object class.

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrBuffer::Select

The IDLgrBuffer::Select function method returns a list of objects selected at a specified location. If no objects are selected, the Select function returns -1.

Note

IDL returns a maximum of 512 objects. This maximum may be smaller if any of the objects are contained in deep model hierarchies. Because of this limit, it is possible that not all objects eligible for selection will appear in the list.

Syntax

```
Result = Obj->[IDLgrBuffer::]Select(Picture, XY [, DIMENSIONS=[width, height]]  
[, /ORDER] [, SUB_SELECTION=variable] [, UNITS={0 | 1 | 2 | 3}])
```

Return Value

Returns a list of objects selected at a specified location. Returns -1 if no objects are selected.

Arguments

Picture

The view, viewgroup, or scene (an instance of the [IDLgrView](#), IDLgrViewgroup, or [IDLgrScene](#) class) whose children are among the candidates for selection.

If the first argument is a scene or viewgroup, then the returned object list will contain one or more views. If the first argument is a view, the list will contain atomic graphic objects (or model objects which have their SELECT_TARGET property set). Objects are returned in order, according to their distance from the viewer. The closer an object is to the viewer, the lower its index in the returned object list. If multiple objects are at the same distance from the viewer (views in a scene or 2-D geometry), the first object drawn will appear at a lower index in the list. (The ORDER keyword can be used to change this behavior.)

XY

A two-element array defining the center of the selection box in device space. By default, the selection box is 3 pixels by 3 pixels.

Keywords

DIMENSIONS

Set this keyword to a two-element array $[w, h]$ to specify that the selection box will have a width w and a height h , and will be centered about the coordinates $[x, y]$ specified in the XY argument. The box occupies the rectangle defined by:

$$(x-(w/2), y-(h/2)) - (x+(w/2), y+(h/2))$$

Any object which intersects this box is considered to be selected. By default, the selection box is 3 pixels by 3 pixels.

ORDER

Set this keyword to control how objects that are the same distance from the viewer are ordered in the selection list. If $ORDER=0$ (the default), the order of objects in the selection list will be the same as the order in which the objects are drawn. If $ORDER=1$, the order of objects in the selection list will be the reverse of the order in which they are drawn. This keyword has no affect on the ordering of objects that are not at the same distance from the viewer.

Tip

If you are using $DEPTH_TEST_FUNCTION=4$ (“less than or equal”) on your graphics objects, set $ORDER=1$ to return objects at the same depth in the order in which they appear visually.

SUB_SELECTION

Set this keyword to a named variable that upon return will contain a vector of references to the graphic objects that were sub-selected at the given location. This vector includes all graphic objects that are:

- Graphic objects contained by the nearest selected object (the first object in the return value)
- Select targets (IDLgrModel objects with the $SELECT_TARGET$ property set)
- Graphic objects displayed at the given XY location

UNITS

Set this keyword to indicate the units of measure. Valid values are:

- 0 = Device (default)

- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to the dimensions of the graphics destination.

Version History

Introduced: 5.0

SUB_SELECTION keyword added: 6.1

IDLgrBuffer:: SetProperty

The IDLgrBuffer::SetProperty procedure method sets the value of a property or group of properties for the buffer.

Syntax

Obj->[IDLgrBuffer:]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrBuffer Properties](#)” on page 3574 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrClipboard

An IDLgrClipboard object sends Object Graphics output to the operating system native clipboard or to a file in bitmap or vector format. The FILENAME, VECTOR, and POSTSCRIPT keyword values in the IDLgrClipboard::Draw method, and the platform on which the call is issued determine the file type and destination. See [IDLgrClipboard::Draw](#) for details.

Note

Objects or subclasses of this type can not be saved or restored.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrClipboard::Init](#)” on page 3624.

Properties

Objects of this class have the following properties. See “[IDLgrClipboard Properties](#)” on page 3602 for details on individual properties.

- [ALL](#)
- [COLOR_MODEL](#)
- [DIMENSIONS](#)
- [GRAPHICS_TREE](#)
- [N_COLORS](#)
- [PALETTE](#)
- [QUALITY](#)
- [REGISTER_PROPERTIES](#)
- [RESOLUTION](#)
- [SCREEN_DIMENSIONS](#)
- [UNITS](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrClipboard::Cleanup](#)
- [IDLgrClipboard::Draw](#)
- [IDLgrClipboard::GetContiguousPixels](#)
- [IDLgrClipboard::GetDeviceInfo](#)
- [IDLgrClipboard::GetFontnames](#)
- [IDLgrClipboard::GetProperty](#)
- [IDLgrClipboard::GetTextDimensions](#)
- [IDLgrClipboard::Init](#)
- [IDLgrClipboard::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrClipboard Properties

IDLgrClipboard objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrClipboard::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrClipboard::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrClipboard::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure that contains the values of all of the retrievable properties associated with this object.

Property Type	Anonymous structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR_MODEL

An integer value that determines whether to use the indexed color model for the clipboard buffer:

- 0 = RGB (default)
- 1 = Color Index

In a property sheet, this property appears as an enumerated list with the following options:

- RGB
- Indexed

Property Type	ENUMLIST		
Name String	Color model		
Get: Yes	Set: No	Init: Yes	Registered: Yes

DIMENSIONS

A two-element integer vector of the form [*width, height*] to specify the dimensions of the clipboard buffer in units specified by the UNITS property. The default is [640,480].

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GRAPHICS_TREE

An object reference of type IDLgrScene, IDLgrViewgroup, or IDLgrView that specifies the graphic tree of this object. If this property is set to a valid object reference, calling the Draw method on the destination object with no arguments will cause the object reference associated with this property to be drawn. If this object is valid and the destination object is destroyed, this object reference will be destroyed as well. By default the GRAPHICS_TREE property is set equal to the null-object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_COLORS

An integer value that determines the number of colors (between 2 and 256) to be used if COLOR_MODEL is set to Color Index.

Property Type	INTEGER		
Name String	Number of colors		
Get: Yes	Set: No	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class) that specifies the red, green, and blue values that are to be loaded into the clipboard buffer's color lookup table.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

QUALITY

An integer indicating the rendering quality at which graphics are to be drawn to the clipboard buffer. Valid values are:

- 0 = Low
- 1 = Medium
- 2 = High (default)

In a property sheet, this property appears as an enumerated list with the following options:

- Low
- Medium
- High

Property Type	ENUMLIST		
Name String	Draw quality		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the

object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESOLUTION

A two-element floating-point vector of the form [*xres*, *yres*] specifying the device resolution in centimeters per pixel. This value is stored in double precision. The default value is: [0.035277778, 0.035277778] (72 DPI).

Note

To match screen rendering on an IDLgrClipboard object, the following properties should be matched between the devices: DIMENSIONS, UNITS, RESOLUTION, COLOR_MODEL and N_COLORS.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Resolution		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SCREEN_DIMENSIONS

A two-element integer vector of the form [*width*, *height*] that specifies the maximum allowed dimensions (measured in device units) for the clipboard object.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

UNITS

An integer value that indicates the units of measure for the DIMENSIONS property. Valid values are:

- 0 = Device (default)

- 1 = Inches
- 2 = Centimeters
- 3 = Normalized (relative to 1600 x 1200)

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLgrClipboard::Cleanup

The IDLgrClipboard::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrClipboard::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrClipboard::Draw

The IDLgrClipboard::Draw procedure method draws the given picture to this graphics destination.

Writing to a File

The file type produced when the IDLgrClipboard::Draw method is passed an IDLgrView, IDLgrViewgroup, or IDLgrScene object varies depending upon keyword settings and the platform on which the call is issued. If the FILENAME keyword is set to a non-empty string, the name of the file IDL creates is specified by the string. If the FILENAME keyword is a non-zero, numeric value, IDL creates a file named `idl.ext` where `ext` is replaced with the appropriate extension shown in parentheses in the following table.

Keyword Settings	Windows File Type	UNIX File Type
VECTOR = 1, POSTSCRIPT = 1	Encapsulated PostScript (EPS)	Encapsulated PostScript (EPS)
VECTOR = 1, POSTSCRIPT = 0	Enhanced MetaFile (EMF)	Encapsulated PostScript (EPS)
VECTOR = 0, POSTSCRIPT = 1	Encapsulated PostScript (EPS)	Encapsulated PostScript (EPS)
VECTOR = 0, POSTSCRIPT = 0	Bitmap (BMP)	Encapsulated PostScript (EPS)

Table 8-4: File Types Produced by IDLgrClipboard Draw Method

Writing to the Clipboard

Objects can be written to the operating system clipboard using IDLgrClipboard::Draw. When the FILENAME keyword equals an empty string (""), equals 0 (zero), or is not specified, the output is written to the clipboard.

Note

The IDLgrClipboard object empties the Windows clipboard before writing to it.

Note

When VECT_SORTING=0, objects are drawn to the destination device in the order that they are added to the model, view, viewgroup, or scene object that contains

them. Otherwise, objects are drawn to the destination device based on their average depth value.

Syntax

```
Obj->[IDLgrClipboard::]Draw [, Picture] [, /CMYK] [, FILENAME=string]  
    [, POSTSCRIPT=value] [, VECT_SHADING={ 0 | 1 } ]  
    [, VECT_SORTING={ 0 | 1 } ]  
    [, VECT_TEXT_RENDER_METHOD={ 0 | 1 } ] [, VECTOR={ 0 | 1 } ]
```

Arguments

Picture

The view (an instance of an [IDLgrView](#) object), viewgroup (an instance of an [IDLgrViewgroup](#) object), or scene (an instance of an [IDLgrScene](#) object) to be drawn.

Keywords

CMYK

This keyword has an effect only when the output format is Encapsulated PostScript (POSTSCRIPT=1). Set this keyword to create a PostScript file using the CMYK (cyan, magenta, yellow, and black) color model. The default is a PostScript file using the RGB (red, green, blue) color model. This keyword applies to both bitmap and vector graphic PostScript files.

The CMYK format is a subtractive color model that is better suited for color printing. This color model allows some document processing applications and printer control software to easily adjust the overall brightness of an image by manipulating a single channel (K). PostScript files created with the CMYK color model must be processed by an interpreter capable of interpreting PostScript Language Level 2 or higher.

FILENAME

To write to a file, set this keyword as follows:

- Set to a string that represents the name of the file to which the output should be written.

- Set to any non-zero numerical value. IDL generates a filename based upon the values of the POSTSCRIPT and VECTOR keywords, and the platform. See [“Writing to a File”](#) in the introductory section for details.

If the value is an empty string (" "), equals 0 (zero), or is not specified, the output is written to the clipboard.

Note

You are strongly encouraged to specify a string for this value. The ability to specify a numerical value is maintained for backward compatibility, but its usage should be deprecated.

POSTSCRIPT

Set this keyword to a non-zero value to indicate that the generated output should be in Encapsulated PostScript format. By default, the generated output is in Bitmap format on Windows platforms and Encapsulated PostScript on UNIX platforms.

VECT_SHADING

This keyword has an effect only when generating vector output (VECTOR=1) and when the output format is Encapsulated PostScript (POSTSCRIPT=1).

This keyword controls the appearance of smooth (Gouraud) shaded IDLgrPolygon and IDLgrSurface objects. Valid values are:

- 0 = disable smooth shading. This causes all polygons and surfaces to be rendered with flat shading. This may be valuable when using slower PostScript interpreters or when memory and stack space are limited. Smoothly shaded polygons require more resources as well as time.
- 1 = enable smooth shading (default). This renders smooth (Gouraud) shaded polygons and surfaces in the Encapsulated PostScript file.

Note

Polygons and surfaces in Enhanced MetaFiles (EMF) files will be rendered using flat shading. Only the output in Encapsulated PostScript (EPS) files is affected by this keyword, and only when the VECTOR keyword has been set. See [“Bitmap and Vector Graphic Output”](#) in Chapter 34 of the *Using IDL* manual for details about vector graphic output.

VECT_SORTING

This keyword has an effect only when generating vector output (VECTOR=1).

This keyword controls the way object primitives in a picture appear in the destination. Valid values are:

- 0 = disable sorting. The object primitives appear in the vector output file in the same order they are drawn on the display device. This is the order in which they appear in the graphics tree.
- 1 = enable sorting (default). The objects are ordered from back to front based on each primitive object's average depth value.

Note

Objects that intersect each other, IDLgrImage objects (which do not update the depth buffer), and objects contained within transparent views might not sort correctly. See [“Controlling What is Displayed in Vector Graphics”](#) in Chapter 34 of the *Using IDL* manual.

VECT_TEXT_RENDER_METHOD

This keyword has an effect only when generating vector output (VECTOR=1).

This keyword controls the way text is rendered in a vector graphic file. Valid values are:

- 0 = render text as text primitives (default). The output device renders the text as text primitives. This allows the text to be edited by object-based graphics programs.
- 1 = render text as triangles. The text is rendered as a series of filled triangles in the vector file. This produces text that closely matches the text on the display device, but the output file size is larger, and the text can not be easily edited.

Note

The results of attempting to edit the text primitives depends upon the sophistication of the graphics program. An Enhanced MetaFile (EMF) inserted into a Microsoft Word document can be edited. However, not all versions of Microsoft Word support advanced 3-D graphic primitives such as those associated with obliquely or vertically aligned text. Choosing to edit a file with non-horizontally aligned text may result in the text being flattened into two dimensions. Typically, each letter becomes its own string and alignment is altered. To edit non-horizontal text and preserve the original quality, create an Encapsulated PostScript (EPS) file that can be modified in a more sophisticated object-oriented image editing program.

Note

See [“Text Rendering in Vector Graphics”](#) in Chapter 34 of the *Using IDL* manual for more information.

VECTOR

Set this keyword to indicate the type of graphics primitives generated. Valid values are:

- 0 = generate bitmap graphics (default). Render the scene to a buffer and then copy the buffer to the clipboard or file in bitmap format. The bitmap retains the quality of the original image, but the bitmaps cannot be effectively scaled on all devices.
- 1 = generate vector graphics. Render the scene using simple vector operations that result in a representation of the scene that is scalable. The vector representation does not retain all the attributes of the original image, but it can be effectively scaled on other devices. On Windows, the representation is the Windows Enhanced Metafile (EMF). On UNIX platforms, the representation is Encapsulated PostScript (EPS).

Note

Transparent objects in a view are not rendered in vector graphic files. Semi-transparent objects are rendered fully opaque. See [“Bitmap and Vector Graphic Output”](#) in Chapter 34 of the *Using IDL* manual for a complete discussion of when to output to bitmap or vector graphics based on picture content.

Examples

This example demonstrates the process of copying the contents of an IDL graphics display object (a buffer or a window) to the system clipboard, where it becomes available for pasting into another application. The example also uses the `IDLgrClipboard::Draw` method to create a platform-dependent file type in the current directory.

```
PRO SendingPlotToClipboard

; Determine the path to the "damp_sn2.dat" file.
signalFile = FILEPATH('damp_sn2.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize the parameters of the data within the file.
signalSize = 512
```

```

signal = BYTARR(signalSize, /NOZERO)

; Open the file, read in data, and then close the file.
OPENR, unit, signalFile, /GET_LUN
READU, unit, signal
FREE_LUN, unit

; Determine viewplane size and margins.
offsetScale = 150.
viewOffset = offsetScale*[-1., -1., 1., 1.]
signalRange = MAX(signal) - MIN(signal)

; Initialize the display objects.
windowSize = [512, 384]
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'Damped Sine Wave with Noise')

; Use an IDLgrViewgroup as the picture rather than
; an IDLgrView. This is used as a container for the
; "loose" objects, such as the IDLgrText objects that
; are the axis titles.
oViewgroup = OBJ_NEW('IDLgrViewgroup')
oWindow->SetProperty, GRAPHICS_TREE = oViewgroup

; Add an IDL container to the viewgroup to hold text objects.
oContainer = OBJ_NEW('IDL_CONTAINER')
oViewgroup->Add, oContainer

oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., signalSize, signalRange] + $
    viewOffset)
oViewgroup->Add, oView
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel

; Initialize the plot object.
oPlot = OBJ_NEW('IDLgrPlot', signal, COLOR = [0, 0, 255])
oModel->Add, oPlot

; Obtain plot ranges.
oPlot->GetProperty, XRANGE = xPlotRange, $
    YRANGE = yPlotRange

; Initialize axes objects, which are based on the plot
; ranges.
oXTitle = OBJ_NEW('IDLgrText', 'Time (seconds)')
oContainer->Add, oXTitle
oXAxis = OBJ_NEW('IDLgrAxis', 0, RANGE = xPlotRange, $

```

```

        LOCATION = [xPlotRange[0], yPlotRange[0]], /EXACT, $
        TITLE = oXTitle, TICKDIR = 0, $
        TICKLEN = (0.02*(yPlotRange[1] - yPlotRange[0]))
oModel->Add, oXAxis
oYTitle = OBJ_NEW('IDLgrText', 'Amplitude (centimeters)')
oContainer->Add, oYTitle
oYAxis = OBJ_NEW('IDLgrAxis', 1, RANGE = yPlotRange, $
        LOCATION = [xPlotRange[0], yPlotRange[0]], /EXACT, $
        TITLE = oYTitle, TICKDIR = 0, $
        TICKLEN = (0.02*(xPlotRange[1] - xPlotRange[0])))
oModel->Add, oYAxis
oModel->Translate, -50., -50., 0.
oWindow->Draw

; Determine the centimeter to pixel resolution of the
; plot on the screen.
oWindow->GetProperty, RESOLUTION = screenResolution

; Initialize clipboard destination object.
oClipboard = OBJ_NEW('IDLgrClipboard', QUALITY = 2, $
        DIMENSIONS = windowSize, $
        RESOLUTION = screenResolution, $
        GRAPHICS_TREE = oViewgroup)

; Determine the type of export file, which depends on
; the screen device.
screenDevice = !D.NAME
CASE screenDevice OF
'X': fileExtension = '.ps'
'WIN': fileExtension = '.emf'
ELSE: BEGIN
        OBJ_DESTROY, [oWindow, oClipboard]
        RETURN
        END
ENDCASE
clipboardFile = 'damp_sn2' + fileExtension

; Display the view within the clipboard destination,
; which exports to a PS or EMF.
oClipboard->Draw, FILENAME = clipboardFile, $
        /VECTOR
oClipboard->Draw, FILENAME = 'damp_sn2.eps', $
        /POSTSCRIPT, /VECTOR

```

```
; Cleanup object references. Before destroying the
; IDLgrClipboard object, reset its GRAPHICS_TREE
; to NULL so the IDLgrViewgroup can remain associated
; with the IDLgrWindow. Interactive destruction
; of the IDLgrWindow will free the other graphics objects
; implicitly.
oClipboard->SetProperty, GRAPHICS_TREE = OBJ_NEW()
OBJ_DESTROY, oClipboard
```

END

Version History

Introduced: 5.1

CMYK, VECT_SHADING, VECT_SORTING, and
VECT_TEXT_RENDER_METHOD keywords added: 6.1

IDLgrClipboard::GetContiguousPixels

The IDLgrClipboard::GetContiguousPixels function method returns an array of long integers whose length is equal to the number of colors available in the index color mode (that is, the value of the N_COLORS property).

The returned array marks contiguous pixels with the ranking of the range's size. This means that within the array, the elements in the largest available range are set to zero, the elements in the second-largest range are set to one, etc. Use this range to set an appropriate colormap for use with the SHADE_RANGE property of the IDLgrSurface and IDLgrPolygon object classes.

To get the largest contiguous range, you could use the following IDL command:

```
result = obj->GetContiguousPixels()  
Range0 = WHERE(result EQ 0)
```

A contiguous region in the colormap can be increasing or decreasing in values. The following would be considered contiguous:

```
[ 0, 1, 2, 3, 4 ]  
[ 4, 3, 2, 1, 0 ]
```

Syntax

```
Result = Obj->[IDLgrClipboard:]GetContiguousPixels()
```

Return Value

Returns an array of long integers whose length is equal to the number of colors available in the index color mode.

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrClipboard::GetDeviceInfo

The IDLgrClipboard::GetDeviceInfo procedure method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=1 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Obj->[IDLgrClipboard::]GetDeviceInfo [, ALL=variable]  
    [, MAX_NUM_CLIP_PLANES=variable]  
    [, MAX_TEXTURE_DIMENSIONS=variable]  
    [, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
    [, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable]
```

Arguments

None

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_NUM_CLIP_PLANES

Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_TEXTURE_DIMENSIONS contains a two element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_VIEWPORT_DIMENSIONS contains a two element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, NAME contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, NUM_CPUS contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The NUM_CPUS keyword accurately returns the number of CPUs for the SGI IRIX, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, VENDOR contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, VERSION contains the version of the rendering device driver as a string.

Version History

Introduced: 5.1

MAX_NUM_CLIP_PLANES keyword added: 5.6

IDLgrClipboard::GetFontnames

The IDLgrClipboard::GetFontnames function method returns the list of available fonts that can be used in IDLgrFont objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned; see [Appendix H, “Fonts”](#) for more information.

Syntax

```
Result = Obj->[IDLgrClipboard:]GetFontnames( FamilyName [, IDL_FONTS={0 | 1 | 2}] [, STYLES=string] )
```

Return Value

Returns the list of available fonts that can be used in IDLgrFont objects.

Arguments

FamilyName

A string representing the name of the font family to which all of the returned fonts must belong. The string may be a fully specified family name—such as “Helvetica”. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. To return all available family names, use “*”.

Keywords

IDL_FONTS

Set this keyword to specify where to search for fonts that IDL may use. Set IDL_FONT to 1 to select only fonts installed by IDL and to 2 to select only fonts detected in the host operating system. The default value is 0, specifying that both IDL and operating system fonts should be returned.

STYLES

Set this keyword to a string specifying the styles that are to be matched by the returned font names. You can set STYLES to a fully specified style string, such as “Bold Italic”. If you set STYLES to the null string, ' ', only fontnames without style modifiers will be returned. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. The default value is

the string, “*”, which returns all fontnames containing the *FamilyName* argument, with or without style modifiers.

Version History

Introduced: 5.1

IDLgrClipboard::GetProperty

The IDLgrClipboard::GetProperty procedure method retrieves the value of a property or group of properties for the clipboard buffer.

Syntax

Obj->[IDLgrClipboard::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrClipboard Properties](#)” on page 3602 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrClipboard::GetTextDimensions

The IDLgrClipboard::GetTextDimensions function method retrieves the dimensions of a text or axis object that will be rendered in the clipboard buffer. The result is a 3-element double-precision floating-point vector [*xDim*, *yDim*, *zDim*] representing the dimensions of the text or axis object, measured in data units. If the object specified is an axis object, the result encompasses the tick labels and the title of the axis (if any).

Syntax

```
Result = Obj->[IDLgrClipboard:]GetTextDimensions( TextObj  
[, DESCENT=variable] [, PATH=objref(s)] )
```

Return Value

Returns the dimensions of a text or axis object that will be rendered in the clipboard buffer.

Arguments

TextObj

The object reference to a text or axis object for which the text dimensions are requested.

Keywords

DESCENT

Set this keyword equal to a named variable that will contain an array of double-precision floating-point values (one for each string in the IDLgrText object). The values represent the distance to travel (parallel to the UPDIR vector) from the text baseline to reach the bottom of the lowest descender in the string. All values will be negative numbers, or zero. This keyword is valid only if *TextObj* is an IDLgrText object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the text dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this

keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If `IDLgrClipboard::GetTextDimensions` is called from within a Draw method and the PATH keyword is not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

Version History

Introduced: 5.1

IDLgrClipboard::Init

The IDLgrClipboard::Init function method initializes the clipboard object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrClipboard' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrClipboard::]Init([PROPERTY=value]) (Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrClipboard Properties](#)” on page 3602 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

IDLgrClipboard:: SetProperty

The IDLgrClipboard::SetProperty procedure method sets the value of a property or group of properties for the clipboard buffer.

Syntax

Obj->[IDLgrClipboard::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under [“IDLgrClipboard Properties”](#) on page 3602 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrColorbar

The IDLgrColorbar object consists of a color-ramp with an optional framing box and annotation axis. The object can be horizontal or vertical.

An IDLgrColorbar object is a *composite object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

This object class is implemented in the IDL language. Its source code can be found in the file `idlgrcolorbar__define.pro` in the `lib` subdirectory of the IDL distribution.

Superclasses

[IDLgrModel](#)

[IDLitComponent](#)

Creation

See “[IDLgrColorbar::Init](#)” on page 3644.

Properties

Objects of this class have the following properties. See “[IDLgrColorbar Properties](#)” on page 3630 for details on individual properties.

- [ALL](#)
- [BLUE_VALUES](#)
- [COLOR](#)
- [DIMENSIONS](#)
- [GREEN_VALUES](#)
- [HIDE](#)
- [MAJOR](#)
- [MINOR](#)
- [PALETTE](#)
- [PARENT](#)

- RED_VALUES
- SHOW_AXIS
- SHOW_OUTLINE
- SUBTICKLEN
- THICK
- THREED
- TICKFORMAT
- TICKFRMTDATA
- TICKLEN
- TICKTEXT
- TICKVALUES
- TITLE
- XCOORD_CONV
- XRANGE
- YCOORD_CONV
- YRANGE
- ZCOORD_CONV
- ZRANGE

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrColorbar::Cleanup`
- `IDLgrColorbar::ComputeDimensions`
- `IDLgrColorbar::GetProperty`
- `IDLgrColorbar::Init`
- `IDLgrColorbar::SetProperty`

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrColorbar Properties

IDLgrColorbar objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrColorbar::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrColorbar::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrColorbar::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

BLUE_VALUES

A byte vector containing the blue values for the color palette. Setting this value is the same as specifying the `aBlue` argument to the `IDLgrColorbar::Init` method.

Property Type	Byte vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

The color to be used as the foreground color for the axis and outline box. The color may be specified as a color lookup table index or as a three-element vector [*red*, *green*, *blue*]. The default is [0, 0, 0].

Property Type	Color		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DIMENSIONS

A two element integer vector [*dx*, *dy*] that specifies the size of the ramp display (not the axis) in pixels. If $dx > dy$, the colorbar is drawn horizontally with the axis placed below or above the ramp box depending on the value of the SHOW_AXIS property. If $dx < dy$, the colorbar is drawn vertically with the axis placed to the right or left of the ramp box depending on the value of the SHOW_AXIS property. The default value is [16,256].

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GREEN_VALUES

A byte vector containing the green values for the color palette. Setting this value is the same as specifying the *aGreen* argument to the IDLgrColorbar::Init method.

Property Type	Byte vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value to indicate whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MAJOR

An integer that represents the number of major tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MAJOR equal to zero suppresses major tickmarks entirely.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MINOR

An integer representing the number of minor tick marks. The default is -1, specifying that IDL will compute the number of tickmarks. Setting MINOR equal to zero suppresses minor tickmarks entirely.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PALETTE

An object reference to an instance of the IDLgrPalette object class. If this property is a valid object reference, the colors within the IDLgrPalette are used to specify the colors for the colorbar.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PARENT

An object reference to the object that contains this colorbar.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

RED_VALUES

A byte vector that contains the red values for the color palette. Setting this value is the same as specifying the *aRed* argument to the IDLgrColorbar::Init method.

Property Type	Byte vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SHOW_AXIS

An integer value that indicates whether the axis should be drawn:

- 0 = Do not display axis (the default)
- 1 = Display axis on left side or below the color ramp
- 2 = Display axis on right side or above the color ramp

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SHOW_OUTLINE

A Boolean value indicating whether the colorbar bounds should be outlined:

- 0 = Do not display outline (the default)
- 1 = Display outline

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SUBTICKLEN

A floating-point scale ratio specifying the length of minor tick marks relative to the length of major tick marks. The default is 0.5, specifying that the minor tick mark is one-half the length of the major tick mark.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0, specifying the line thickness used to draw the axis and box outline, in points. The default is 1.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THREED

A Boolean value that determines whether to create the colorbar as a graphic object that can be fully transformed in three dimensions on initialization. By default, the colorbar always faces the viewer and is drawn at $z=0$.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set:	Init: Yes	Registered: No

TICKFORMAT

Either a standard IDL format string (see “*Format Codes*” in Chapter 11 of the *Building IDL Applications* manual for details on format codes) or a string containing the name of a user-supplied function that returns a string to be used to format the axis tick mark labels. The function should accept integer arguments for the direction of the axis, the index of the tick mark, and the value of the tick mark, and should return a string to be used as the tick mark’s label. The function may optionally accept a keyword called DATA, which will be automatically set to the TICKFRMTDATA value. The default TICKFORMAT is "", the null string, which indicates that IDL will determine the appropriate format for each value.

Note

If the user-defined function’s name does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether TICKUNITS is specified. For more information, see “[XYZ]TICKFORMAT” on page 4345.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKFRMTDATA

A user-defined value of any type passed via the DATA keyword to the user-supplied formatting function specified via the TICKFORMAT property, if any. By default, this value is undefined, indicating that the DATA property will be undefined (and furthermore, need not be supported by the user-supplied function). Note that TICKFRMTDATA will not be included in the structure returned via the ALL property to the IDLgrColorbar::GetProperty method.

Property Type	Scalar of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKLEN

A floating-point value that specifies the length of each major tick mark, measured in data units. The recommended, and default, tick mark length is 0.2. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKTEXT

An object reference to either a single instance of the IDLgrText object class (with multiple strings) or to a vector of instances of the IDLgrText object class (each with a single string) that specifies the annotations to be assigned to the tick marks. By default, TICKTEXT is set to the NULL object, which indicates that IDL will compute tick annotations based upon the major tick values. The positions and orientation of the provided text object(s) may be overwritten by the colorbar.

Property Type	Object reference or object reference vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TICKVALUES

A floating-point vector of data values that represent the values at each tick mark. If TICKVALUES is set to 0, the default, IDL computes the tick values based on the axis range and the number of major ticks. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TITLE

An object reference to an instance of the IDLgrText object class to specify the title for the axis. The default is the null object, specifying that no title is drawn. The title will be centered along the axis, even if the text object itself has an associated location.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element double-precision floating-point vector of the form $[xmin, xmax]$ specifying the range of the x data coordinates covered by the colorbar.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors that converts Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Y = s_0 + s_1 * \text{Data}Y$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element double-precision floating-point vector of the form [*ymin*, *ymax*] that specifies the range of the *Y* data coordinates covered by the colorbar.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors that converts *Z* coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Z = s_0 + s_1 * \text{Data}Z$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ specifying the range of the Z data coordinates covered by the colorbar.

Until the colorbar is drawn to the destination object, the [XYZ]RANGE properties will be zero. Use the ComputeDimensions method on the colorbar object to get the data dimensions of the colorbar prior to a draw operation.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLgrColorbar::Cleanup

The IDLgrColorbar::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY,*Obj*

or

Obj->[IDLgrColorbar::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrColorbar::ComputeDimensions

The IDLgrColorbar::ComputeDimensions function method retrieves the dimensions of a colorbar object for the given destination object. The result is a three-element double-precision floating-point vector [*xDim*, *yDim*, *zDim*] representing the dimensions of the colorbar object measured in data units.

Syntax

```
Result = Obj->[IDLgrColorbar::]ComputeDimensions( DestinationObj  
[, PATH=objref(s)] )
```

Return Value

Returns the dimensions of a colorbar object for the given destination object.

Arguments

DestinationObject

The object reference to a destination object (IDLgrBuffer, IDLgrClipboard, IDLgrPrinter, or IDLgrWindow) for which the dimensions of the colorbar are being requested.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrColorbar::ComputeDimensions is called from within a Draw method and the PATH keyword is not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in IDLgrModel::Add.

Version History

Introduced: 5.1

IDLgrColorbar::GetProperty

The IDLgrColorbar::GetProperty procedure method retrieves the value of a property or group of properties for the colorbar.

Syntax

Obj->[IDLgrColorbar:]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrColorbar Properties](#)” on page 3630 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrColorbar::Init

The IDLgrColorbar::Init function method initializes the colorbar object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW( 'IDLgrColorbar' [, aRed, aGreen, aBlue] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrColorbar::]Init([aRed, aGreen, aBlue] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

aRed

A vector containing the red values for the color palette. These values should be within the range of $0 < Value < 255$. The number of elements comprising the *aRed* vector must not exceed 256.

aGreen

A vector containing the green values for the color palette. These values should be within the range of $0 < Value < 255$. The number of elements comprising the *aGreen* vector must not exceed 256.

aBlue

A vector containing the blue values for the color palette. These values should be within the range of $0 < Value < 255$. The number of elements comprising the *aBlue* vector must not exceed 256.

If *aRed*, *aGreen*, and *aBlue* are not provided, the color palette will default to a 256 entry greyscale ramp.

Keywords

Any property listed under “[IDLgrColorbar Properties](#)” on page 3630 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

IDLgrColorbar:: SetProperty

The IDLgrColorbar::SetProperty procedure method sets the value of a property or group of properties for the colorbar.

Syntax

Obj->[IDLgrColorbar::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrColorbar Properties](#)” on page 3630 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrContour

The IDLgrContour object draws a contour plot from data stored in a rectangular array or from a set of unstructured points. Both line contours and filled contour plots can be created.

An IDLgrContour object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

The object stores the following argument or property in double-precision if the `DOUBLE_DATA` property is specified, and in single-precision otherwise.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrContour::Init](#)” on page 3684.

Properties

Objects of this class have the following properties. See “[IDLgrContour Properties](#)” on page 3650 for details on individual properties.

- [ALL](#)
- [AM_PM](#)
- [C_COLOR](#)
- [C_LABEL_INTERVAL](#)
- [C_LABEL_OBJECTS](#)
- [C_LINestyle](#)
- [C_USE_LABEL_COLOR](#)
- [C_VALUE](#)
- [COLOR](#)
- [DAYS_OF_WEEK](#)
- [DEPTH_TEST_DISABLE](#)
- [ALPHA_CHANNEL](#)
- [ANISOTROPY](#)
- [C_FILL_PATTERN](#)
- [C_LABEL_NOGAPS](#)
- [C_LABEL_SHOW](#)
- [C_THICK](#)
- [C_USE_LABEL_ORIENTATION](#)
- [CLIP_PLANES](#)
- [DATA_VALUES](#)
- [DEPTH_OFFSET](#)
- [DEPTH_TEST_FUNCTION](#)

- DEPTH_WRITE_DISABLE
- DOUBLE_GEOM
- FILL
- GEOMX
- GEOMZ
- LABEL_FONT
- LABEL_FRMTDATA
- MAX_VALUE
- MONTHS
- PALETTE
- PLANAR
- REGISTER_PROPERTIES
- SHADING
- TICKLEN
- XCOORD_CONV
- YCOORD_CONV
- ZCOORD_CONV
- DOUBLE_DATA
- DOWNHILL
- GEOM
- GEOMY
- HIDE
- LABEL_FORMAT
- LABEL_UNITS
- MIN_VALUE
- N_LEVELS
- PARENT
- POLYGONS
- SHADE_RANGE
- TICKINTERVAL
- USE_TEXT_ALIGNMENTS
- XRANGE
- YRANGE
- ZRANGE

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- IDLgrContour::AdjustLabelOffsets
- IDLgrContour::Cleanup
- IDLgrContour::GetCTM
- IDLgrContour::GetLabelInfo
- IDLgrContour::GetProperty
- IDLgrContour::Init
- IDLgrContour::SetProperty

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrContour Properties

IDLgrContour objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrContour::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrContour::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrContour::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the retrievable properties associated with this object except DATA_VALUES and GEOM.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

Set this property to a floating-point value in the range [0.0, 1.0] to specify the opacity of the contour. The default value of 1.0 causes IDL to draw the contours completely opaque. If the value of this property is less than 1.0, then the pixels of the contours are blended with the pixels already on the screen, where the color of the contour is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha. Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results. This property has no effect on devices using indexed color mode. If contour colors (C_COLOR property) are supplied, then the value of this property is used for the contour alpha if the C_COLOR property does not contain alpha values (3-channel RGB colors). If the C_COLOR property does contain alpha values (4-channel RGBA colors), then the alpha from each contour color is used. When contour labels (either text or symbols) are used and a label's corresponding value in the C_USE_LABEL_COLOR property is zero, the value of this property is used for the label alpha if contour colors are not provided or if the contour colors are provided and do not contain alpha values. If the alpha values are supplied in the

C_COLOR property, then each label uses the alpha from its corresponding C_COLOR contour color. When contour labels (either text or symbols) are used and a label's corresponding value in the C_USE_LABEL_COLOR property is non-zero, then the value of the label object's ALPHA_CHANNEL property is used to draw the label. In other words, C_USE_LABEL_COLOR also implies using the label object's alpha as well as its color.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see [“Alpha Channel and Objects”](#) in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

AM_PM

A string vector of two values indicating the names of the AM and PM strings when processing explicitly formatted dates (CAPA, CApA, and CapA format codes) with the LABEL_FORMAT property.

Property Type	String vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ANISOTROPY

A three-element floating-point vector $[x, y, z]$ that represents the multipliers to be applied to the internally computed correction factors along each axis that account for anisotropic geometry. Correcting for anisotropy is particularly important for the appropriate representations of downhill tickmarks.

By default, IDL will automatically compute correction factors for anisotropy based on the [XYZ] range of the contour geometry. If the geometry (as provided via the GEOMX, GEOMY, and GEOMZ keywords) falls within the range $[x_{min}, y_{min}, z_{min}]$ to $[x_{max}, y_{max}, z_{max}]$, then the default correction factors are computed as follows:

$$\begin{aligned} dx &= x_{max} - x_{min} \\ dy &= y_{max} - y_{min} \\ dz &= z_{max} - z_{min} \end{aligned}$$

```

; Get the maximum of the ranges:
maxRange = (dx > dy) > dz
IF (dx EQ 0) THEN xcorrection = 1.0 ELSE $
    xcorrection = maxRange / dx
IF (dy EQ 0) THEN ycorrection = 1.0 ELSE $
    ycorrection = maxRange / dy
IF (dz EQ 0) THEN zcorrection = 1.0 ELSE $
    zcorrection = maxRange / dz

```

This internally computed correction is then multiplied by the corresponding $[x, y, z]$ values of the ANISOTROPY property. The default value for this property is $[1,1,1]$. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Anistrophy		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

C_COLOR

A vector of colors to be used to specify the color of each contour level. The vector may be of the form $[n]$ where each entry is a color index, or of the form $[3,n]$ where each 3-element row is an RGB color, or of the form $[4,n]$ where each 4-element row is an RGBA color. To remove contour colors after they have been set, set C_COLOR to a scalar.

If C_COLOR is not specified, the entire contour plot is drawn in the single color provided by the COLOR property, which is the default action. If there are no contour colors, GetProperty returns a value of -1 for this property.

If there are more contour levels than elements in C_COLOR, the elements of C_COLOR are cyclically repeated.

Property Type	USERDEF		
Name String	Colors		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

C_FILL_PATTERN

An array of IDLgrPattern objects that represent the patterns to be applied at each contour level if the FILL property is non-zero. If there are more contour levels than fill patterns, the patterns will be cyclically repeated. If this property is set to any

scalar value, all contour levels are filled with a solid color (this is the default). If there are no fill pattern objects stored in this property, GetProperty will return a value of -1.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Fill patterns		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

C_LABEL_INTERVAL

A floating-point vector that indicates the distance (measured parametrically relative to the length of each contour path) between labels for each contour level. If the number of contour levels exceeds the number of provided intervals, the C_LABEL_INTERVAL values will be repeated cyclically. The default is 0.4.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

C_LABEL_NOGAPS

An integer vector that indicates whether gaps should be computed for the labels at the corresponding contour value. A zero value indicates that gaps will be computed for labels at that contour value; a non-zero value indicates that no gaps will be computed for labels at that contour value. If the number of contour levels exceeds the number of elements in this vector, the C_LABEL_NOGAPS values will be repeated cyclically. By default, gaps for the labels are computed for all levels (so that a contour line does not pass through the label).

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

C_LABEL_OBJECTS

An array of object references that provides examples of labels to be drawn for each contour level. The objects specified via this property must inherit from one of the following classes:

- IDLgrSymbol
- IDLgrText

If a single object is provided, and it is an IDLgrText object, each of its strings will correspond to a contour level. If a vector of objects is used, any IDLgrText objects should have only a single string; each object will correspond to a contour level.

By default, with C_LABEL_OBJECTS set equal to a null object, IDL computes text labels that are the string representations of the corresponding contour level values. Note that the objects specified via this property are used as descriptors only. The actual objects drawn as labels are generated by IDL, and may be accessed via the IDLgrContour::GetLabelInfo method. The contour labels will have the same color as the corresponding contour level (see C_COLOR) unless the C_USE_LABEL_COLOR property is specified. The orientation of the label will be automatically computed unless the C_USE_LABEL_ORIENTATION property is specified. The horizontal and vertical alignment of any text labels will default to 0.5 (i.e., centered) unless the USE_TEXT_ALIGNMENTS property is specified.

Note

The object(s) set via this property will not be destroyed automatically when the contour is destroyed.

Property Type	Object reference array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

C_LABEL_SHOW

An integer vector that indicates whether labels are shown. For each contour value, if the corresponding value in the C_LABEL_SHOW vector is non-zero, the contour line for that contour value will be labeled. If the number of contour levels exceeds the number of elements in this vector, the C_LABEL_SHOW values will be repeated cyclically. The default is 0 indicating that no contour levels will be labeled.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

C_LINestyle

An integer array of linestyles that represent the linestyles to be applied at each contour level. The array may be either a vector of integers representing pre-defined linestyles, or an array of 2-element vectors representing a stippling pattern specification. If there are more contour levels than linestyles, the linestyles will be cyclically repeated. If this property is set to any scalar value, all levels are drawn as solid lines (this is the default). If there are no linestyles specified for this property, `GetProperty` will return a value of -1.

To use a pre-defined line style, set the `C_LINestyle` property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Linestyles		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

C_THICK

A floating-point array of line thicknesses that represent the thickness to be applied at each contour level, where each element is a value between 1.0 and 10.0. If there are more contour levels than line thicknesses, the thicknesses will be cyclically repeated. If this property is set to any scalar value, all contour levels are drawn with a line thickness of 1.0 points (this is the default). If there are no thicknesses specified for this property, `GetProperty` will return a value of -1.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Thicknesses		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

C_USE_LABEL_COLOR

An integer vector that indicates whether the COLOR property value for each of the label objects (for the corresponding contour level) is to be used to draw that label. If the number of contour levels exceeds the number of elements in this vector, the C_USE_LABEL_COLOR values will be repeated cyclically. By default, this value is zero, indicating that the COLOR properties of the label objects will be ignored, and the C_COLOR property for the contour object will be used instead.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Note

When user-defined IDLgrSymbol objects are used for labels in an IDLgrContour object and C_USE_LABEL_COLORS is set to non-zero values, all properties including the color are defined by the object composing the user-defined symbol, and not from the values set on the symbol object itself. For example, a green bow tie-shaped IDLgrPolyline is assigned to the symbol object, and this symbol object is assigned to the C_LABEL_OBJECTS property. If C_USE_LABEL_COLOR is set, there will be green bow tie shaped labels on the contour lines. If C_USE_LABEL_COLOR is not set, the bow tie color is defined by C_COLOR. Properties defined on the symbol object itself are ignored in all cases.

C_USE_LABEL_ORIENTATION

An integer vector that indicates whether the orientation for each of the label objects (for the corresponding contour level) is to be used when drawing the label. For text, the orientation of the object corresponds to the BASELINE and UPDIR property values; for a symbol, this refers to the default (un-rotated) orientation of the symbol. If the number of contour levels exceeds the number of elements in this vector, the C_USE_LABEL_ORIENTATION values will be repeated cyclically. By default, this

value is zero, indicating that orientation of the label object(s) will be set to automatically computed values (to correspond to the direction of the contour paths).

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

C_VALUE

A floating-point value or a floating-point vector for which contour values are to be drawn. Setting ONLY a scalar value of 0 will cause the default action of even sampling of contour levels across the DATA_VALUES, using the value of the N_LEVELS property to determine the number of samples. IDL will convert any other scalar value to a single element vector containing that value and will then use it as if the user had supplied it as a vector. If there are no values specified for this property, GetProperty will return a value of -1. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Level values		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by-*N* floating-point array of dimensions that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any x , y , or z coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the `MAX_NUM_CLIP_PLANES` keyword of the `GetDeviceInfo` method for the `IDLgrBuffer`, `IDLgrClipboard`, `IDLgrWindow`, and `IDLgrVRML` objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

The color to be used to draw the contours. The color may be specified as a color lookup table index or as a three-element vector [*red*, *green*, *blue*]. The default is [0,0,0]. This value will be ignored if the `C_COLOR` property is set to a vector.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DATA_VALUES

A floating-point vector or two-dimensional array that specifies the values to be contoured. The property is the same as the *Values* argument described in [“IDLgrContour::Init”](#) on page 3684. IDL converts and stores this data as double-precision floating-point if the argument is of type `DOUBLE` or if the `DOUBLE_DATA` property is set. Otherwise, the data is stored in single-precision. IDL returns the data as double-precision if it was stored in double-precision.

Property Type	Floating-point vector or array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DAYS_OF_WEEK

A string vector of 7 values that indicates the names to be used for the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdWA format codes) with the LABEL_FORMAT property.

Property Type	String vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEPTH_OFFSET

An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.

The units are “Z-Buffer units,” where a value of 1 is used to specify a distance that corresponds to a single step in the device’s Z-Buffer.

Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms. This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for “layering” complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value, such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because a set of DEPTH_OFFSET values may produce better results on one machine than on another. Using IDL’s software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.

Note

DEPTH_OFFSET has no effect unless the FILL property is set.

Property Type	INTEGER		
Name String	Depth offset		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.

- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DOUBLE_DATA

A Boolean value that indicates whether the object is to store data provided by either the *Values* argument or the DATA_VALUES property parameter in double-precision or single-precision floating-point.

- 0 = Single-precision floating-point
- 1 = Double-precision floating-point

IDL converts any value data already stored in the object to the requested precision, if necessary.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DOUBLE_GEOM

A Boolean value that indicates whether the object is to store data provided by any of the GEOMX, GEOMY, or GEOMZ property parameters in double-precision or single-precision floating-point.

- 0 = Single-precision floating-point
- 1 = Double-precision floating-point

IDL converts any geometry data already stored in the object to the requested precision, if necessary.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DOWNHILL

A Boolean value that indicates whether downhill tick marks should be rendered as part of each contour level to indicate the downhill direction relative to the contour line.

- 0 = Do not render downhill tick marks.
- 1 = Render downhill tick marks

In a property sheet, this property appears as an enumerated list with the following options:

- Hide
- Show

Property Type	ENUMLIST		
Name String	Downhill ticks		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

FILL

A Boolean value that indicates whether the contours should be filled. The default is to draw the contour levels as lines without filling. Filling contour may produce less than satisfactory results if your data contains NaNs, or if the contours are not closed.

Property Type	BOOLEAN		
Name String	Fill contours		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

GEOM

A floating-point array that contains the geometry associated with the contour. IDL returns this data in single-precision floating-point by default or in double-precision floating-point if the `DOUBLE_GEOM` property is set in the [IDLgrContour::Init](#) method.

The shape of the data contained in `GEOM` depends on the value of the `GEOMZ` property (and potentially the `GEOMX` and `GEOMY` properties). The following rules apply:

- If GEOMZ is a vector of Z values $[z_0, z_1, z_2, \dots]$ (in which case it is presumed that GEOMX and GEOMY were also set as vectors), GEOM will contain a $3 \times N$ array of XYZ values: $[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots]$.
- If GEOMZ is a $3 \times N$ array of XYZ values $[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots]$, GEOM will contain a $3 \times N$ array of XYZ values: $[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots]$.
- If GEOMZ is an $N \times M$ array of Z values $[[z_0, z_1, z_2, \dots], [z_N, z_{N+1}, z_{N+2}, \dots], \dots]$ (in which case it is presumed that GEOMX and GEOMY were also set as $N \times M$ arrays), GEOM will contain a $3 \times N \times M$ array of XYZ values: $[[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots], \dots, [[x_N, y_N, z_N], \dots], \dots]$
- If GEOMZ is a scalar and the PLANAR property contains a True value (in which case it is presumed that DATA_VALUES was set as an $N \times M$ array), GEOM will contain a $3 \times N \times M$ array of XYZ values: $[[[x_0, y_0, z_0], [x_1, y_1, z_1], \dots], \dots, [[x_N, y_N, z_N], \dots], \dots]$
- If GEOMZ is a scalar and the PLANAR property contains a False value, GEOM will be undefined.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

GEOMX

A floating-point vector or two-dimensional array that specifies the X coordinates of the geometry with which the contour values correspond. If X is a vector, it must match the number of elements in the *Values* argument or DATA_VALUES property value, or it must match the first of the two dimensions of the *Values* argument or DATA_VALUES property value (in which case, the X coordinates will be repeated for each row of data values). IDL converts and maintains this data as double-precision floating-point if the parameter is of type DOUBLE or if the DOUBLE_GEOM property is non-zero. Otherwise, the data is stored in single-precision. IDL returns the data as double-precision if it was stored in double-precision.

Property Type	Floating-point vector or array		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

GEOMY

A floating-point vector or two-dimensional array specifying the Y coordinates of the geometry with which the contour values correspond. If Y is a vector, it must match the number of elements in the *Values* argument or DATA_VALUES property value, or it must match the second of the two dimensions of the *Values* argument or DATA_VALUES property value (in which case, the Y coordinates will be repeated for each column of data values). IDL converts and maintains this data as double precision floating point if the parameter is of type DOUBLE or if the DOUBLE_GEOM property is non-zero. Otherwise, the data is stored in single precision. IDL returns the data as double precision if it was stored in double precision.

Property Type	Floating-point vector or array		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

GEOMZ

A floating-point scalar, a vector, or a two-dimensional array specifying the Z coordinates of the geometry with which the contour values correspond.

- If GEOMZ is a scalar and the PLANAR property contains a True value, the resulting contour geometry will be projected onto the plane Z=GEOMZ. If GEOMZ is a scalar, and the PLANAR property is not set, any geometry associated with the contour will be freed.
- If GEOMZ is a vector of Z coordinates, the number of elements in the vector should be the same as the number of elements in DATA_VALUES.
- If GEOMZ is an array and the leading dimension is 3, then the data is treated as a list or array of 3-D vertices. The product of the array dimensions following the 3 should equal the number of DATA_VALUES.
- If GEOMZ is an array and the leading dimension is not 3, then the data is treated as a list or array of single Z coordinates. If the number of dimensions is 2, then the data is treated as a 2-D array of Z values. Otherwise, it is treated as a vector of Z values. In either case, the total number of elements in the array should be the same as the number of DATA_VALUES.
- If GEOMZ is not set, the geometry will be derived from the DATA_VALUES property (if it is set to a two-dimensional array). In this case, the connectivity is

implied. The X and Y coordinates match the row and column indices of the array, and the Z coordinates match the data values.

IDL converts and maintains this data as double precision floating point if the parameter is of type DOUBLE or if the DOUBLE_GEOM property is non-zero. Otherwise, the data is stored in single precision. IDL returns the data as double precision if it was stored in double precision.

Property Type	Floating-point scalar, vector, or array		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value that indicates whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LABEL_FONT

An object reference to an IDLgrFont object that describes the default font to be used for contour labels. This font will be used for all text labels automatically generated by IDL (i.e., if C_LABEL_SHOW is set but the corresponding C_LABEL_OBJECTS text object is not provided), or for any text label objects provided via C_LABEL_OBJECTS that do not already have the font property set. The default

value for this property is a NULL object reference, indicating that 12 pt. Helvetica will be used.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LABEL_FORMAT

A string that represents a format string or the name of a function to be used to format the contour labels. If the string begins with an open parenthesis, it is treated as a standard format string. (Refer to the Format Codes in the IDL Reference Guide.) If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate contour level labels.

The callback function is called with three parameters: Axis, Index, and Value, where:

- Axis is simply the value 2 to indicate that values along the Z axis are being formatted. (This allows a single callback routine to be used for both axis labeling and contour labeling.)
- Index is the contour level index (indices start at 0).
- Value is the data value of the current contour level.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LABEL_FRMTDATA

A user-defined value of any type passed via the DATA keyword to the user-supplied formatting function specified via the LABEL_FORMAT property, if any. By default, this value is 0, indicating that the DATA keyword will not be set (and furthermore, need not be supported by the user-supplied function).

Note

LABEL_FRMTDATA will not be included in the structure returned via the ALL property to the IDLgrContour::GetProperty method.

Property Type	Scalar of any type		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

LABEL_UNITS

A string that indicates the units to be used for default contour level labeling.

Valid unit strings include:

- "Numeric"
- "Years"
- "Months"
- "Days"
- "Hours"
- "Minutes"
- "Seconds"
- "Time" - Use this value to indicate that the contour levels correspond to time values; IDL will determine the appropriate label format based upon the range of values covered by the contour Z data.
- "" - The empty string is equivalent to the "Numeric" unit. This is the default.

If any of the time units are utilized, then the contour values are interpreted as Julian date/time values.

Note

The singular form of each of the time unit strings is also acceptable (for example, LEVEL_UNITS='Day' is equivalent to LEVEL_UNITS='Days').

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MAX_VALUE

A floating point value that indicates the maximum value to be plotted. Data values greater than this value are treated as missing data. The default is the maximum value of the input Z data. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Maximum value		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MONTHS

A string vector of 12 values indicating the names to be used for the months when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the C_LABEL_FORMAT keyword.

Property Type	String vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MIN_VALUE

A floating-point value that indicates the minimum value to be plotted. Data values less than this value are treated as missing data. The default is the minimum value of the input Z data. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Minimum value		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

N_LEVELS

An integer value that indicates the number of contour levels to generate. This property is ignored if the C_VALUE property is set to a vector, in which case, the number of levels is derived from the number of elements in that vector. Set this

property to zero to indicate that IDL should compute a default number of levels based on the range of data values. This is the default.

Property Type	INTEGER		
Name String	Number of levels		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class) that defines the color palette of this object. This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the PALETTE property on this object is not set, the destination object PALETTE property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: No	Set: No	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this contour.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

PLANAR

A Boolean value that indicates whether the contoured data is to be projected onto a plane. This property is ignored if GEOMZ is not a scalar. The default is non-planar (i.e., to display the contoured data at the Z locations provided by the GEOMZ property).

In a property sheet, this property appears as an enumerated list with the following options:

- Planar (the default)
- Three-D

Property Type	ENUMLIST		
Name String	Projection		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

POLYGONS

An integer array of polygon descriptions that represents the connectivity information for the data to be contoured (as specified in the *Values* argument or the DATA_VALUES property). A polygon description is an integer or long word array of the form: $[n, i0, i1, \dots, in-1]$, where n is the number of vertices that define the polygon, and $i0..in-1$ are indices into the *X*, *Y*, and *Z* arguments that represent the polygon vertices. To ignore an entry in the POLYGONS array, set the vertex count, n , to 0. To end the drawing list, even if additional array space is available, set n to -1. If this property is not specified, a single polygon will be generated.

Note

The connectivity array described by POLYGONS allows an individual object to contain more than one polygon. Vertex, normal, and color information can be shared by the multiple polygons. Consequently, the polygon object can represent an entire mesh and compute reasonable normal estimates in most cases.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the

object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

SHADE_RANGE

A two-element integer array that specifies the range of pixel values (color indices) to use for shading. The first element is the color index for the darkest pixel. The second element is the color index for the brightest pixel. This value is ignored when the contour is drawn to a graphics destination that uses the RGB color model.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SHADING

An integer that indicates the type of shading to use:

- 0 = Flat (default): The color has a constant intensity for each face of the contour, based on the normal vector.
- 1 = Gouraud: The colors are interpolated between vertices, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

In a property sheet, this property appears as an enumerated list with the following options:

- Flat (the default)
- Gouraud

Property Type	ENUMLIST		
Name String	Shading		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKINTERVAL

A floating-point value that indicates the distance between downhill tickmarks, in data units. If TICKINTERVAL is not set, or if you explicitly set it to zero, IDL will compute the distance based on the geometry of the contour. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Downhill tick interval		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TICKLEN

A floating-point value that indicates the length of the downhill tickmarks, in data units. If TICKLEN is not set, or if you explicitly set it to zero, IDL will compute the length based on the geometry of the contour. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Downhill tick length		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

USE_TEXT_ALIGNMENTS

A Boolean value that indicates whether, for any IDLgrText labels (as specified via the C_LABEL_OBJECTS property), the ALIGNMENT and VERTICAL_ALIGNMENT property values for the given IDLgrText object(s) are to be used to draw the corresponding labels. By default, this value is zero, indicating that the ALIGNMENT and VERTICAL_ALIGNMENT properties of the label

IDLgrText object(s) will be set to default values (0.5 for each, indicating centered labels).

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element double-precision floating-point vector of the form $[xmin, xmax]$ specifying the range of the X data coordinates covered by the contour.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element double-precision floating-point vector of the form [*ymin*, *ymax*] that specifies the range of the *Y* data coordinates covered by the contour.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors that converts *Z* coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ that specifies the range of the Z data coordinates covered by the contour.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

ALPHA_CHANNEL added: 6.1

IDLgrContour::AdjustLabelOffsets

The IDLgrContour::AdjustLabelOffsets procedure method adjusts the offsets at which contour labels are positioned.

Syntax

Obj->[IDLgrContour::]AdjustLabelOffsets, *LevelIndex*, *LabelOffsets*

Arguments

LevelIndex

The index of the contour level for which the label offsets are being adjusted. This value must be greater than or equal to zero and less than the number of levels (refer to the N_LEVELS property in the IDLgrContour::Init method).

LabelOffsets

A scalar or vector of floating point offsets, [t0, t1, ...], that indicate the parametric offsets along the length of each contour line at which each label is to be positioned. The number of elements in this vector must exactly match the number of elements returned in the LABEL_OFFSETS vector retrieved via the IDLgrContour::GetLabelInfo method for the same level.

Keywords

None

Version History

Introduced: 5.1

IDLgrContour::Cleanup

The IDLgrContour::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrContour::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrContour::GetCTM

The IDLgrContour::GetCTM method returns the 4 x 4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrContour::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref] )
```

Return Value

Returns the graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the surface object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrContour::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.1

IDLgrContour::GetLabelInfo

The IDLgrContour::GetLabelInfo procedure method retrieves information about the labels for a contour. The returned information is only valid until the next time the C_LABEL_INTERVAL or C_LABEL_OBJECTS property is modified using the IDLgrContour::SetProperty method, or the offsets are adjusted using the IDLgrContour::AdjustLabelOffsets method.

Syntax

```
Obj->[IDLgrContour::]GetLabelInfo, Destination, LevelIndex  
    [, LABEL_OFFSETS=variable] [, LABEL_POLYLINES=variable]  
    [, LABEL_OBJECTS=variable]
```

Arguments

Destination

A reference to a destination object (such as an IDLgrWindow or IDLgrBuffer object). The contour label information will be computed so that the requested font size is satisfied for this destination device.

LevelIndex

The index of the contour level for which the label information is being requested. This value must be greater than or equal to zero and less than the number of levels (refer to the N_LEVELS keyword in the IDLgrContour::Init method).

Keywords

LABEL_OFFSETS

Set this keyword to a named variable that upon return will contain a vector of floating point offsets, [t0, t1, ...], that indicate the parametric offsets along the length of each contour line at which the contour labels are positioned.

LABEL_POLYLINES

Set this keyword to a named variable that upon return will contain a vector of contour polyline indices, [P₀, P₁, ...], that indicate which contour lines are labeled. P_i corresponds to the ith contour line.

Note that if a given contour line has more than one label along its perimeter, then the corresponding polyline index may appear more than once in the LABEL_POLYLINES vector.

LABEL_OBJECTS

Set this keyword to a named variable that upon return will contain a vector of objects that represent the labels for each contour label.

Version History

Introduced: 5.1

IDLgrContour::GetProperty

The IDLgrContour::GetProperty procedure method retrieves the value of a property or group of properties for the contour.

Syntax

Obj->[IDLgrContour::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrContour Properties](#)” on page 3650 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrContour::Init

The IDLgrContour::Init function method initializes the contour object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrContour' [, Values] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrContour::]Init([Values] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Values

A vector or two-dimensional array of values to be contoured. If no values are provided, the values will be derived from the GEOMZ keyword value (if set and the PLANAR keyword is not set). In this case, the values to be contoured will match the Z coordinates of the provided geometry. IDL converts and maintains this data as double-precision floating-point if the argument is of type DOUBLE or if the DOUBLE_DATA keyword is set. Otherwise, the data is stored in single-precision. IDL returns the data as double-precision if it was stored in double-precision.

Keywords

Any property listed under “[IDLgrContour Properties](#)” on page 3650 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

AM_PM, C_LABEL_INTERVAL, CLIP_PLANES, C_LABEL_OBJECTS, C_LABEL_NOGAPS, C_LABEL_SHOW, C_USE_LABEL_COLOR, C_USE_LABEL_ORIENTATION, DAYS_OF_WEEK, LABEL_FONT, LABEL_FORMAT, LABEL_FRMATTDATA, LABEL_UNITS, MONTHS, USE_TEXT_ALIGNMENTS keywords added: 5.6

IDLgrContour:: SetProperty

The IDLgrContour::SetProperty procedure method sets the value of a property or group of properties for the contour.

Syntax

Obj->[IDLgrContour::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrContour Properties](#)” on page 3650 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrFont

A font object represents a typeface, style, weight, and point size that may be associated with text objects.

Superclasses

None.

Creation

See “[IDLgrFont::Init](#)” on page 3692.

Properties

Objects of this class have the following properties. See “[IDLgrFont Properties](#)” on page 3688 for details on individual properties.

- [ALL](#)
- [SIZE](#)
- [SUBSTITUTE](#)
- [THICK](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrFont::Cleanup](#)
- [IDLgrFont::GetProperty](#)
- [IDLgrFont::Init](#)
- [IDLgrFont::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrFont Properties

IDLgrFont objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrFont::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrFont::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrFont::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

SIZE

A floating-point value representing the point size of the font. The default is 12.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SUBSTITUTE

A string that indicates the font to use as a substitute if the specified *Fontname* is not available on the graphics destination. Valid values are only those fonts that are available on all destination objects (the fonts included with IDL). These are: 'Helvetica' (the default), 'Courier', 'Times', 'Symbol', or 'Hershey'.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0, indicating the line thickness (measured in points) to use for the Hershey vector fonts. The default is 1.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLgrFont::Cleanup

The IDLgrFont::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrFont::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrFont::GetProperty

The IDLgrFont::GetProperty procedure method retrieves the value of a property or group of properties for the font.

Syntax

Obj->[IDLgrFont:]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrFont Properties](#)” on page 3688 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrFont::Init

The IDLgrFont::Init function method initializes the font object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrFont' [, Fontname] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrFont::]Init([Fontname] [, PROPERTY=value])
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Fontname

A string representing the name of the font to be used. This string should take the form 'fontname*modifier1*modifier2*...*modifierN'. All destination objects support the following fontnames: Helvetica, Courier, Times, Symbol, and Monospace Symbol. (These fonts are included with IDL; you may have other fonts installed on your system as well.) Valid modifiers for each of these fonts (except Symbol and Monospace Symbol) are:

- *Font weight*: Bold
- *Font angle*: Italic

For example, 'Helvetica*Bold*Italic'.

To select a Hershey font, use a fontname of the form: 'Hershey*fontnum'. See [Appendix H, “Fonts”](#) for further information and a list of fonts supported by IDL.

Note

Beginning with IDL version 5.1, only TrueType and Hershey fonts are supported in the Object Graphics system.

Keywords

Any property listed under “[IDLgrFont Properties](#)” on page 3688 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrFont:: SetProperty

The IDLgrFont::SetProperty procedure method sets the value of a property or group of properties for the font.

Syntax

Obj->[IDLgrFont:]SetProperty[, *PROPERTY*=*value*]

Arguments

None

Keywords

Any property listed under “[IDLgrFont Properties](#)” on page 3688 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrImage

An image object represents a mapping from a two-dimensional array of data values to a two dimensional array of pixel colors, resulting in a flat 2-D-scaled version of the image, drawn at $Z = 0$.

An IDLgrImage object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Transformations of IDLgrImage Objects

IDLgrImage objects are drawn at $Z=0$ and are positioned and sized with respect to two points:

```
p1 = [LOCATION(0), LOCATION(1), 0]
p2 = [LOCATION(0) + DIMENSION(0), LOCATION(1) + DIMENSION(1), 0].
```

where LOCATION and DIMENSION are properties of the image object. These points are transformed in *three* dimensions, resulting in screen space points designated as $p1'$ and $p2'$. The image data is drawn on the display as a 2-D image within the 2-D rectangle defined by $(p1'[0], p1'[1] - p2'[0], p2'[1])$. The 2-D image data is *scaled* in two dimensions (not rotated) to fit into this projected rectangle and then drawn with Z buffering disabled.

Note

Image objects do not take into account the Z locations of other objects that may be included in the view object. This means that objects that are drawn to the destination object (window or printer) *after the image is drawn* will appear to be in front of the image, even if they are located at a negative Z value (behind the image object). Objects are drawn to a destination device in the order that they are added (via the Add method) to the model, view, or scene that contains them. To rotate or position image objects in three-dimensional space, use the [IDLgrPolygon](#) object with texture mapping enabled.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrImage::Init](#)” on page 3716.

Properties

Objects of this class have the following properties. See “[IDLgrImage Properties](#)” on page 3698 for details on individual properties.

- [ALL](#)
- [CHANNEL](#)
- [DATA](#)
- [DEPTH_TEST_FUNCTION](#)
- [DIMENSIONS](#)
- [HIDE](#)
- [INTERPOLATE](#)
- [NO_COPY](#)
- [PALETTE](#)
- [REGISTER_PROPERTIES](#)
- [SHARE_DATA](#)
- [XCOORD_CONV](#)
- [XRANGE](#)
- [YRANGE](#)
- [ZRANGE](#)
- [BLEND_FUNCTION](#)
- [CLIP_PLANES](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_WRITE_DISABLE](#)
- [GREYSCALE](#)
- [INTERLEAVE](#)
- [LOCATION](#)
- [ORDER](#)
- [PARENT](#)
- [RESET_DATA](#)
- [SUB_RECT](#)
- [XCOORD_CONV](#)
- [YCOORD_CONV](#)
- [ZCOORD_CONV](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrImage::Cleanup](#)
- [IDLgrImage::GetCTM](#)
- [IDLgrImage::GetProperty](#)
- [IDLgrImage::Init](#)
- [IDLgrImage::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrImage Properties

IDLgrImage objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrImage::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrImage::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrImage::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

BLEND_FUNCTION

A two-element integer vector that controls how the alpha channel values will be interpreted. Set this property equal to a two-element vector [*src*, *dst*] specifying one of the functions listed below for each of the source and destination objects. These are only valid for RGB model destinations. If an alpha channel is not specified in an image, the image’s alpha blend factor is assumed to be 1.0. The values of the blending function (V_{src} and V_{dst}) are used in the following equation

$$C_d' = (V_{src} \cdot C_i) + (V_{dst} \cdot C_d)$$

where C_d is the initial color of a pixel on the destination device (the background color), C_i is the color of the pixel in the image, and C_d' is the resulting color of the pixel.

Setting *src* and *dst* in the BLEND_FUNCTION vector to the following values determine how each term in the equation is calculated:

src or dst	V_{src} or V_{dst}	What the function does
0	n/a	Alpha blending is disabled, which is the default setting. $C_d' = C_i$
1	0	The value of V_{src} or V_{dst} in the equation is zero, thus the value of the term is zero.
2	1	The value of V_{src} or V_{dst} in the equation is one, thus the value of the term is the same as the color value.
3	$Image_a$	The value of V_{src} or V_{dst} in the equation is the blend factor of the image's alpha channel.
4	$1-Image_a$	The value of V_{src} or V_{dst} in the equation is one minus the blend factor of the image's alpha channel.

Table 8-5: Values for *src* and *dst* in BLEND_FUNCTION

For example, setting BLEND_FUNCTION = [3, 4] creates an image in which you can see through the foreground image to the background to the extent defined by the alpha channel values of the foreground image.

Since the alpha blending operation is dependent on the values of pixels already drawn to the destination for some blending functions, the final result may depend more on the order of drawing the images, and not necessarily on their relative location along the Z axis. IDL draws images in the order that they are stored in the IDLgrModel object that contains them.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Blend function		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CHANNEL

A hexadecimal bitmask that defines which color channel(s) to draw. Each bit that is a 1 is drawn; each bit that is a 0 is not drawn. For example, 'ff0000'X represents a Blue channel write. The default is to draw all channels, and is represented by the hexadecimal value 'ffffff'X.

Note

This property is ignored for CI destination objects.

This property is registered, but it is hidden by default.

Property Type	INTEGER		
Name String	Color channel		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by-*N* floating-point array of dimensions [4,*N*] specifying the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any *x*, *y*, or *z* coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DATA

A $n \times m$, $2 \times n \times m$, $3 \times n \times m$, or $4 \times n \times m$ array (any type) of image data for the object. The n and m values may be in any position as specified by the INTERLEAVE property. This property is equivalent to the optional argument, *ImageData*.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DIMENSIONS

A two-element integer vector of the form [*width, height*] specifying the dimensions of the rectangle in which the image is to be drawn on the device. The image will be resampled as necessary to fit within this rectangle. The default is derived from the dimensions of the given image data and is measured in pixels. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Draw dimensions		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

GREYSCALE

A Boolean value that determines whether the image is drawn through a palette.

0 = Use palette (the default).

1 = Do not use a palette.

If this property is not set, for an RGB color model destination, if a palette is present in the image object, it is used. If there is no current destination palette, a greyscale palette is used. For a Color Index color model destination, the current destination palette is used.

Note

Only single band images (i.e. $1 \times n \times m$) are affected by this property. By default, GREYSCALE is disabled.

Property Type	BOOLEAN		
Name String	Grayscale		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value that indicates whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

INTERLEAVE

An integer value that indicates the dimension over which color is interleaved for images with more than one channel:

- 0 = Pixel interleaved: Images with dimensions $(3, m, n)$
- 1 = Scan line interleaved (row interleaved): Images with dimensions $(m, 3, n)$
- 2 = Planar interleaved: Images with dimensions $(m, n, 3)$.

Note

If an alpha channel is present, the 3s should be replaced by 4s. In a greyscale image with an alpha channel, the 3s should be replaced by 2s.

In a property sheet, this property appears as an enumerated list with the following options:

- Pixel
- Scanline
- Planar.

This property is registered, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Interleaving		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

INTERPOLATE

An integer value to determine whether to display the IDLgrImage object using bilinear interpolation. The default is to use nearest neighbor interpolation.

- 0 = Nearest neighbor interpolation (the default)
- 1 = Bilinear interpolation

In a property sheet, this property appears as an enumerated list with the following options:

- Nearest neighbor
- Bilinear

Property Type	ENUMLIST		
Name String	Interpolation		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LOCATION

A two- or three-element floating-point vector $[x, y]$ or $[x, y, z]$ specifying the position of the lower lefthand corner of the image, measured in data units. If the vector is of

the form $[x, y]$, then the z value is set equal to zero. The default is $[0, 0, 0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Location		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

NO_COPY

A Boolean value that determines whether to relocate the image data from the input variable to the image object, leaving the input variable *ImageData* undefined. Only the *ImageData* argument is affected. If this property is omitted, the input image data will be duplicated and a copy will be stored in the object.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ORDER

A Boolean value that determines whether to force the rows of the image data to be drawn from top to bottom. By default, image data is drawn from the bottom row up to the top row.

In a property sheet, this property appears as an enumerated list with the following options:

- Bottom-to-top
- Top-to-bottom

Property Type	ENUMLIST		
Name String	Row order		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the [IDLgrPalette](#) object class) that specifies the red, green, and blue values of the color lookup table to be associated with the image if it is an indexed color image. This property is ignored if the image is a greyscale or RGB image.

Note

This table is only used when the destination is an RGB model device. The Indexed color model writes the indices directly to the device. In order to ensure that these colors are used when the image is displayed, this palette must be copied to the graphics destination's palette for any graphics destination that uses the Indexed color model.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESET_DATA

A Boolean value that determines whether to treat the data provided via the DATA property as a new data set unique to this object, rather than overwriting data that is shared by other objects.

- 0 = Overwrite data that is shared by other objects (the default).
- 1 = Treat DATA as a new data set unique to this object.

There is no reason to use this property if the object on which the property is being set does not currently share data with another object (that is, if the SHARE_DATA property is not in use). This property has no effect if no new data is provided via the DATA property.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHARE_DATA

An object reference to an object with which data is to be shared by this image. An image may only share data with another image. The SHARE_DATA property is intended for use when data values are not set via an argument to the object's Init method or by setting the object's DATA property.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SUB_RECT

A four-element floating-point vector, $[x, y, xdim, ydim]$, specifying the position of the lower left-hand corner and the dimensions of the sub-rectangle to display.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Sub-rectangle		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element double-precision floating-point vector of the form $[xmin, xmax]$ that specifies the range of x data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element double-precision floating-point vector of the form [*ymin*, *ymax*] that specifies the range of y data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

Two-element double-precision floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLgrImage::Cleanup

The IDLgrImage::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrImage::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrImage::GetCTM

The IDLgrImage::GetCTM function method returns the 4 x 4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrImage::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the image object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrImage::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrImage::GetProperty

The IDLgrImage::GetProperty procedure method retrieves the value of the property or group of properties for the image.

Syntax

```
Obj->[IDLgrImage::]GetProperty[, PROPERTY=variable]
```

Arguments

None

Keywords

Any property listed under “[IDLgrImage Properties](#)” on page 3698 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrImage::Init

The IDLgrImage::Init function method initializes the image object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrImage' [, ImageData] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrImage::]Init([ImageData] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

ImageData

An array of data values to be displayed as an image. If this argument is not already of byte type, it is converted to byte type when the image object is created. Since IDL maintains the image data using the byte type, the input data values should range from 0 through 255. Image objects can have a single channel (one value per pixel—greyscale or color indexed), two channels (greyscale and alpha), three channels (red, green, and blue), or four channels (red, green, blue, and alpha). The alpha channel, if present, determines the transparency of the pixel. The BLEND_FUNCTION property controls the interpretation of the alpha channel values. With channels, the data value of 0 specifies minimum intensity and the data value of 255 specifies maximum intensity. The alpha channel values are also specified in the image data in the range 0

through 255, with an image data value of 0 corresponding to an alpha blend factor of 0 and an image data value of 255 corresponding to an alpha blend factor of 1.0.

ImageData can be any of the following, where n is the width of the image, and m is the height:

- An $n \times m$ array of color lookup table indices.
- An $n \times m$ greyscale image, or a $2 \times n \times m$, $n \times 2 \times m$, or $n \times m \times 2$ greyscale image with an alpha channel. (The alpha channel is ignored if the destination device uses indexed color mode.)
- A $3 \times n \times m$, $n \times 3 \times m$, or $n \times m \times 3$ RGB image, or a $4 \times n \times m$, $n \times 4 \times m$, or $n \times m \times 4$ RGB image with an alpha channel.

If the array has more than one channel, the interleave is specified by the `INTERLEAVE` property.

Keywords

Any property listed under “[IDLgrImage Properties](#)” on page 3698 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

`CLIP_PLANES` keyword added: 5.6

IDLgrImage:: SetProperty

The IDLgrImage::SetProperty procedure method sets the value of the property or group of properties for the image.

Syntax

Obj->[IDLgrImage::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrImage Properties](#)” on page 3698 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrLegend

The IDLgrLegend object provides a simple interface for displaying a legend. The legend itself consists of a (filled and/or framed) box around one or more legend items (arranged in a single column) and an optional title string. Each legend item consists of a glyph patch positioned to the left of a text string. The glyph patch is drawn in a square which is a fraction of the legend label font height. The glyph itself can be in one of two types (see the TYPE keyword). In line type, the glyph is a line segment with line style, thickness and color attributes and an optional symbol object drawn over it. In fill type, the glyph is a square patch drawn with color and optional pattern object attributes.

An IDLgrLegend object is a *composite object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

This object class is implemented in the IDL language. Its source code can be found in the file `idlgrlegend_undefine.pro` in the `lib` subdirectory of the IDL distribution.

Superclasses

[IDLgrModel](#)

[IDLitComponent](#)

Creation

See “[IDLgrLegend::Init](#)” on page 3737.

Properties

Objects of this class have the following properties. See “[IDLgrLegend Properties](#)” on page 3722 for details on individual properties.

- [ALL](#)
- [BORDER_GAP](#)
- [COLUMNS](#)
- [FILL_COLOR](#)
- [FONT](#)

- GAP
- GLYPH_WIDTH
- HIDE
- ITEM_COLOR
- ITEM_LINestyle
- ITEM_NAME
- ITEM_OBJECT
- ITEM_THICK
- ITEM_TYPE
- OUTLINE_COLOR
- OUTLINE_THICK
- PARENT
- RECOMPUTE
- SHOW_FILL
- SHOW_OUTLINE
- TEXT_COLOR
- TITLE
- XCOORD_CONV
- XRANGE
- YCOORD_CONV
- YRANGE
- ZCOORD_CONV
- ZRANGE

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrLegend::Cleanup`
- `IDLgrLegend::ComputeDimensions`

- [IDLgrLegend::GetProperty](#)
- [IDLgrLegend::Init](#)
- [IDLgrLegend::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrLegend Properties

IDLgrLegend objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrLegend::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrLegend::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrLegend::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the retrievable properties associated with this object.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

BORDER_GAP

A floating-point value to indicate the amount of blank space to be placed around the outside of the glyphs and text items. The units for this property are fractions of the legend label font height. The default is 0.1 (10% of the label font height).

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLUMNS

An integer value to indicate the number of columns the legend items should be displayed in. The default is one column.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FILL_COLOR

The color to be used to fill the legend background box. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default is [255,255,255].

Property Type	COLOR		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FONT

An object reference to an instance of an IDLgrFont object class that describes the font to use to draw the legend labels. The default is 12 point Helvetica.

Note

If the default font is in use, retrieving the value of the FONT property (using the GetProperty method) will return a font object that will be destroyed when this legend object is destroyed, leaving a dangling object reference.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GAP

A floating-point value that indicates the amount of blank space to be placed vertically between each legend item. The units for this property are fractions of the legend label

font height. The default is 0.1 (10% of the label font height). The same gap is placed horizontally between the legend glyph and the legend text string.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GLYPH_WIDTH

A floating-point value to indicate the width of the glyphs, measured as a fraction of the font height. The default is 0.8 (80% of the font height).

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value that indicates whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ITEM_COLOR

An array of colors defining the color of each item. The array defines M different colors, and should be either of the form $[3, M]$ or $[M]$. In the first case, the three values

are used as an RGB triplet, in the second case, the single value is used as a color index value. The default color is [0,0,0].

Property Type	Color array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ITEM_LINestyle

An integer array of integers defining the style of the line to be drawn if the TYPE property is set to zero. The array can be of the form [M] or [2,M]. The first form selects the linestyle for each legend item from the predefined defaults:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot
- 5 = long dash
- 6 = no line drawn

The second form specifies the stippling pattern explicitly for each legend item (see the LINestyle property to [IDLgrPolyline::Init](#) for details).

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ITEM_NAME

An string array representing the names of items in the legend. Specifying this property is the same as providing the *aName* argument for the `IDLgrLegend::Init` method.

Property Type	String array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ITEM_OBJECT

An array of object references of type `IDLgrSymbol` or `IDLgrPattern` that represents the shapes of the items in the legend. A symbol object is drawn only if the `TYPE` property is set to zero. A pattern object is used when drawing the color patch if the `TYPE` property is set to one. The default object is the `NULL` object.

Note

If one or more `IDLgrSymbol` object references are provided, the `SIZE` property of those objects may be modified by this legend to suit its layout needs.

Property Type	Object reference array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ITEM_THICK

A floating-point array of floating-point values that define the thickness of each item line, in points, where each element is a value between 1.0 and 10.0. This property is only used if the `TYPE` property is set to zero. The default is 1.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ITEM_TYPE

An integer array that defines the type of glyph to be displayed for each item:

- 0 = line type (the default)
- 1 = filled box type

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

OUTLINE_COLOR

The color to be used to draw the legend outline box. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default is [0,0,0].

Property Type	Color		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

OUTLINE_THICK

A floating-point value between 1.0 and 10.0 that defines the thickness of the outline frame, in points. The default is 1.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PARENT

An object reference to the object that contains this legend.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

RECOMPUTE

A Boolean value that determines whether to recompute the legend dimensions when the legend is redrawn.

- 0 = Prevent IDL from recomputing legend dimensions (the default).
- 1 = Recompute the legend dimensions when the legend is redrawn.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: No	Registered: No

SHOW_FILL

A Boolean value that indicates whether the background should be filled with a color:

- 0 = Do not fill background (the default)
- 1 = Fill background

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SHOW_OUTLINE

A Boolean value indicating whether the outline box should be displayed:

- 0 = Do not display outline (the default)

- 1 = Display outline

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TEXT_COLOR

The color to be used to draw the legend item text. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default is [0,0,0].

Property Type	Color		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TITLE

An object reference to an instance of the IDLgrText object class to specify the title for the legend. The default is the null object, specifying that no title is drawn. The title will be centered at the top of the legend, even if the text object itself has an associated location.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XCOORD_CONV

A floating-point vector, [s_0, s_1], of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element double-precision floating-point vector of the form [*xmin*, *xmax*] that specifies the range of the *X* data coordinates covered by the legend.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors that convert *Y* coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element double-precision floating-point vector of the form $[ymin, ymax]$ that specifies the range of the Y data coordinates covered by the legend.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors that convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is $[0.0, 1.0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ that specifies the range of the Z data coordinates covered by the legend.

Note

Until the legend is drawn to the destination object, the [XYZ]RANGE properties will be zero. Use the `ComputeDimensions` method on the legend object to get the data dimensions of the legend prior to a draw operation.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLgrLegend::Cleanup

The IDLgrLegend::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrLegend::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrLegend::ComputeDimensions

The IDLgrLegend::ComputeDimensions function method retrieves the dimensions of a legend object for the given destination object. The result is a three-element double-precision floating-point vector [*xDim*, *yDim*, *zDim*] representing the dimensions of the legend object measured in data units.

Syntax

```
Result = Obj->[IDLgrLegend::]ComputeDimensions( DestinationObject  
[, PATH=objref(s)] )
```

Return Value

Returns the dimensions of a legend object for the given destination object.

Arguments

DestinationObject

The object reference to a destination object (IDLgrBuffer, IDLgrClipboard, IDLgrPrinter, or IDLgrWindow) for which the dimensions of the legend are being requested.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrLegend::ComputeDimensions is called from within a Draw method and the PATH keyword is not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in IDLgrModel::Add.

Version History

Introduced: 5.1

IDLgrLegend::GetProperty

The IDLgrLegend::GetProperty procedure method retrieves the value of a property or group of properties for the legend.

Syntax

Obj->[IDLgrLegend::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrLegend Properties](#)” on page 3722 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrLegend::Init

The IDLgrLegend::Init function method initializes the legend object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrLegend' [, aItemNames] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrLegend::]Init([aItemNames] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

altemNames

An array of strings to be used as the displayed item label. The length of this array is used to determine the number of items to be displayed. Each item is defined by taking one element from the ITEM_NAME, ITEM_TYPE, ITEM_LINestyle, ITEM_THICK, ITEM_COLOR, and ITEM_OBJECT vectors. If the number of items (as defined by the ITEM_NAME array) exceeds any of the attribute vectors, the attribute defaults will be used for any additional items.

Keywords

Any property listed under [“IDLgrLegend Properties”](#) on page 3722 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

IDLgrLegend:: SetProperty

The IDLgrLegend::SetProperty procedure method sets the value of a property or group of properties for the legend.

Syntax

Obj->[IDLgrLegend::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrLegend Properties](#)” on page 3722 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrLight

A light object represents a source of illumination for three-dimensional graphic objects. Lights may be either ambient, positional, directional, or spotlights. A maximum of 8 lights per view are allowed. Lights are not required for objects displayed in two dimensions.

An IDLgrLight object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See [“IDLgrLight::Init”](#) on page 3753.

Properties

Objects of this class have the following properties. See [“IDLgrLight Properties”](#) on page 3742 for details on individual properties.

- [ALL](#)
- [ATTENUATION](#)
- [COLOR](#)
- [CONEANGLE](#)
- [DIRECTION](#)
- [FOCUS](#)
- [HIDE](#)
- [INTENSITY](#)
- [LOCATION](#)
- [PALETTE](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)

- [TYPE](#)
- [XCOORD_CONV](#)
- [YCOORD_CONV](#)
- [ZCOORD_CONV](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrLight::Cleanup](#)
- [IDLgrLight::GetCTM](#)
- [IDLgrLight::GetProperty](#)
- [IDLgrLight::Init](#)
- [IDLgrLight::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrLight Properties

IDLgrLight objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrLight::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrLight::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrLight::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure that contains the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ATTENUATION

A 3-element floating-point vector of the form [constant, linear, quadratic] that describes the factor by which light intensity is to fall with respect to distance from the light source. ATTENUATION applies only to Positional and Spot lights, as specified by the TYPE property. The overall attenuation factor is computed as follows:

$$\text{attenuation} = 1 / (\text{constant} + \text{linear} * \text{distance} + \text{quadratic} * \text{distance}^2)$$

By default, the values are [1, 0, 0].

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Attenuation		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

COLOR

A three-element byte vector specifying the RGB color of the light. The default is [255, 255, 255], which is a white light. The color of a light is ignored when graphics are sent to graphics destinations using the Indexed color model, in which case light intensities are scaled into the range of colors available on the graphics destination.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CONEANGLE

A floating-point value that specifies the angle (measured in degrees) of coverage for a spotlight. The default is 60.

Property Type	FLOAT		
Name String	Spotlight cone angle		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DIRECTION

A three-element floating-point vector representing the direction in which a spotlight is to be pointed. The default is [0,0,-1].

Note

For directional lights, the light's parallel rays follow a vector beginning at the position specified by LOCATION and ending at [0, 0, 0].

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Spotlight direction		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

FOCUS

A floating-point value that describes the attenuation of intensity for spotlights as the distance from the center of the cone of coverage increases. This factor is used as an exponent to the cosine of the angle between the direction of the spotlight and the direction from the light to the vertex being lighted. The default is 0.0.

Property Type	FLOAT		
Name String	Spotlight attenuation		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value that indicates whether this light should be enabled:

- 0 = Enable light (the default)
- 1 = Disable light

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Note

If no lights are present in the view (or if all lights in the view are hidden), an ambient light will be provided by default.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

INTENSITY

A floating-point value between 0.0 (darkest) and 1.0 (brightest) that indicates the intensity of the light. The default is 1.0.

Property Type	FLOAT		
Name String	Intensity		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LOCATION

A floating-point vector of the form $[x, y, z]$ describing the position of the light. By default, the position is $[0, 0, 0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Location		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the [IDLgrPalette](#) object class) that specifies the red, green, and blue values of the color lookup table to be associated with the image if it is an indexed color image. This property is ignored if the image is a greyscale or RGB image.

Note

This table is only used when the destination is an RGB model device. The Indexed color model writes the indices directly to the device. In order to ensure that these colors are used when the image is displayed, this palette must be copied to the graphics destination's palette for any graphics destination that uses the Indexed color model.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

TYPE

An integer that indicates the type of light. Valid values are:

- 0 = Ambient light. An ambient light is a universal light source, which has no direction or position. An ambient light illuminates every surface in the scene equally, which means that no edges are made visible by contrast. Ambient lights control the overall brightness and color of the entire scene. If no value is specified for the TYPE property, an ambient light is created.
- 1 = Positional light. A positional light supplies divergent light rays, and will make the edges of surfaces visible by contrast if properly positioned. A positional light source can be located anywhere in the scene.

- 2 = Directional light. A directional light supplies parallel light rays. The effect is that of a positional light source located at an infinite distance from scene.
- 3 = Spot light. A spot light illuminates only a specific area defined by the light's position, direction, and the cone angle, or angle which the spotlight covers.

In a property sheet, this property appears as an enumerated list with the following options:

- Ambient
- Positional
- Directional
- Spotlight

Property Type	ENUMLIST		
Name String	Type		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors that convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Double-precision floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors that convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Double-precision floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors that convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Double-precision floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLgrLight::Cleanup

The IDLgrLight::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrLight::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrLight::GetCTM

The IDLgrLight::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrLight::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the light object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrLight::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrLight::GetProperty

The IDLgrLight::GetProperty procedure method retrieves the value of a property or group of properties for the light.

Syntax

Obj->[IDLgrLight::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrLight Properties](#)” on page 3742 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrLight::Init

The IDLgrLight::Init function method initializes the light object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrLight' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrLight::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “IDLgrLight Properties” on page 3742 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrLight:: SetProperty

The IDLgrLight::SetProperty procedure method sets the value of a property or group of properties for the light.

Syntax

Obj->[IDLgrLight::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrLight Properties](#)” on page 3742 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrModel

A model object represents a graphical item or group of items that can be transformed (rotated, scaled, and/or translated). It serves as a container of other IDLgrModel objects or atomic graphic objects. IDLgrModel applies a transform to the current view tree.

Superclasses

[IDL_Container](#)

[IDLitComponent](#)

Creation

See “[IDLgrModel::Init](#)” on page 3772.

Properties

Objects of this class have the following properties. See “[IDLgrModel Properties](#)” on page 3758 for details on individual properties.

- [ALL](#)
- [CLIP_PLANES](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_TEST_FUNCTION](#)
- [DEPTH_WRITE_DISABLE](#)
- [HIDE](#)
- [LIGHTING](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [SELECT_TARGET](#)
- [TRANSFORM](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrModel::Add](#)
- [IDLgrModel::Cleanup](#)
- [IDLgrModel::Draw](#)
- [IDLgrModel::GetByName](#)
- [IDLgrModel::GetCTM](#)
- [IDLgrModel::GetProperty](#)
- [IDLgrModel::Init](#)
- [IDLgrModel::Reset](#)
- [IDLgrModel::Rotate](#)
- [IDLgrModel::Scale](#)
- [IDLgrModel::SetProperty](#)
- [IDLgrModel::Translate](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrModel Properties

IDLgrModel objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrModel::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrModel::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrModel::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with this object.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

CLIP_PLANES

An 4-by- N floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$.

Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Note - Clipping planes are applied in the data space of the objects this model contains (prior to the application of this model's transform).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Clipping planes		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always

sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.

- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw model and children (the default)
- 1 = Do not draw model or children

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Note

HIDE only controls the display attributes of IDLgrModel children since the IDLgrModel object itself lacks geometry.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LIGHTING

An integer value that indicates whether lighting is to be enabled or disabled for all atomic graphic objects that have this model as a parent. IDLgrModel objects that have this model as a parent will not be effected, as they have their own value for this

property. If this value is set to 0, any lights added as children of this model will be used to illuminate any other models in the view hierarchy that have lighting enabled.

- 0 = Disable lighting
- 1 = Enable single-sided lighting
- 2 = Enable double-sided lighting (the default)

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Lighting		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

SELECT_TARGET

A Boolean value that tags the model object as the target object to be returned when any object contained by the model is selected via the `IDLgrWindow::Select` method.

- 0 = Do not tag the model object as the target object (default).

- 1 = Tag the model object as the target object.

By default, an IDLgrModel object cannot be returned as the target of a SELECT operation since it contains no geometry.

This property is registered as a Boolean property, but it is hidden by default.

Property Type	BOOLEAN		
Name String	Select target		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TRANSFORM

A 4-by-4 floating-point transformation matrix to be applied to the object. This matrix will be multiplied by its parent's transformation matrix (if the parent has one). The default is the identity matrix. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Model transform		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

IDLgrModel::Add

The IDLgrModel::Add procedure method adds a child to this Model.

Syntax

Obj->[IDLgrModel::]Add, *Object* [, /ALIAS] [, POSITION=*index*]

Arguments

Object

An instance of an atomic graphic object or another model object to be added to the model object.

Keywords

ALIAS

Set this keyword to a nonzero value to indicate that an alias—rather than the object itself—is to be added to the model. With this keyword you can add the same object to multiple models without duplicating that object and its children. If this keyword is set, the PARENT keyword on the object being added will not change. Furthermore, if this keyword is set, the object being added will not be destroyed when the model is destroyed.

POSITION

Set this keyword equal to the zero-based index of the position within the container at which the new object should be placed.

Version History

Introduced: 5.0

IDLgrModel::Cleanup

The IDLgrModel::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrModel::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrModel::Draw

The IDLgrModel::Draw procedure method draws the specified picture to the specified graphics destination. *This method is provided for purposes of sub-classing only, and is intended to be called only from the Draw method of a subclass of IDLgrModel.*

Note

Objects are drawn to the destination device in the order that they are added to the model, view, viewgroup, or scene object that contains them.

Syntax

Obj->[IDLgrModel::]Draw, Destination, Picture

Arguments

Destination

The destination object ([IDLgrBuffer](#), [IDLgrClipboard](#), [IDLgrPrinter](#), or [IDLgrWindow](#)) to which the specified view object will be drawn.

Picture

The view (an instance of an [IDLgrView](#) object), viewgroup (an instance of an [IDLgrViewgroup](#) object), or scene (an instance of an [IDLgrScene](#) object) to be drawn.

Keywords

None

Version History

Introduced: 5.0

IDLgrModel::GetByName

The IDLgrModel::GetByName function method finds contained objects by name and returns the object reference to the named object. If the named object is not found, the GetByName function returns a null object reference.

Note

The GetByName function does *not* perform a recursive search through the object hierarchy. If a fully qualified object name is not specified, only the contents of the current container object are inspected for the named object.

Syntax

Result = *Obj*->[IDLgrModel::]GetByName(*Name*)

Return Value

Returns the object reference to the named object or a null object reference.

Arguments

Name

A string containing the name of the object to be returned.

Object naming syntax is very much like the syntax of a UNIX file system. Objects contained by other objects can include the name of their parent object; this allows you to create a fully qualified name specification. For example, if `object1` contains `object2`, which in turn contains `object3`, the string specifying the fully qualified object name of `object3` would be `'object1/object2/object3'`.

Object names are specified relative to the object on which the GetByName method is called. If used at the beginning of the name string, the `/` character represents the top of an object hierarchy. The string `'..'` represents the object one level “up” in the hierarchy.

Keywords

None

Version History

Introduced: 5.0

IDLgrModel::GetCTM

The IDLgrModel::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrModel::]GetCTM( [, DESTINATION=objref]  
    [, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the model object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrModel::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrModel::GetProperty

The IDLgrModel::GetProperty procedure method retrieves the value of a property or group of properties for the model.

Syntax

Obj->[IDLgrModel:]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrModel Properties](#)” on page 3758 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrModel::Init

The IDLgrModel::Init procedure method initializes the model object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrModel' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrModel::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrModel Properties](#)” on page 3758 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES keyword: 5.6

IDLgrModel::Reset

The IDLgrModel::Reset procedure method sets the current transform matrix for the model object to the identity matrix.

Note

Using this method is functionally identical to the following statement:

```
Obj->[IDLgrModel::]SetProperty, TRANSFORM=IDENTITY(4)
```

Syntax

```
Obj->[IDLgrModel:]Reset
```

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrModel::Rotate

The IDLgrModel::Rotate procedure method rotates the model about the specified axis by the specified angle. IDL computes and maintains the resulting transform matrix in double-precision floating-point.

Syntax

Obj->[IDLgrModel:]Rotate, *Axis*, *Angle* [, /PREMULTIPLY]

Arguments

Axis

A three-element vector of the form $[x, y, z]$ describing the axis about which the model is to be rotated.

Angle

The angle (measured in degrees) by which the rotation is to occur.

Keywords

PREMULTIPLY

Set this keyword to cause the rotation matrix specified by *Axis* and *Angle* to be pre-multiplied to the model's transformation matrix. By default, the rotation matrix is post-multiplied.

Version History

Introduced: 5.0

IDLgrModel::Scale

The IDLgrModel::Scale procedure method scales the model by the specified scaling factors. IDL computes and maintains the resulting transform matrix in double-precision floating-point.

Syntax

Obj->[IDLgrModel:]Scale, *Sx*, *Sy*, *Sz* [, /PREMULTIPLY]

Arguments

Sx, Sy, Sz

The scaling factors in the *x*, *y*, and *z* dimensions by which the model is to be scaled.

Keywords

PREMULTIPLY

Set this keyword to cause the scaling matrix specified by *Sx*, *Sy*, *Sz* to be pre-multiplied to the model's transformation matrix. By default, the scaling matrix is post-multiplied.

Version History

Introduced: 5.0

IDLgrModel:: SetProperty

The IDLgrModel::SetProperty procedure method sets the value of a property or group of properties for the model.

Syntax

Obj->[IDLgrModel::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrModel Properties](#)” on page 3758 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrModel::Translate

The IDLgrModel::Translate procedure method translates the model about the specified axis by the specified translation offsets. IDL computes and maintains the resulting transform matrix in double-precision floating-point.

Syntax

Obj->[IDLgrModel::]Translate, *Tx*, *Ty*, *Tz* [, /PREMULTIPLY]

Arguments

Tx, Ty, Tz

The offsets in *X*, *Y*, and *Z*, respectively, by which the model is to be translated.

Keywords

PREMULTIPLY

Set this keyword to cause the translation matrix specified by *Tx*, *Ty*, *Tz* to be pre-multiplied to the model's transformation matrix. By default, the translation matrix is post-multiplied.

Version History

Introduced: 5.0

IDLgrMPEG

An IDLgrMPEG object creates an MPEG movie file from an array of image frames.

Note

The MPEG standard does not allow movies with odd numbers of pixels to be created.

Note

MPEG support in IDL requires a special license. For more information, contact your RSI sales representative or technical support.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrMPEG::Init](#)” on page 3790.

Properties

Objects of this class have the following properties. See “[IDLgrMPEG Properties](#)” on page 3781 for details on individual properties.

- [ALL](#)
- [BITRATE](#)
- [DIMENSIONS](#)
- [FILENAME](#)
- [FORMAT](#)
- [FRAME_RATE](#)
- [IFRAME_GAP](#)
- [INTERLACED](#)
- [MOTION_VEC_LENGTH](#)
- [QUALITY](#)
- [SCALE](#)

- [STATISTICS](#)
- [TEMP_DIRECTORY](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrMPEG::Cleanup](#)
- [IDLgrMPEG::GetProperty](#)
- [IDLgrMPEG::Init](#)
- [IDLgrMPEG::Put](#)
- [IDLgrMPEG::Save](#)
- [IDLgrMPEG::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrMPEG Properties

IDLgrMPEG objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrMPEG::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrMPEG::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrMPEG::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the retrievable properties associated with this object.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

BITRATE

A double-precision floating-point value to specify the MPEG movie bit rate. Higher bit rates will create higher quality MPEGs but will increase file size. The following table describes the valid values:

MPEG Version	Range
MPEG 1	0.1 to 104857200.0
MPEG 2	0.1 to 429496729200.0

Table 8-6: BITRATE Value Range

Set this property to 0.0 (the default setting) to indicate that IDL should compute the BITRATE value based upon the value you have specified for the QUALITY property. The value of BITRATE returned by [IDLgrMPEG::GetProperty](#) is either the value computed by IDL from the QUALITY value or the last non-zero valid value stored in this property.

Note

Only use the BITRATE property if changing the QUALITY property value does not produce the desired results. It is highly recommended to set the BITRATE to at least several times the frame rate to avoid unusable MPEG files or file generation errors.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DIMENSIONS

A two-element integer array specifying the dimensions (in pixels) of each of the images to be used as frames for the movie. If this property is not specified, the dimensions of the first image loaded will be used. Once `IDLgrMPEG::Put` has been called, this property can no longer be set.

Note

When creating MPEG files, you must be aware of the capabilities of the MPEG decoder you will be using to view it. Some decoders only support a limited set of sampling and bitrate parameters to normalize computational complexity, buffer size, and memory bandwidth. For example, the Windows Media Player supports a limited set of sampling and bitrate parameters. In this case, it is best to use 352 x 240 x 30 fps or 352 x 288 x 25 fps when determining the dimensions and frame rate for your MPEG file. When opening a file in Windows Media Player that does not use these dimensions, you will receive a “Bad Movie File” error message. The file is not “bad”, this decoder just doesn’t support the dimensions of the MPEG.

FILENAME

A string that represents the name of the file in which the encoded MPEG sequence is to be stored. The default is 'idl.mpg'.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FORMAT

A Boolean value that specifies the type of MPEG encoding to use:

- 0 = MPEG1 (the default)
- 1 = MPEG2

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FRAME_RATE

An integer value that specifies the frame rate used in creating the MPEG file:

Value	Descriptions
1	23.976 frames/sec: NTSC encapsulated film rate
2	24 frames/sec: Standard international film rate
3	25 frames/sec: PAL video frame rate
4	29.97 frames/sec: NTSC video frame rate
5	30 frames/sec: NTSC drop frame video frame rate (the default)
6	50 frames/sec: Double frame rate/progressive PAL
7	59.94 frames/sec: Double frame rate NTSC
8	60 frames/sec: Double frame rate NTSC drop frame video

Table 8-7: FRAME_RATE Values

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IFRAME_GAP

A positive integer value that specifies the number of frames between I frames to be created in the MPEG file. I frames are full-quality image frames that may have a number of predicted or interpolated frames between them.

Set this property to 0 (the default setting) to indicate that IDL should compute the IFRAME_GAP value based upon the value you have specified for the QUALITY property. The value of IFRAME_GAP returned by `IDLgrMPEG::GetProperty` is either the value computed by IDL from the QUALITY value or the last non-zero valid value stored in this property.

Note

Only use the IFRAME_GAP property if changing the QUALITY property value does not produce the desired results.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

INTERLACED

A Boolean value that indicates whether frames in the encoded MPEG file should be interlaced.

- 0 = Non-interlaced (the default)
- 1 = Interlaced

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

MOTION_VEC_LENGTH

An integer value that specifies the length of the motion vectors to be used to generate predictive frames. The following table describes the valid values:

Value	Description
1	Small motion vectors.
2	Medium motion vectors.
3	Large motion vectors.

Table 8-8: MOTION_VEC_LENGTH Values

0 (the default setting) indicates that IDL should compute the MOTION_VEC_LENGTH value based upon the value you have specified for the QUALITY property. The value of MOTION_VEC_LENGTH returned by [IDLgrMPEG::GetProperty](#) is either the value computed by IDL from the QUALITY value or the last non-zero value stored in this property.

Note

Only use the MOTION_VEC_LENGTH property if changing the QUALITY value does not produce the desired results.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

QUALITY

An integer value between 0 (low quality) and 100 (high quality), inclusive, that specifies the quality at which the MPEG stream is to be stored. Higher quality values result in lower rates of time compression and less motion prediction which provide higher quality MPEGs but with substantially larger file size. Lower quality factors may result in longer MPEG generation times. The default is 50.

Note

Since MPEG uses JPEG (lossy) compression, the original picture quality cannot be reproduced even when setting QUALITY to its highest setting.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SCALE

A two-element floating-point vector, [*xscale*, *yscale*], indicating the scale factors to be stored with the MPEG file as hints for playback. The default is [1.0, 1.0], indicating that the movie should be played back at the dimensions of the stored image frames.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

STATISTICS

A Boolean value that determines whether to save statistical information about MPEG encoding for the supplied image frames in a file when the IDLgrMPEG::Save method is called. The information will be saved in a file with a name that matches that specified by the FILENAME property, with the extension “.stat”.

- 0 = Statistics are not saved (the default).
- 1 = Save statistics.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TEMP_DIRECTORY

A string value that specifies the directory in which to place temporary files while creating the MPEG movie file. The default value is platform specific.

Property Type	String		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLgrMPEG::Cleanup

The IDLgrMPEG::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrMPEG::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrMPEG::GetProperty

The IDLgrMPEG::GetProperty procedure method retrieves the value of a property or group of properties for the MPEG object.

Syntax

Obj->[IDLgrMPEG::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrMPEG Properties](#)” on page 3781 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrMPEG::Init

The IDLgrMPEG::Init function method initializes the MPEG object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Note

MPEG support in IDL requires a special license. For more information, contact your RSI sales representative or technical support.

Syntax

```
Obj = OBJ_NEW('IDLgrMPEG' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrMPEG::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under [“IDLgrMPEG Properties”](#) on page 3781 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

IDLgrMPEG::Put

The IDLgrMPEG::Put procedure method puts a given image into the MPEG sequence at the specified frame. Note that all images in a given MPEG movie must have matching dimensions. If no dimensions were explicitly specified when the MPEG object was initialized, the dimensions will be set according to the dimensions of the first image.

Syntax

Obj->[IDLgrMPEG::]Put, *Image*[, *Frame*]

Arguments

Image

An instance of an IDLgrImage object or a $m \times n$ or $3 \times m \times n$ array representing the image to be loaded at the given frame.

Frame

An integer specifying the index of the frame at which the image is to be added. Frame indices start at zero. If *Frame* is not supplied, the frame number used will be one more than the last frame that was put. Frame number need not be consecutive; in case of a gap in frame numbers, the frame before the gap is repeated to fill the space.

Keywords

None

Version History

Introduced: 5.1

IDLgrMPEG::Save

The IDLgrMPEG::Save procedure method encodes and saves the MPEG sequence to the specified filename.

Note

The MPEG standard does not allow movies with odd numbers of pixels to be created.

Syntax

```
Obj->[IDLgrMPEG::]Save [, FILENAME=string]
```

Arguments

None

Keywords

FILENAME

Set this keyword to a string representing the name of the file in which the encoded MPEG sequence is to be stored. The default is `idl.mpg`.

Obsolete Keywords

The following keywords are obsolete:

- CREATOR_TYPE

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Version History

Introduced: 5.1

IDLgrMPEG:: SetProperty

The IDLgrMPEG::SetProperty procedure method sets the value of a property or group of properties for the MPEG object.

Syntax

Obj->[IDLgrMPEG::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrMPEG Properties](#)” on page 3781 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrPalette

A palette object represents a color lookup table that maps indices to red, green, and blue values.

Superclasses

This class has no superclass.

Creation

See “[IDLgrPalette::Init](#)” on page 3803.

Properties

Objects of this class have the following properties. See “[IDLgrPalette Properties](#)” on page 3797 for details on individual properties.

- [ALL](#)
- [BLUE_VALUES](#)
- [BOTTOM_STRETCH](#)
- [GAMMA](#)
- [GREEN_VALUES](#)
- [N_COLORS](#)
- [RED_VALUES](#)
- [TOP_STRETCH](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has this following methods:

- [IDLgrPalette::Cleanup](#)
- [IDLgrPalette::GetRGB](#)
- [IDLgrPalette::GetProperty](#)
- [IDLgrPalette::Init](#)

- [IDLgrPalette::LoadCT](#)
- [IDLgrPalette::NearestColor](#)
- [IDLgrPalette::SetRGB](#)
- [IDLgrPalette::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrPalette Properties

IDLgrPalette objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrPalette::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrPalette::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrPalette::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure that contains the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

BLUE_VALUES

A byte vector containing the blue values for the color palette. Setting this value is the same as specifying the *aBlue* argument to the [IDLgrPalette::Init](#) method.

Property Type	Byte vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

BOTTOM_STRETCH

An integer value in the range of $0 \leq Value \leq 100$ to indicate what percentage of the palette entries at the bottom of the palette should be filled with the value of the first palette entry. The entire range of red, green, and blue values will be compressed to fit within the range of palette entries beginning at this entry and ending at the entry specified by the value of the TOP_STRETCH property. The default is 0 (zero).

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GAMMA

A floating-point value that indicates the gamma value to be applied to the color palette. This value should be in the range of $0.1 \leq Gamma \leq 10.0$. The default is 1.0.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GREEN_VALUES

A byte vector containing the green values for the color palette. Setting this value is the same as specifying the *aGreen* argument to the IDLgrPalette::Init method.

Property Type	Byte-vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_COLORS

An integer value that determines the number of elements in the color palette.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

RED_VALUES

A byte vector containing the red values for the color palette. Setting this value is the same as specifying the *aRed* argument to the `IDLgrPalette::Init` method.

Property Type	Byte vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TOP_STRETCH

A floating-point value in the range of $0 \leq \text{Value} \leq 100$ to indicate what percentage of the palette entries at the top of the palette should be filled with the value of the last palette entry. The entire range of red, green, and blue values will be compressed to fit within the range of palette entries beginning at the entry specified by the value of the `BOTTOM_STRETCH` property and ending at this entry. The default is 0 (zero).

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLgrPalette::Cleanup

The IDLgrPalette::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrPalette::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPalette::GetRGB

The IDLgrPalette::GetRGB function method returns the RGB values contained in the palette at the given index. The returned value is a three-element vector of the form [red, green, blue].

Syntax

Result = *Obj*->[IDLgrPalette::]GetRGB(*Index*)

Return Value

Returns the RGB values contained in the palette at the given index.

Arguments

Index

The index whose RGB values are desired. This value should be in the range of $0 \leq \textit{Index} < \text{N_COLORS}$, where N_COLORS is the number of elements in the color palette, as returned by the N_COLORS keyword to the IDLgrPalette:GetProperty method.

Keywords

None

Version History

Introduced: 5.0

IDLgrPalette::GetProperty

The IDLgrPalette::GetProperty procedure method retrieves the value of a property or group of properties for the palette.

Syntax

Obj->[IDLgrPalette::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrPalette Properties](#)” on page 3797 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrPalette::Init

The IDLgrPalette::Init function method initializes a palette object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrPalette', aRed, aGreen, aBlue [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrPalette::]Init([aRed, aGreen, aBlue] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

aRed

A vector containing the red values for the color palette. These values should be within the range of $0 \leq \text{Value} \leq 255$. The number of elements comprising the *aRed* vector must not exceed 256.

aGreen

A vector containing the green values for the color palette. These values should be within the range of $0 \leq \text{Value} \leq 255$. The number of elements comprising the *aGreen* vector must not exceed 256.

aBlue

A vector containing the blue values for the color palette. These values should be within the range of $0 \leq \text{Value} \leq 255$. The number of elements comprising the *aBlue* vector must not exceed 256.

Keywords

Any property listed under “[IDLgrPalette Properties](#)” on page 3797 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrPalette::LoadCT

The IDLgrPalette::LoadCT procedure method loads one of the IDL predefined color tables into an IDLgrPalette object.

Syntax

Obj->[IDLgrPalette::]LoadCT, *TableNum* [, **FILE**=*colortable filename*]

Arguments

TableNum

The number of the pre-defined IDL color table to load, from 0 to 40.

Keywords

FILE

Set this keyword to the name of a colortable file to be used instead of the file `colors1.tbl` in the IDL distribution. The `MODIFYCT` procedure can be used to create and modify colortable files.

Version History

Introduced: 5.0

IDLgrPalette::NearestColor

The IDLgrPalette::NearestColor function method returns the index of the color in the palette that best matches the given RGB values.

Syntax

Result = Obj->[IDLgrPalette::]NearestColor(Red, Green, Blue)

Return Value

Returns the index of the color in the palette that best matches the given RGB values.

Arguments

Red

The red value of the color that should be matched. This value should be within the range of $0 \leq \text{Value} \leq 255$.

Green

The green value of the color that should be matched. This value should be within the range of $0 \leq \text{Value} \leq 255$.

Blue

The blue value of the color that should be matched. This value should be within the range of $0 \leq \text{Value} \leq 255$.

Keywords

None

Version History

Introduced: 5.0

IDLgrPalette::SetRGB

The IDLgrPalette::SetRGB procedure method sets the color values at a specified index in the palette to the specified Red, Green and Blue values.

Syntax

Obj->[IDLgrPalette::]SetRGB, *Index*, *Red*, *Green*, *Blue*

Arguments

Index

The index within the Palette object to be set. This value should be in the range of $0 \leq \textit{Value} < \textit{N_COLORS}$.

Red

The red value to set in the color palette.

Green

The green value to set in the color palette.

Blue

The blue value to set in the color palette.

Keywords

None

Version History

Introduced: 5.0

IDLgrPalette:: SetProperty

The IDLgrPalette::SetProperty procedure method sets the value of a property or group of properties for the palette.

Syntax

Obj->[IDLgrPalette::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrPalette Properties](#)” on page 3797 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrPattern

A pattern object describes which pixels are filled and which are left blank when an area is filled. Pattern objects are used by setting the `FILL_PATTERN` property of a polygon object equal to the object reference of the pattern object.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrPattern::Init](#)” on page 3816.

Properties

Objects of this class have the following properties. See “[IDLgrPattern Properties](#)” on page 3811 for details on individual properties.

- [ALL](#)
- [ORIENTATION](#)
- [PATTERN](#)
- [SPACING](#)
- [STYLE](#)
- [THICK](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has this following methods:

- [IDLgrPattern::Cleanup](#)
- [IDLgrPattern::GetProperty](#)
- [IDLgrPattern::Init](#)
- [IDLgrPattern::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrPattern Properties

IDLgrPattern objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrPattern::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrPattern::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrPattern::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ORIENTATION

An integer representing the angle (measured in degrees counterclockwise from the horizontal) of the lines used in a Line Fill. This property is ignored unless the *Style* argument (or STYLE property) is set to one.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PATTERN

A 32 x 32 bit array (bitmap) describing the pattern that will be tiled over a polygon when a pattern fill is used. The bitmap must be configured as a 4 x 32 “bitmap byte array” as created by the CVTTOBM function. Each bit that is a 1 is drawn, each bit that is 0 is not drawn. Each bit in this array represents a 1 point by 1 point square area of pixels on the destination device. This property is ignored unless the *Style* argument (or STYLE property) is set to 2.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SPACING

A floating-point value representing the distance (measured in points) between the lines used for a Line Fill. This property is ignored unless the *Style* argument (or STYLE property) is set to 1. The default is 2.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

STYLE

An integer value that specifies the type of pattern, as follows:

- 0 = Solid (default)
- 1 = Line Fill
- 2 = Pattern

This property is the same as the *Style* argument described in “[IDLgrPattern::Init](#)” on page 3816.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0 that specifies the line thickness to be used to draw the pattern lines for a Line Fill, in points. The default is 1.0 points. This property is ignored unless the *Style* argument or STYLE property is set to 1.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLgrPattern::Cleanup

The IDLgrPattern::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrPattern::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPattern::GetProperty

The IDLgrPattern::GetProperty procedure method retrieves the value of a property or group of properties for the pattern.

Syntax

Obj->[IDLgrPattern::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrPattern Properties](#)” on page 3811 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrPattern::Init

The IDLgrPattern::Init function method initializes the pattern object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrPattern' [, Style] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrPattern::]Init([Style] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Style

A integer value representing the type of pattern. Valid values are:

- 0 = Solid color (default)
- 1 = Line Fill
- 2 = Pattern

Keywords

Any property listed under [“IDLgrPattern Properties”](#) on page 3811 that contains the word “Yes” in the “Init” column of the properties table can be initialized during

object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrPattern:SetProperty

The IDLgrPattern::SetProperty procedure method sets the value of a property or group of properties for the pattern.

Syntax

Obj->[IDLgrPattern::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrPattern Properties](#)” on page 3811 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrPlot

A plot object creates a set of polylines connecting data points in two-dimensional space.

An IDLgrPlot object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrPlot::Init](#)” on page 3841.

Properties

Objects of this class have the following properties. See “[IDLgrPlot Properties](#)” on page 3821 for details on individual properties.

- [ALL](#)
- [CLIP_PLANES](#)
- [DATA](#)
- [DATAY](#)
- [DEPTH_TEST_FUNCTION](#)
- [DOUBLE](#)
- [HISTOGRAM](#)
- [MAX_VALUE](#)
- [NSUM](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [SHARE_DATA](#)
- [THICK](#)
- [ALPHA_CHANNEL](#)
- [COLOR](#)
- [DATAX](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_WRITE_DISABLE](#)
- [HIDE](#)
- [LINESTYLE](#)
- [MIN_VALUE](#)
- [PALETTE](#)
- [POLAR](#)
- [RESET_DATA](#)
- [SYMBOL](#)
- [USE_ZVALUE](#)

- `VERT_COLORS`
- `XRANGE`
- `YRANGE`
- `ZRANGE`
- `XCOORD_CONV`
- `YCOORD_CONV`
- `ZCOORD_CONV`
- `ZVALUE`

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrPlot::Cleanup`
- `IDLgrPlot::GetCTM`
- `IDLgrPlot::GetProperty`
- `IDLgrPlot::Init`
- `IDLgrPlot::SetProperty`

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrPlot Properties

IDLgrPlot objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrPlot::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrPlot::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrPlot::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

A floating-point value in the range [0.0, 1.0] to specify the opacity of the plot. The default value of 1.0 causes IDL to draw the plot completely opaque. If the value of this property is less than 1.0, then the pixels of the plot are blended with the pixels already on the screen, where the color of the plot is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha. Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results.

This property has no effect on devices using indexed color mode. If 3-channel (RGB) VERT_COLORS are supplied, then the value of this property is used for the alpha at each vertex when drawing the plot line. If 4-channel (RGBA) VERT_COLORS are

supplied, then the alpha from each vertex color is used for the alpha at each vertex when drawing the plot line.

When a plot is drawn with symbols, the symbol's color and alpha values are always used to draw the symbols unless VERT_COLORS are supplied. If the VERT_COLORS are RGB, then the alpha value of the symbol is used to draw the symbol. If the VERT_COLORS are RGBA, then the alpha from the vertex color is used to draw the symbol.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see [“Alpha Channel and Objects”](#) in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by-*N* floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any *x*, *y*, or *z* coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES keyword of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

The color to be used as the foreground color for this plot. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default is [0, 0, 0].

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DATA

The plot data of any type in a $3 \times n$ array, [*DataX, DataY, DataZ*].

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DATA X

A vector of any type that specifies the X values to be plotted. This property is the same as the X argument.

Property Type	Vector of any type		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

DATAY

A vector of any type that specifies the Y values to be plotted. This property is the same as the Y argument.

Property Type	Vector of any type		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DOUBLE

A Boolean value that indicates whether data provided by any of the input arguments will be stored in this object as using double-precision floating-point format.

- Set this property equal to 1 to convert input data to double-precision floating-point format.
- Set this property equal to 0 to convert input data to single-precision floating-point format.
- If you do not specify a value for this property, no data type conversion will be performed, and the data will be stored with its original precision.

Setting this property may be desirable if the data consists of large integers that cannot be accurately represented in single-precision floating-point arithmetic. This property is also automatically set to 1 if any of the input arguments are stored using a variable of type DOUBLE.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value that indicates whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HISTOGRAM

A Boolean value that determines whether to use only horizontal and vertical lines to connect the plotted points.

- 0 = The points are connected using a single straight line (the default).
- 1 = Use only horizontal and vertical lines to connect the plotted points.

Property Type	BOOLEAN		
Name String	Histogram plot		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LINestyle

An integer value that indicates the line style to be used to draw the plot lines. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the `LINestyle` property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

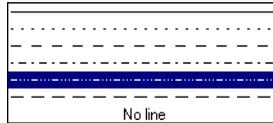
To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the

bitmask should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq repeat \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINestyle = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

In a property sheet, this property appears as follows:



Property Type	LINestyle		
Name String	Line style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MAX_VALUE

A double-precision floating-point value that determines the maximum value to be plotted. When this property is set, data values greater than the value of `MAX_VALUE` are treated as missing data and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Maximum value		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MIN_VALUE

A double-precision floating-point value that determines the minimum value to be plotted. If this property is present, data values less than the value of `MIN_VALUE` are treated as missing data and are not plotted. Note that the IEEE floating-point value

NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Minimum value		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

NSUM

An integer value representing the number of data points to average when plotting. If NSUM is larger than 1, every group of NSUM points is averaged to produce one plotted point. If there are M data points, then M/NSUM points are plotted.

Property Type	INTEGER		
Name String	Number of points to average		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class). This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the PALETTE property on this object is not set, the destination object PALETTE property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

POLAR

A Boolean value that determines whether to create a polar plot. The *X* and *Y* arguments must both be present. The *X* argument represents the radius, and the *Y* argument represents the angle expressed in radians.

Property Type	BOOLEAN		
Name String	Polar plot		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESET_DATA

A Boolean value that determines whether to treat the data provided via one of the DATA[XY] properties as a new data set unique to this object, rather than overwriting data that is shared by other objects. There is no reason to use this property if the object on which the property is being set does not currently share data with another

object (that is, if the `SHARE_DATA` property is not in use). This property has no effect if no new data is provided via a `DATA` property.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHARE_DATA

An object reference to an object with which data is to be shared by this plot. A plot may only share data with another plot. The `SHARE_DATA` property is intended for use when data values are not set via an argument to the object's `Init` method or by setting the object's `DATA` property.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SYMBOL

An object reference vector containing instances of the [IDLgrSymbol](#) object class. Each symbol in the vector will be drawn at the corresponding plotted point. If there are more points than elements in `SYMBOL`, the elements of the `SYMBOL` vector are cyclically repeated. By default, no symbols are drawn. To remove symbols from a plot, set the `SYMBOL` property equal to a null object reference.

Property Type	Object reference vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0, specifying the line thickness to be used to draw the plotted lines, in points. The default is 1.0 points.

In a property sheet, this property appears as follows:



Property Type	THICKNESS		
Name String	Thickness		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

USE_ZVALUE

A Boolean value that determines whether to use the current ZVALUE. The plot is considered three-dimensional if this property is set.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

VERT_COLORS

A vector of colors to be used to specify the color of the plot segment between each vertex. The vector may be of the form [n] where each entry is a color index, or of the form [3,n] where each 3-element row is an RGB color, or of the form [4,n] where each 4-element row is an RGBA color. To remove vertex colors after they have been set, set VERT_COLORS to a scalar.

If VERT_COLORS is not specified, the entire plot is drawn in the single color provided by the COLOR property, which is the default action.

If there are more vertices than elements in VERT_COLORS, the elements of VERT_COLORS are cyclically repeated.

When VERT_COLORS is used with symbols, the vertex colors and vertex alpha override any colors and alpha specified by the symbol object or any colors and alpha specified by graphic atoms contained in a user-defined symbol.

Property Type	USERDEF		
Name String	Vertex colors		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form $[xmin, xmax]$ specifying the range of x data coordinates covered by the graphic object. If this property is not specified, the minimum and maximum data values are used. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form $[ymin, ymax]$ specifying the range of y data values covered by the graphic object. If this property is not specified, the minimum and maximum data values are used. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element floating-point vector of the form [*zmin*, *zmax*] specifying the range of *z* data values covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Note

The XRANGE and YRANGE properties can also be retrieved via the GetProperty method; ZRANGE, however, can only be retrieved, not initialized (Init method) or set (SetProperty method).

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZVALUE

A floating-point value to be used as the Z coordinate for the entire plot. By default, 0.0 is used as the Z coordinate.

Note

The USE_ZVALUE property needs to be set in order for ZVALUE to take effect.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Version History

ALPHA_CHANNEL added: 6.1

IDLgrPlot::Cleanup

The IDLgrPlot::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrPlot:]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPlot::GetCTM

The IDLgrPlot::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrPlot::]GetCTM( [, DESTINATION=objref] [, PATH=objref(s)]  
[, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the plot object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrPlot::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrPlot::GetProperty

The IDLgrPlot::GetProperty procedure method retrieves the value of the property or group of properties for the plot.

Syntax

Obj->[IDLgrPlot::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrPlot Properties](#)” on page 3821 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrPlot::Init

The IDLgrPlot::Init function method initializes the plot object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrPlot' [, [X,] Y] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrPlot::]Init([[X,] Y] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

X

A vector representing the abscissa values to be plotted. If X is provided, Y is plotted as a function of X. The value for this argument is double-precision floating-point if the DOUBLE keyword is set or the inputted value is of type DOUBLE. Otherwise it is converted to single-precision floating-point.

Y

Either a vector of two-element arrays [x, y] representing the points to be plotted, or a vector representing the ordinate values to be plotted. If Y is a vector of ordinate values and X is not specified, Y is plotted as a function of the vector index of Y. The value for this argument is double-precision floating-point if the DOUBLE keyword is set or the

inputted value is of type DOUBLE. Otherwise it is converted to single-precision floating-point.

Keywords

Any property listed under [“IDLgrPlot Properties”](#) on page 3821 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES keyword added: 5.6

IDLgrPlot:: SetProperty

The IDLgrPlot::SetProperty procedure method sets the value of the property or group of properties for the plot.

Syntax

Obj->[IDLgrPlot::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrPlot Properties](#)” on page 3821 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrPolygon

A polygon object represents one or more polygons that share a given set of vertices and rendering attributes. All polygons must be convex—that is, a line connecting any pair of vertices on the polygon cannot fall outside the polygon. Concave polygons can be converted to a set of convex polygons using the [IDLgrTessellator](#) object.

An IDLgrPolygon object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrPolygon::Init](#)” on page 3872.

Properties

Objects of this class have the following properties. See “[IDLgrPolygon Properties](#)” on page 3846 for details on individual properties.

- [ALL](#)
- [ALPHA_CHANNEL](#)
- [AMBIENT](#)
- [BOTTOM](#)
- [CLIP_PLANES](#)
- [COLOR](#)
- [DATA](#)
- [DEPTH_OFFSET](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_TEST_FUNCTION](#)
- [DEPTH_WRITE_DISABLE](#)
- [DIFFUSE](#)
- [DOUBLE](#)
- [EMISSION](#)
- [FILL_PATTERN](#)
- [HIDDEN_LINES](#)
- [HIDE](#)
- [LINESTYLE](#)
- [NORMALS](#)
- [PALETTE](#)
- [PARENT](#)
- [POLYGONS](#)
- [REGISTER_PROPERTIES](#)
- [REJECT](#)

- `RESET_DATA`
- `SHADING`
- `SHININESS`
- `STYLE`
- `TEXTURE_INTERP`
- `THICK`
- `XCOORD_CONV`
- `YCOORD_CONV`
- `ZCOORD_CONV`
- `ZRANGE`
- `SHADE_RANGE`
- `SHARE_DATA`
- `SPECULAR`
- `TEXTURE_COORD`
- `TEXTURE_MAP`
- `VERT_COLORS`
- `XRANGE`
- `YRANGE`
- `ZERO_OPACITY_SKIP`

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrPolygon::Cleanup`
- `IDLgrPolygon::GetCTM`
- `IDLgrPolygon::GetProperty`
- `IDLgrPolygon::Init`
- `IDLgrPolygon::SetProperty`

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

`AMBIENT`, `DIFFUSE`, `EMISSION`, `SHININESS`, and `SPECULAR` added: 6.1

IDLgrPolygon Properties

IDLgrPolygon objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrPolygon::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrPolygon::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrPolygon::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

Set this property to a floating-point value in the range [0.0, 1.0] to specify the opacity of the polygon. The default value of 1.0 causes IDL to draw the polygon completely opaque. If the value of this property is less than 1.0, then the pixels of the polygon are blended with the pixels already on the screen. The color of the polygon is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha.

Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results. Note that since a single polygon object may contain many polygons, the ordering of the polygons within the object (POLYGONS property) must be considered as well.

This property has no effect on devices using indexed color mode.

If VERT_COLORS are supplied, then the value of this property is used for the vertex color alpha if the VERT_COLORS do not contain alpha values (3-channel RGB colors). If the VERT_COLORS do contain alpha values (4-channel RGBA colors), then the alpha for each vertex color is used.

When used with textures, the ALPHA_CHANNEL value controls the opacity of the polygon in the same manner as when not using a texture. If the texture contains alpha information, the effective alpha of each texel is the texel's alpha multiplied by the polygon's alpha value (ALPHA_CHANNEL property).

Note

For more detailed information on alpha channel support for IDL Object Graphics, see [“Alpha Channel and Objects”](#) in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

AMBIENT

Set this property to a three-element vector [*red, green, blue*] to specify the ambient reflective color of the object. The default is black or [0,0,0], which specifies that the value of the COLOR property determines the ambient reflective color of the object. Setting an ambient color other than black makes the object use that color for ambient reflectance. Ambient reflectance affects the overall color of the object and is most noticeable where the object is not directly lit by a light source. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Ambient reflective color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

BOTTOM

A three-element vector [*red*, *green*, *blue*] specifying the color used for drawing the bottom of the surface. Set this property to a scalar to draw the bottom with the same color as the top. Setting a bottom color is only supported when the destination device uses RGB color mode.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Bottom color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by-*N* floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any *x*, *y*, or *z* coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

A color index or a three-element vector [*red, green, blue*] specifying the color used to draw polygons. The default color is [0, 0, 0] (black). If the TEXTURE_MAP property is used, the final color is modulated by the texture map pixel values. This property is ignored if the VERT_COLORS property is provided.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DATA

A 2-by-*n* or a 3-by-*n* array of any type which defines, respectively, the 2-D or 3-D vertex data. DATA is equivalent to the optional arguments, X, Y, and Z. This property is stored as double precision floating point values if the property variable is of type DOUBLE or if the DOUBLE property parameter is also specified, otherwise it is converted to single precision floating point.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEPTH_OFFSET

An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.

The units are “Z-Buffer units,” where a value of 1 is used to specify a distance that corresponds to a single step in the device’s Z-Buffer.

Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms. This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for “layering” complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value, such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because a set of DEPTH_OFFSET values may produce better results on one machine than on another. Using IDL’s software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.

Note

DEPTH_OFFSET has no effect unless the value of the STYLE property is 2 (Filled).

Property Type	INTEGER		
Name String	Depth offset		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DIFFUSE

Set this property to a three-element vector [*red*, *green*, *blue*] to specify the diffuse reflectance color of the object. The default is black or [0,0,0], which specifies that the value of the COLOR property determines the diffuse reflective color of the object. Setting a diffuse color other than black makes the object use that color for diffuse reflectance. Diffuse reflectance is an important factor in determining the color of the object and is affected by the color and angle of the light sources in a scene. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Diffuse reflective color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DOUBLE

A Boolean value that indicates whether data provided by any of the input arguments will be stored in this object as using double-precision floating-point format.

- Set this property equal to 1 to convert input data to double-precision floating-point format.
- Set this property equal to 0 to convert input data to single-precision floating-point format.

- If you do not specify a value for this property, no data type conversion will be performed, and the data will be stored with its original precision.

Setting this property may be desirable if the data consists of large integers that cannot be accurately represented in single-precision floating-point arithmetic. This property is also automatically set to 1 if any of the input arguments are stored using a variable of type DOUBLE.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

EMISSION

Set this property to a three-element vector [*red, green, blue*] to specify the emissive color of the object. The default is black or [0,0,0], which specifies that the object contributes no additional color in an emissive sense. Setting an emission color other than black makes the object appear as if it were emitting that color. This property is used to simulate light sources or other objects that emit light in a scene. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Emissive color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

FILL_PATTERN

An object reference to an IDLgrPattern object (or an array of IDLgrPattern objects) that specifies the fill pattern to use for filling the polygons. By default, FILL_PATTERN is set to a null object reference, specifying a solid fill.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDDEN_LINES

A Boolean value that determines whether to draw point and wireframe surfaces using hidden line (point) removal. By default, hidden line removal is disabled.

Property Type	BOOLEAN		
Name String	Remove hidden		
Get: No	Set: Nos	Init: Yes	Registered: Yes

HIDE

A Boolean value that indicates whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LINestyle

An integer value that indicates the line style that should be used to draw the polygon. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINestyle property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot

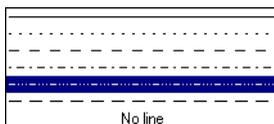
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq repeat \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

In a property sheet, this property appears as follows:



Property Type	LINESTYLE		
Name String	Line style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

NORMALS

A 3-by-*n* floating-point array of unit polygon normals at each vertex. If this property is not set, vertex normals are computed by averaging shared polygon normals at each vertex. Normals are computed using the Right Hand Rule; that is, if the polygon is facing the viewer, vertices are taken in counterclockwise order. To remove previously specified normals, set `NORMALS` to a scalar.

Note

Computing normals is a computationally expensive operation. Rendering speed increases significantly if you supply the surface normals explicitly. You can compute the array of polygon normals used by this property automatically. See [“COMPUTE_MESH_NORMALS”](#) on page 299 for details.

Once you use the NORMALS property in a call to IDLgrPolygon::Init or IDLgrPolygon::SetProperty, you are responsible for that IDLgrPolygon's normals from then on. IDL will not calculate that IDLgrPolygon's normals for you automatically, even if you draw the IDLgrPolygon after vertices or connectivity have been changed.

If you do not use the NORMALS property, IDL calculates normals the first time it draws the IDLgrPolygon. IDL reuses those normals for subsequent draws unless it determines that a fresh recalculation of normals is required, such as if the vertices of the IDLgrPolygon are changed, or you supply new normals via the NORMALS property.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class). This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the PALETTE property on this object is not set, the destination object PALETTE property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: No	Set: No	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

POLYGONS

An integer array of polygon descriptions. A polygon description is an integer or long word array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0..i_{n-1}$ are indices into the X, Y, and Z arguments that represent the polygon vertices. To ignore an entry in the POLYGONS array, set the vertex count, n , to 0. To end the drawing list, even if additional array space is available, set n to -1. If this property is not specified, a single polygon will be generated.

Note

Setting the POLYGONS property to a scalar value will reset the property to its initial, undefined state.

Tip

To ignore an entry in the POLYGONS array, set the entry to 0.

Note

The connectivity array described by POLYGONS allows an individual object to contain more than one polygon. Vertex, normal, and color information can be shared by the multiple polygons. Consequently, the polygon object can represent an entire mesh and compute reasonable normal estimates in most cases.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

REJECT

An integer value to reject polygons as being hidden depending on the orientation of their normals. Select from one of the following values:

- 0 = No polygons are hidden
- 1 = Polygons whose normals point away from the viewer are hidden
- 2 = Polygons whose normals point toward the viewer are hidden

Set this property to zero to draw all polygons regardless of the direction of their normals.

In a property sheet, this property appears as an enumerated list with the following options:

- None
- Normals point away
- Normals point toward

Property Type	ENUMLIST		
Name String	Polygon rejection		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

RESET_DATA

A Boolean value that determines whether to treat the data provided via the DATA property as a new data set unique to this object, rather than overwriting data that is shared by other objects. There is no reason to use this property if the object on which the property is being set does not currently share data with another object (that is, if the SHARE_DATA property is not in use). This property has no effect if no new data is provided via the DATA property.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHADE_RANGE

A two-element integer array that specifies the range of pixel values (color indices) to use for shading. The first element is the color index for the darkest pixel. The second element is the color index for the brightest pixel. The default is [0, 255]. This property is ignored when the polygons are drawn to a graphics destination that uses the RGB color model.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SHADING

An integer that represents the type of shading to use:

- 0 = Flat (default): The color of the first vertex in each polygon is used to define the color for the entire polygon. The color has a constant intensity based upon the normal vector.
- 1 = Gouraud: The colors along each line are interpolated between vertex colors, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

In a property sheet, this property appears as an enumerated list with the following options:

- Flat
- Gouraud

Property Type	ENUMLIST		
Name String	Shading		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SHARE_DATA

An object reference to an object with which data is to be shared by this polygon(s). Polygons may only share data with another polygons object or a polyline. The

SHARE_DATA property is intended for use when data values are not set via an argument to the object's Init method or by setting the object's DATA property.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHININESS

Set this property to a floating-point value between 0.0 and 128.0, inclusive. The default value is 25.0. Higher values of shininess concentrate specular highlights into smaller and brighter areas, while lower values will spread out specular highlights over a wider area and make them appear dimmer.

Property Type	FLOAT		
Name String	Shininess		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SPECULAR

Set this property to a three-element vector [*red, green, blue*] to specify the color of the specular highlights of an object. The default is white or [255,255,255]. This property is used to simulate and control reflective highlights on an object. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Specular highlight color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

STYLE

An integer value that specifies how the polygon should be drawn:

- 0 = Points: Only vertices are drawn, using either COLOR or VERT_COLORS.
- 1 = Lines: Each polygon is outlined by connecting vertices.
- 2 = Filled (default): The polygon faces are shaded.

In a property sheet, this property appears as an enumerated list with the following options:

- Points
- Lines
- Filled

Property Type	ENUMLIST		
Name String	Style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTURE_COORD

A 2 by n array containing the texture map coordinates, where n is the number of polygon vertices. Each two-element entry in this array specifies the texture coordinates for the corresponding vertex in the vertex list. One texture coordinate pair should exist for each vertex. Use this property in conjunction with the TEXTURE_MAP property to wrap images over 2-D and 3-D polygons. Default coordinates are not provided.

Texture coordinates are normalized. This means that the $m \times n$ image object specified via the TEXTURE_MAP property is mapped into the range [0.0, 0.0] to [1.0, 1.0]. If texture coordinates outside the range [0.0, 0.0] to [1.0, 1.0] are specified, the image object is tiled into the larger range.

For example, suppose the image object specified via TEXTURE_MAP is a 256 x 256 array, and we want to map the image into a square two units on each side. To completely fill the square with a single copy of the image:

```
TEXTURE_COORD = [[0,0], [1,0], [1,1], [0,1]]
```

To fill the square with four tiled copies of the image:

```
TEXTURE_COORD = [[0,0], [2,0], [2,2], [0,2]]
```

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TEXTURE_INTERP

A Boolean value that indicates whether bilinear sampling is to be used for texture mapping an image onto the polygon(s). The default is nearest neighbor sampling.

In a property sheet, this property appears as an enumerated list with the following options:

- Nearest neighbor
- Bilinear

Property Type	ENUMLIST		
Name String	Texture interpolation		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTURE_MAP

An object reference to an IDLgrImage object to be texture mapped onto the polygons. The tiling or mapping of the texture is defined expressly by TEXTURE_COORD. If this property is omitted, polygons are filled with the color specified by the COLOR or VERT_COLORS property. If both TEXTURE_MAP and COLOR or VERT_COLORS properties exist, the color of the texture is modulated by the base color of the object. (This means that for the clearest display of the texture image, the COLOR property should be set equal to [255,255,255].) To remove a texture map, set TEXTURE_MAP equal to a null object reference.

Setting TEXTURE_MAP to the object reference of an IDLgrImage that contains an alpha channel allows you to create a transparent IDLgrPolygon object. If an alpha channel is present in the IDLgrImage object, IDL blends the texture using the blend function $\text{src}=\text{alpha}$ and $\text{dst}=1 - \text{alpha}$, which corresponds to a BLEND_FUNCTION of (3,4) as described for the IDLgrImage object.

If the width and/or height of the provided image is not an exact power of two, then the texture map will consist of the given image pixel values resampled to the nearest larger dimensions that are exact powers of two.

Note

Texture mapping is disabled when rendering to a destination object that uses Indexed color mode.

Note

Texture mapping is applied to all styles that are set by the STYLE property.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Texture map		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

THICK

A floating-point value between 1.0 and 10.0, specifying the size of the points or the thickness of the lines to be drawn when STYLE is set to either 0 (Points) or 1 (Lines), in points. The default is 1.0 points.

Note

The value of this property is ignored if STYLE is set to 2 (Filled).

In a property sheet, this property appears as follows:



Property Type	THICKNESS		
Name String	Thickness		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VERT_COLORS

A vector of colors to be used to specify the color of a polygon vertex. The vector may be of the form $[n]$ where each entry is a color index, or of the form $[3,n]$ where each 3-element row is an RGB color, or of the form $[4,n]$ where each 4-element row is an RGBA color. To remove vertex colors after they have been set, set VERT_COLORS to a scalar.

If VERT_COLORS is not specified, the entire polygon is drawn in the single color provided by the COLOR property, which is the default action.

If SHADING is set to 0 (Flat, the default) the color of a polygon is the color of the first vertex of that polygon. Color is interpolated between vertices if SHADING is set to 1 (Gouraud).

If there are more vertices than elements in VERT_COLORS, the elements of VERT_COLORS are cyclically repeated.

Note

If the polygon object is being rendered on a destination device that uses the Indexed color model, and the view that contains the polygon also contains one or more light objects, the VERT_COLORS property is ignored and the SHADE_RANGE property is used instead.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Vertex colors		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form $[xmin, xmax]$ that specifies the range of x data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is $[0.0, 1.0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZERO_OPACITY_SKIP

A Boolean value that makes it possible to gain finer control over the rendering of textured polygon pixels (texels) with an opacity of 0 in the texture map. Texels with zero opacity do not affect the color of a screen pixel since they have no opacity.

- 1 = Any texels are “skipped” and not rendered at all (the default).
- 0 = The Z-buffer is updated for these pixels and the display image is not affected as noted above.

By updating the Z-buffer without updating the display image, the polygon can be used as a *clipping* surface for other graphics primitives drawn after the current graphics object. The default value for this property is 1.

Note

This property has no effect if no texture map is used or if the texture map in use does not contain an opacity channel.

Property Type	BOOLEAN		
Name String	Skip zero opacity		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ZRANGE

A two-element floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

ALPHA_CHANNEL, AMBIENT, DIFFUSE, EMISSION, SHININESS, and SPECULAR properties added: 6.1

IDLgrPolygon::Cleanup

The IDLgrPolygon::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrPolygon::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPolygon::GetCTM

The IDLgrPolygon::GetCTM The IDLgrPolygon::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrPolygon::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the polygon object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrPolygonl::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrPolygon::GetProperty

The IDLgrPolygon::GetProperty procedure method retrieves the value of the property or group of properties for the polygons.

Syntax

Obj->[IDLgrPolygon::]GetProperty[, *PROPERTY=variable*]

Arguments

There are no arguments for this methods.

Keywords

Any property listed under “[IDLgrPolygon Properties](#)” on page 3846 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrPolygon::Init

The IDLgrPolygon::Init function method initializes the polygons object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrPolygon' [, X [, Y[, Z]]] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrPolygon::]Init([X, [Y, [Z]]] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

X

A vector argument providing the X coordinates of the vertices. The vector must contain at least three elements. If the Y and Z arguments are not provided, X must be an array of either two or three vectors (i.e., [2,*] or [3,*]), in which case, X[0,*] specifies the X values, X[1,*] specifies the Y values, and X[2,*] specifies the Z values.

This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is converted to single precision floating point.

Y

A vector argument providing the Y coordinates of the vertices. The vector must contain at least three elements. This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is converted to single precision floating point.

Z

A vector argument providing the Z coordinates of the vertices. The vector must contain at least three elements. This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is converted to single precision floating point.

Keywords

Any property listed under [“IDLgrPolygon Properties”](#) on page 3846 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES keyword added: 5.6

IDLgrPolygon:: SetProperty

The IDLgrPolygon::SetProperty procedure method sets the value of the property or group of properties for the polygons.

Syntax

Obj->[IDLgrPolygon::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrPolygon Properties](#)” on page 3846 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrPolyline

A polyline object represents one or more polylines that share a set of vertices and rendering attributes.

An IDLgrPolyline object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrPolyline::Init](#)” on page 3899.

Properties

Objects of this class have the following properties. See “[IDLgrPolyline Properties](#)” on page 3877 for details on individual properties.

- [ALL](#)
- [CLIP_PLANES](#)
- [DATA](#)
- [DEPTH_TEST_FUNCTION](#)
- [DOUBLE](#)
- [LABEL_NOGAPS](#)
- [LABEL_OBJECTS](#)
- [LABEL_USE_VERTEX_COLOR](#)
- [PALETTE](#)
- [POLYLINES](#)
- [RESET_DATA](#)
- [SHARE_DATA](#)
- [THICK](#)
- [ALPHA_CHANNEL](#)
- [COLOR](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_WRITE_DISABLE](#)
- [HIDE](#)
- [LABEL_OFFSETS](#)
- [LABEL_POLYLINES](#)
- [LINESTYLE](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [SHADING](#)
- [SYMBOL](#)
- [USE_LABEL_COLOR](#)

- `USE_LABEL_ORIENTATION`
- `USE_TEXT_ALIGNMENTS`
- `VERT_COLORS`
- `XCOORD_CONV`
- `XRANGE`
- `YCOORD_CONV`
- `YRANGE`
- `ZCOORD_CONV`
- `ZRANGE`

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrPolyline::Cleanup`
- `IDLgrPolyline::GetCTM`
- `IDLgrPolyline::GetProperty`
- `IDLgrPolyline::Init`
- `IDLgrPolyline::SetProperty`

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrPolyline Properties

IDLgrPolyline objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrPolyline::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrPolyline::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrPolyline::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

A floating-point value in the range [0.0, 1.0] to specify the opacity of the polyline. The default value of 1.0 causes IDL to draw the line completely opaque. If the value of this property is less than 1.0, then the pixels of the polyline are blended with the pixels already on the screen, where the color of the polyline is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha.

Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results. Note that since a single polyline object may contain many polylines, or because a line may draw in back or in front of itself, the ordering of the polylines within the object (POLYLINES property) must be considered as well.

This property has no effect on devices using indexed color mode.

If 3-channel (RGB) VERT_COLORS are supplied, then the value of this property is used for the alpha at each vertex to draw the line. If 4-channel (RGBA) VERT_COLORS are supplied, then the alpha from each vertex color is used for the alpha at each vertex to draw the line.

When a polyline is drawn with symbols, the symbol's color and alpha values are always used to draw the symbols unless VERT_COLORS are supplied. If the VERT_COLORS are RGB, then the alpha value of the symbol is used to draw the symbol. If the VERT_COLORS are RGBA, then the alpha from the vertex color is used to draw the symbol.

When used with labels, the value of this property is used as the alpha channel value for a label if its corresponding value in the USE_LABEL_COLOR property is zero and the value of LABEL_USE_VERTEX_COLOR is zero.

If a label's corresponding value in the USE_LABEL_COLOR property is zero and the LABEL_USE_VERTEX_COLOR property is non-zero, then the label is drawn with the alpha value from its corresponding vertex color, if the alpha values are included in the vertex colors. If the vertex colors do not include alpha values, the value of this property is used as the alpha value for the label.

If a label's corresponding value in the USE_LABEL_COLOR property is non-zero, then the value of the label object's ALPHA_CHANNEL property is used to draw the label. In other words, USE_LABEL_COLOR also implies using the label object's alpha as well as its color.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see [“Alpha Channel and Objects”](#) in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by- N floating-point array of dimensions that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$. Portions of this

object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any x , y , or z coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String			
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

A color index or a three-element vector [*red*, *green*, *blue*] specifying the color used to draw polylines. The default color is [0, 0, 0] (black). This property is ignored if the VERT_COLORS property is provided.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DATA

A 2-by- n or a 3-by- n floating-point array which defines, respectively, the 2-D or 3-D vertex data. DATA is equivalent to the optional arguments, X , Y , and Z . This property

is converted to double-precision floating-point values if the DOUBLE property is set. Otherwise, it is converted to single-precision floating-point.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.

- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DOUBLE

A Boolean value that indicates whether data provided by any of the input arguments will be stored in this object as using double-precision floating-point format.

- Set this property equal to 1 to convert input data to double-precision floating-point format.
- Set this property equal to 0 to convert input data to single-precision floating-point format.
- If you do not specify a value for this property, no data type conversion will be performed, and the data will be stored with its original precision.

Setting this property may be desirable if the data consists of large integers that cannot be accurately represented in single-precision floating-point arithmetic. This property is also automatically set to 1 if any of the input arguments are stored using a variable of type DOUBLE.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LABEL_NOGAPS

An integer vector of values indicating whether gaps should be computed for the corresponding label. A zero value indicates that a gap will be computed for the labels; a non-zero value indicates that no gap will be computed for the label. If the number of labels exceeds the number of elements in this vector, the LABEL_NOGAPS values will be repeated cyclically. By default, gaps are computed for all labels (so that the polyline does not pass through the label).

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LABEL_OFFSETS

A scalar or vector of floating-point offsets, [t0, t1, ...], that indicate the parametric offsets along the length of each polyline (specified via the LABEL_POLYLINES property) at which each label (as specified via the LABEL_OBJECTS property) would be positioned. If LABEL_OFFSETS is set to a scalar less than zero, then the offsets will be automatically computed to be evenly distributed along the length of the polyline. If a scalar value greater than or equal to zero is provided, it is used for all labels. If a vector is provided, the number of offsets must match the number of labels provided via LABEL_OBJECTS. By default, this property is set to the scalar, -1, indicating that the label offsets will be automatically computed.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LABEL_OBJECTS

An object reference (or vector of object references) to specify the labels to be drawn along the polyline path(s). The objects specified via this property must inherit from one of the following classes:

- IDLgrSymbol
- IDLgrText

If a single object is provided, and it is an IDLgrText object, each of its strings will correspond to a label. If a vector of objects is used, any IDLgrText objects should have only a single string; each object will correspond to a label.

If one or more IDLgrText objects are provided, the LOCATION property of the provided text object(s) may be overwritten; position is determined according to the values provided via the LABEL_OFFSETS property. The labels will have the same color as the corresponding polyline (see the COLOR property) unless the USE_LABEL_COLOR property is specified. The orientation of the label objects USE_LABEL_ORIENTATION property is specified. The horizontal and vertical alignment for any text labels will each default to 0.5 (i.e., centered) unless the USE_TEXT_ALIGNMENTS property is specified.

Note

The objects provided via this property will not be destroyed automatically when this IDLgrPolyline is destroyed.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LABEL_POLYLINES

An integer or integer vector of polyline indices, [P0, P1, ...], that indicate which polylines are to be labeled. Pi corresponds to the ith polyline specified via the POLYLINES property. This property is intended to be used in conjunction with the LABEL_OBJECTS property. If a scalar is provided, all labels will be drawn along the single indicated polyline. If a vector is provided, the number of polyline indices must match the number of labels provided via LABEL_OBJECTS.

By default, this property is set to the scalar, 0, indicating that only the first polyline will be labeled.

Note

If a given polyline has more than one label, then the corresponding polyline index may appear more than once in the LABEL_POLYLINES vector.

Property Type	Integer or integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LABEL_USE_VERTEX_COLOR

An integer value that indicates whether labels should be colored according to the vertex coloring (if the VERT_COLORS property is set).

- A non-zero value = labels should be colored according to the vertex coloring
- Zero (the default) = the label will be drawn using the color specified via the COLOR property of the polyline object (unless the USE_LABEL_COLOR property is set).

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LINestyle

An integer or array value that indicates the line style to be used to draw the polyline. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINestyle property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot

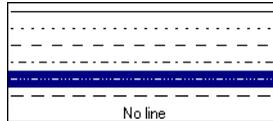
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq repeat \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

In a property sheet, this property appears as follows:



Property Type	LINESTYLE		
Name String	Line style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the `IDLgrPalette` object class) that defines the color palette of this object. This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the `PALETTE` property on this object is not set, the destination object `PALETTE` property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

POLYLINES

An integer array of polyline descriptions. A polyline description is an integer or long word array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polyline, and $i_0..i_{n-1}$ are indices into the X, Y, and Z arguments that represent the vertices of the polyline(s). To ignore an entry in the POLYLINES array, set the vertex count, n , and all associated indices to 0. To end the drawing list, even if additional array space is available, set n to -1. If this property is not specified, a single connected polyline will be generated from the X, Y, and Z arguments.

Note

Setting the POLYLINES property to a scalar value will reset the property to its initial, undefined state.

Note

The connectivity array described by POLYLINES allows an individual object to contain more than one polyline. Vertex, normal and color information can be shared by the multiple polylines. Consequently, the polyline object can represent an entire mesh and compute reasonable normal estimates in most cases.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the

object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESET_DATA

A Boolean value that determines whether to treat the data provided via one of the DATA property as a new data set unique to this object, rather than overwriting data that is shared by other objects. There is no reason to use this property if the object on which the property is being set does not currently share data with another object (that is, if the SHARE_DATA property is not in use). This property has no effect if no new data is provided via the DATA property. .

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHADING

An integer representing the type of shading to use:

- 0 = Flat (default): The color of the second vertex in a line segment is used to define the color for the entire line segment. The color has a constant intensity based upon the normal vector.
- 1 = Gouraud: The colors along each line are interpolated between vertex colors.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

In a property sheet, this property appears as an enumerated list with the following options:

- Flat
- Gouraud

Property Type	ENUMLIST		
Name String	Shading		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SHARE_DATA

An object reference to an object whose data is to be shared by this polyline. A polyline may only share data with a polygon object or another polyline. The `SHARE_DATA` property is intended for use when data values are not set via an argument to the object's `Init` method or by setting the object's `DATA` property.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SYMBOL

An object reference vector containing one or more instances of the [IDLgrSymbol](#) object class to indicate the plotting symbols to be used at each vertex of the polyline. If there are more vertices than elements in `SYMBOL`, the elements of the `SYMBOL` vector are cyclically repeated. By default, no symbols are drawn. To remove symbols from a polyline, set `SYMBOL` to a scalar.

Property Type	Object reference vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0, specifying the line thickness to be used to draw the polyline, in points. The default is 1.0 points.

In a property sheet, this property appears as follows:



Property Type	THICKNESS		
Name String	Thickness		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

USE_LABEL_COLOR

An integer vector of values to indicate whether the COLOR property value for the corresponding label object is to be used to draw that label. If the number of labels exceeds the number of elements in this vector, the USE_LABEL_COLOR values will be repeated cyclically. By default, this value is zero, indicating that the COLOR property of each label object will be ignored, and the COLOR property for the polyline object will be used instead.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

USE_LABEL_ORIENTATION

An integer vector of values to indicate whether the orientation of the corresponding label object is to be used to draw that label. For IDLgrText objects, this refers to the BASELINE and UPDIR property values. For IDLgrSymbol objects, this refers to the default (un-rotated) orientation of the symbol. If the number of labels exceeds the number of elements in this vector, the USE_LABEL_ORIENTATION values will be repeated cyclically. By default, USE_LABEL_ORIENTATION is zero, indicating

that the orientation will be automatically computed so that the baseline is parallel to the polyline, and the updir is perpendicular to the polyline. .

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

USE_TEXT_ALIGNMENTS

A Boolean value that indicates whether, for any IDLgrText labels (as specified via the LABEL_OBJECTS property), the ALIGNMENT and VERTICAL_ALIGNMENT property values for the given IDLgrText object(s) are to be used to draw those labels. By default, this value is zero, indicating that the ALIGNMENT and VERTICAL_ALIGNMENT properties of the IDLgrText object(s) will be overwritten with default values (0.5 for each, indicating centered labels).

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

VERT_COLORS

A vector of colors to be used to specify the color of the line segment between each vertex. The vector may be of the form $[n]$ where each entry is a color index, or of the form $[3,n]$ where each 3-element row is an RGB color, or of the form $[4,n]$ where each 4-element row is an RGBA color. To remove vertex colors after they have been set, set VERT_COLORS to a scalar.

If VERT_COLORS is not specified, the entire polyline is drawn in the single color provided by the COLOR property, which is the default action.

If SHADING is set to 0 (Flat, the default) the color of a line segment is the color of the second vertex of that line segment. Color is interpolated between vertices if SHADING is set to 1 (Gouraud).

If there are more vertices than elements in VERT_COLORS, the elements of VERT_COLORS are cyclically repeated.

When VERT_COLORS is used with symbols, the vertex colors and vertex alpha override any colors and alpha specified by the symbol object or any colors and alpha specified by graphic atoms contained in a user-defined symbol. When

VERT_COLORS is used with labels, the application of the color and alpha of the labels is controlled by the USE_LABEL_COLOR and LABEL_USE_VERTEX_COLOR properties.

Property Type	USERDEF		
Name String	Vertex colors		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form $[xmin, xmax]$ that specifies the range of x data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Y = s_0 + s_1 * \text{Data}Y$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form [*ymin*, *ymax*] that specifies the range of y data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Z = s_0 + s_1 * \text{Data}Z$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

ALPHA_CHANNEL added: 6.1

IDLgrPolyline::Cleanup

The IDLgrPolyline::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrPolyline::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPolyline::GetCTM

The IDLgrPolyline::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrPolyline::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the polyline object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrPolyline::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrPolyline::GetProperty

The IDLgrPolyline::GetProperty procedure method retrieves the value of a property or group of properties for the polylines.

Syntax

Obj->[IDLgrPolyline::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrPolyline Properties](#)” on page 3877 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrPolyline::Init

The IDLgrPolyline::Init function method initializes the polylines object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrPolyline' [, X [, Y[, Z]]] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrPolyline::]Init([X, [Y, [Z]]] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

X

A vector providing the X components of the points to be connected. If the Y and Z arguments are not provided, X must be an array of either two or three vectors (i.e., [2,*] or [3,*]), in which case, X[0,*] specifies the X values, X[1,*] specifies the Y values, and X[2,*] specifies the Z values. This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Y

A vector providing the Y coordinates of the points to be connected. This argument is stored as double precision floating point values if the argument variable is of type

DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Z

A vector providing the Z coordinates of the points to be connected. This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Keywords

Any property listed under “[IDLgrPolyline Properties](#)” on page 3877 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES, LABEL_NOGAPS, LABEL_POLYLINES, LABEL_OFFSETS, LABEL_OBJECTS, LABEL_USE_VERTEX_COLOR, USE_LABEL_COLOR, USE_LABEL_ORIENTATION, USE_TEXT_ALIGNMENTS keywords added: 5.6

IDLgrPolyline:: SetProperty

The IDLgrPolyline::SetProperty procedure method sets the value of a property or group of properties for the polylines.

Syntax

Obj->[IDLgrPolyline::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrPolyline Properties](#)” on page 3877 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrPrinter

A printer object represents a hardcopy graphics destination. When a printer object is created, the printer device to which it refers is the default system printer. To change the printer, utilize the printer dialogs (see “[DIALOG_PRINTJOB](#)” on page 559 and “[DIALOG_PRINTERSETUP](#)” on page 557.)

Note

Objects or subclasses of this type can not be saved or restored.

Superclasses

This class has no superclass.

Creation

See “[IDLgrPrinter::Init](#)” on page 3922.

Properties

Objects of this class have the following properties. See “[IDLgrPrinter Properties](#)” on page 3904 for details on individual properties.

- [ALL](#)
- [COLOR_MODEL](#)
- [DIMENSIONS](#)
- [GAMMA](#)
- [GRAPHICS_TREE](#)
- [LANDSCAPE](#)
- [N_COLORS](#)
- [N_COPIES](#)
- [PALETTE](#)
- [PRINT_QUALITY](#)
- [QUALITY](#)
- [REGISTER_PROPERTIES](#)

- [RESOLUTION](#)
- [UNITS](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrPrinter::Cleanup](#)
- [IDLgrPrinter::Draw](#)
- [IDLgrPrinter::GetContiguousPixels](#)
- [IDLgrPrinter::GetFontnames](#)
- [IDLgrPrinter::GetProperty](#)
- [IDLgrPrinter::GetTextDimensions](#)
- [IDLgrPrinter::Init](#)
- [IDLgrPrinter::NewDocument](#)
- [IDLgrPrinter::NewPage](#)
- [IDLgrPrinter::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Examples

See [Chapter 34, “Using Destination Objects”](#) in the *Using IDL* manual for information on the IDLgrPrinter object. See [“Positioning Objects Within a Page”](#) in Chapter 34 of the *Using IDL* manual for examples that define the locations of objects within the hardcopy page.

Version History

Introduced: 5.0

IDLgrPrinter Properties

IDLgrPrinter objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrPrinter::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrPrinter::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrPrinter::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR_MODEL

An integer value that determines the color model to be used for the buffer:

- 0 = RGB (default)
- 1 = Color Index

In a property sheet, this property appears as an enumerated list with the following options:

- RGB
- Indexed

Property Type	ENUMLIST		
Name String	Color model		
Get: Yes	Set: No	Init: Yes	Registered: Yes

DIMENSIONS

A two-element integer vector of the form [*width, height*] specifying the overall ‘drawable’ area that may be printed on a page. By default, the dimensions are measured in device units (refer to the [UNITS](#) property).

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

GAMMA

An object reference to an IDLgrPalette object whose entries will be used as the gamma correction table for color printing. Gamma correction only applies if COLOR_MODEL=1 (Indexed).

When the color palette (specified via the PALETTE property) is loaded for the printer, if a gamma correction table is provided, then each of the [R,G,B] entries in the color palette will be translated through the gamma correction table. For example, at color index *i*:

```
correctedRed[i] = gammaRed[paletteRed[i]]
correctedGreen[i] = gammaGreen[paletteGreen[i]]
correctedBlue[i] = gammaBlue[paletteBlue[i]]
```

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GRAPHICS_TREE

An object reference of type IDLgrScene, IDLgrViewgroup, or IDLgrView that specifies the graphics tree of this object. If this property is set to a valid object reference, calling the Draw method on the destination object with no arguments will

cause the object reference associated with this property to be drawn. If this object is valid and the destination object is destroyed, this object reference will be destroyed as well. By default the GRAPHICS_TREE property is set equal to the null-object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

LANDSCAPE

A Boolean value that indicates whether to produce hardcopy output in landscape mode. The default value of zero indicates Portrait mode.

Note

The printer driver may not support the LANDSCAPE option; in general, it is best to use the printer dialogs to set orientation.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_COLORS

An integer value that indicates the number of colors (between 2 and 256) to be used if the COLOR_MODEL is set to Indexed (1). This property is ignored if the COLOR_MODEL is set to RGB (0).

Property Type	INTEGER		
Name String	Number of colors		
Get: Yes	Set: No	Init: Yes	Registered: Yes

N_COPIES

An integer that determines the number of copies of print data to be generated. The default is 1 copy.

Note

Your specific printer driver may not support the N_COPIES option. You can also use the printer dialogs to set the number of copies.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

NAME

A string containing the operating system-specific name of the print stream. e.g.
'\\BORG\HpJet '.

Property Type	STRING		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

PALETTE

An object reference to a palette object (an instance of the [IDLgrPalette](#) object class) to specify the red, green, and blue values that are to be loaded into the graphics destination's color lookup table if the Indexed color model is used.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PRINT_QUALITY

An integer value indicating the print quality at which graphics are to be drawn to the printer. Note that the print quality is independent of the rendering quality (as set by the QUALITY property). Valid values are:

- 0 = Low

- 1 = Normal (this is the default)
- 2 = High

Generally, setting the print quality to a lower value will increase the speed of the printing job, but decrease the resolution; setting it to a higher value will cause the printing job to take more time, but will increase the resolution.

Note

Some printer drivers may not be able to support different printing qualities. In these cases, the setting of the PRINT_QUALITY property will be quietly ignored.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

QUALITY

An integer value that indicates the rendering quality at which graphics are to be drawn to this destination. Note that the rendering quality is independent of the print quality (as set by the PRINT_QUALITY property). Valid values are:

- 0 = Low
- 1 = Medium
- 2 = High (default)

In a property sheet, this property appears as an enumerated list with the following options:

- Low
- Medium
- High

Property Type	ENUMLIST		
Name String	Draw quality		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESOLUTION

A floating-point vector of the form [*xres*, *yres*] defining the pixel resolution, measured in centimeters per pixel. This value is stored in double precision.

Property Type	FLOAT		
Name String	Resolution		
Get: Yes	Set: No	Init: No	Registered: Yes

UNITS

An integer value that indicates the units of measure for the DIMENSIONS property. Valid values are:

- 0 = Device (default)
- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to the drawable area on a page.

Note

If you change the value of the UNITS property (using the SetProperty method), IDL will convert the current value of the DIMENSIONS property to the new units.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IDLgrPrinter::Cleanup

The IDLgrPrinter::Cleanup procedure method performs all cleanup on the object. If a document is open (that is, if graphics have been draw to the printer), the document is closed and the pending graphics are sent to the current printer.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrPrinter:]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPrinter::Draw

The IDLgrPrinter::Draw procedure method draws the given picture to this graphics destination.

Printing in Bitmap or Vector Graphic Mode

The IDLgrPrinter::Draw method VECTOR keyword generates output in bitmap or vector format. The following table shows the keyword options and results for each platform.

Keyword Settings	Windows Printer Output	UNIX File Type
VECTOR = 0	Bitmap (BMP)	Encapsulated PostScript (EPS) file (e.g. xprinter.eps)
VECTOR = 1	Enhanced MetaFile (EMF)	Encapsulated PostScript (EPS) file (e.g. xprinter.eps)

Table 8-9: File Types Produced by IDLgrPrinter Draw Method

Because Windows printer output is usually sent directly to the printer, EMF and BMP files are not viewable. On UNIX, the printer output is usually directed to a file named xprinter.eps by default.

Note

Objects are drawn to the destination device in the order that they are added to the model, view, viewgroup, or scene object that contains them when VECT_SORTING=0. Otherwise, objects are drawn to the destination device based on their average depth value. See “[Bitmap and Vector Graphic Output](#)” in Chapter 34 of the *Using IDL* manual for more information on bitmap and vector output.

Syntax

```
Obj->[IDLgrPrinter::]Draw [, Picture] [, VECT_SORTING={ 0 | 1 } ]
    [, VECT_TEXT_RENDER_METHOD={ 0 | 1 } ] [, VECTOR={ 0 | 1 } ]
```

Arguments

Picture

The view (an instance of an [IDLgrView](#) object), viewgroup (an instance of an [IDLgrViewgroup](#) object), or scene (an instance of an [IDLgrScene](#) object) to be drawn.

Keywords

VECT_SORTING

This keyword has an effect only when generating vector output (VECTOR=1).

This keyword controls the way object primitives in a picture appear in the destination. Valid values are:

- 0 = disable sorting. This causes the object primitives to appear in the vector output file in the same order they are drawn on the display device. This is the order in which they appear in the graphics tree.
- 1 = enable sorting (default). Objects are ordered from back to front based on each primitive object's average depth value.

Note

Objects that intersect each other, [IDLgrImage](#) objects (which do not update the depth buffer), and objects contained within transparent views may not sort correctly. See “[Controlling What is Displayed in Vector Graphics](#)” in Chapter 34 of the *Using IDL* manual for details.

VECT_TEXT_RENDER_METHOD

This keyword has an effect only when generating vector output (VECTOR=1).

This keyword controls the way text is rendered in a vector graphic file. Valid values are:

- 0 = render text as text primitives (default). Use the output device's text primitives when rendering text.
- 1 = render text as primitives composed of triangles. Render text in the vector file as a series of filled triangles. This produces text that closely matches the text on the display device, but the output file size is larger.

Note

When using the IDLgrPrinter object under UNIX, the Xprinter output is regarded as write-only. As there is no support for 3-D text, IDL always generates filled triangles when rendering text in the Xprinter output. On Windows, the vector output is always Enhanced MetaFile (EMF) file format. Because printer output is usually sent directly to the printer the EMF file cannot be seen by the user.

Note

See [“Text Rendering in Vector Graphics”](#) in Chapter 34 of the *Using IDL* manual for more information.

VECTOR

Set this keyword to indicate the type of graphics primitives generated. Valid values are:

- 0 = generate bitmap graphics (default). Render the scene to a buffer and then copy the buffer to the printer in bitmap format. The bitmap retains the quality of the original image.
- 1 = generate vector graphics. Render the scene using simple vector operations that result in a representation of the scene that is scalable to the printer. The vector representation, which is sent to the printer, does not retain all the attributes of the original image.

Note

Transparent objects in a view are not rendered in vector graphic files. Semi-transparent objects are rendered fully opaque. See [“Bitmap and Vector Graphic Output”](#) in Chapter 34 of the *Using IDL* manual for a complete discussion of when to output to bitmap or vector graphics based on picture content.

Examples

This example demonstrates the process of printing the contents of an IDL graphics display object (a buffer or a window) to an IDLgrPrinter object. The resolution of the printed page is based on the resolution of the screen. The model object in the printer object must be scaled to maintain the same size as displayed on the screen. The location of the view must also be changed to center the display on the page.

```
PRO PrintingAnImage

; Determine the path to the "convec.dat" file.
convecFile = FILEPATH('convec.dat', $
```

```

SUBDIRECTORY = ['examples', 'data'])

; Initialize the parameters of the image with the file.
convecSize = [248, 248]
convecImage = BYTARR(convecSize[0], convecSize[1])

; Open the file, read in the image, and then close the
; file.
OPENR, unit, convecFile, /GET_LUN
READU, unit, convecImage
FREE_LUN, unit

; Initialize the display objects.
windowSize = convecSize
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'Earth Mantle Convection')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., windowSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image object with its palette.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette->LOADCT, 27
oImage = OBJ_NEW('IDLgrImage', convecImage, $
    PALETTE = oPalette)

; Add image to model, which is added to the view, and
; then the view is displayed in the window.
oModel->Add, oImage
oView->Add, oModel
oWindow->Draw, oView

; Determine the centimeter measurements of the image
; on the screen.
oWindow->GetProperty, RESOLUTION = screenResolution
windowSizeCM = windowSize*screenResolution

; Initialize printer destination object.
oPrinter = OBJ_NEW('IDLgrPrinter', PRINT_QUALITY = 2, $
    QUALITY = 2)

; Obtain page parameters to determine the page
; size in centimeters.
oPrinter->GetProperty, DIMENSIONS = pageSize, $
    RESOLUTION = pageResolution
pageSizeCM = pageSize*pageResolution

; Calculate a ratio between screen size and page size.

```

```
pageScale = windowSizeCM/pageSizeCM

; Use ratio to scale the model within the printer to the
; same size as the model on the screen.
oModel->Scale, pageScale[0], pageScale[1], 1.

; Determine the center of the page and the image in
; pixels.
centering = ((pageSizeCM - windowSizeCM)/2.) $
            /pageResolution

; Move the view to center the image.
oView->SetProperty, LOCATION = centering

; Display the view within the printer destination.
oPrinter->Draw, oView
oPrinter->NewDocument

; Cleanup object references.
OBJ_DESTROY, [oPrinter, oView, oPalette]

END
```

Version History

Introduced: 5.0

VECT_SORTING, VECT_TEXT_RENDER_METHOD keywords added: 6.1

IDLgrPrinter::GetContiguousPixels

The IDLgrPrinter::GetContiguousPixels function method returns an array of long integers whose length is equal to the number of colors available in the index color mode (that is, the value of the N_COLORS property).

The returned array marks contiguous pixels with the ranking of the range's size. This means that within the array, the elements in the largest available range are set to zero, the elements in the second-largest range are set to one, etc. Use this range to set an appropriate colormap for use with the SHADE_RANGE property of the IDLgrSurface and IDLgrPolygon object classes.

To get the largest contiguous range, you could use the following IDL command:

```
result = obj->GetContiguousPixels()  
Range0 = WHERE(result EQ 0)
```

A contiguous region in the colormap can be increasing or decreasing in values. The following would be considered contiguous:

```
[0,1,2,3,4]  
[4,3,2,1,0]
```

Syntax

Result = *Obj*->[IDLgrPrinter::]GetContiguousPixels()

Return Value

Returns an array of long integers whose length is equal to the number of colors available in the index color mode.

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPrinter::GetFontnames

The `IDLgrPrinter::GetFontnames` function method returns the list of available fonts that can be used in `IDLgrFont` objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned; see [Appendix H, “Fonts”](#) for more information.

Syntax

```
Result = Obj->[IDLgrPrinter::]GetFontnames( FamilyName [, IDL_FONTS={0 | 1 | 2}] [, STYLES=string] )
```

Return Value

Returns the list of available fonts that can be used in `IDLgrFont` objects.

Arguments

FamilyName

A string representing the name of the font family to which all of the returned fonts must belong. The string may be a fully specified family name—such as “Helvetica”. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. To return all available family names, use “*”.

Keywords

IDL_FONTS

Set this keyword to specify where to search for fonts that IDL may use. Set `IDL_FONT` to 1 to select only fonts installed by IDL and to 2 to select only fonts detected in the host operating system. The default value is 0, specifying that both IDL and operating system fonts should be returned.

STYLES

Set this keyword to a string specifying the styles that are to be matched by the returned font names. You can set `STYLES` to a fully specified style string, such as “Bold Italic”. If you set `STYLES` to the null string, ‘’, only fontnames without style modifiers will be returned. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. The default value is

the string, “*”, which returns all fontnames containing the *FamilyName* argument, with or without style modifiers.

Version History

Introduced: 5.0

IDLgrPrinter::GetProperty

The IDLgrPrinter::GetProperty procedure method retrieves the value of a property or group of properties for the printer.

Syntax

Obj->[IDLgrPrinter::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrPrinter Properties](#)” on page 3904 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrPrinter::GetTextDimensions

The IDLgrPrinter::GetTextDimensions function method retrieves the dimensions of a text or axis object that will be rendered in a window. The result is a 3-element double-precision floating-point vector [$xDim$, $yDim$, $zDim$] representing the dimensions of the text or axis object, measured in data units. If the object specified is an axis object, the result encompasses the tick labels and the title of the axis (if any).

Syntax

```
Result = Obj->[IDLgrPrinter::]GetTextDimensions( TextObj [, DESCENT=variable]  
[, PATH=objref(s)] )
```

Return Value

Returns a 3-element double-precision floating-point vector [$xDim$, $yDim$, $zDim$] representing the dimensions of the text or axis object, measured in data units.

Arguments

TextObj

The object reference to a text or axis object for which the text dimensions are requested.

Keywords

DESCENT

Set this keyword equal to a named variable that will contain an array of double-precision floating-point values (one for each string in the IDLgrText object). The values represent the distance to travel (parallel to the UPDIR vector) from the text baseline to reach the bottom of the lowest descender in the string. All values will be negative numbers, or zero. This keyword is valid only if *TextObj* is an IDLgrText object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the text dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this

keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If `IDLgrPrinter::GetTextDimensions` is called from within a Draw method and the PATH keyword is not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

Version History

Introduced: 5.0

IDLgrPrinter::Init

The IDLgrPrinter::Init function method initializes the printer object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLgrPrinter' [, *PROPERTY=value*])

or

Result = *Obj*->[IDLgrPrinter::]Init([*PROPERTY=value*])
(Only in a subclass' Init method.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrPrinter Properties](#)” on page 3904 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrPrinter::NewDocument

The IDLgrPrinter::NewDocument procedure method closes the current document (a page or group of pages), which causes any pending output to be sent to the printer, finishing the printer job.

Syntax

Obj->[IDLgrPrinter::]NewDocument

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPrinter::NewPage

The IDLgrPrinter::NewPage procedure method issues a new page command to the printer.

Syntax

Obj->[IDLgrPrinter::]NewPage

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrPrinter:: SetProperty

The IDLgrPrinter::SetProperty procedure method sets the value of a property or group of properties for the printer.

Syntax

Obj->[IDLgrPrinter::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrPrinter Properties](#)” on page 3904 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrROI

The IDLgrROI object class is an object graphics representation of a region of interest.

Superclasses

[IDLanROI](#)

[IDLitComponent](#)

Creation

See “[IDLgrROI::Init](#)” on page 3941.

Properties

Objects of this class have the following properties. See “[IDLgrROI Properties](#)” on page 3929 for details on individual properties.

- [ALL](#)
- [ALPHA_CHANNEL](#)
- [CLIP_PLANES](#)
- [COLOR](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_TEST_FUNCTION](#)
- [DEPTH_WRITE_DISABLE](#)
- [DOUBLE](#)
- [HIDE](#)
- [LINESTYLE](#)
- [PALETTE](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [STYLE](#)
- [SYMBOL](#)
- [THICK](#)

- [XCOORD_CONV](#)
- [XRANGE](#)
- [YCOORD_CONV](#)
- [YRANGE](#)
- [ZCOORD_CONV](#)
- [ZRANGE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

The IDLgrROI object class has the following methods:

- [IDLgrROI::Cleanup](#)
- [IDLgrROI::GetProperty](#)
- [IDLgrROI::Init](#)
- [IDLgrROI::PickVertex](#)
- [IDLgrROI::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.3

IDLgrROI Properties

IDLgrROI objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrROI::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrROI::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrROI::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure with the values of all of the properties associated with the state of this object. State information about the object may include things like color, line style, etc., but not vertex data or user values.

Note

The fields in this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

A floating-point value in the range [0.0, 1.0] to specify the opacity of the ROI. The default value of 1.0 causes IDL to draw the ROI completely opaque. If the value of this property is less than 1.0, then the pixels of the ROI are blended with the pixels already on the screen, where the color of the ROI is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha.

Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results.

This property has no effect on devices using indexed color mode.

When the ROI is drawn with symbols, the symbol color and alpha values are always used to draw the symbols.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see “[Alpha Channel and Objects](#)” in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by-*N* floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any *x*, *y*, or *z* coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

A color index or a three-element vector [*red*, *green*, *blue*] specifying the color used to draw the region. The default color is [0, 0, 0].

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DOUBLE

A Boolean value that indicates whether data provided by any of the input arguments will be stored in this object as using double-precision floating-point format.

- Set this property equal to 1 to convert input data to double-precision floating-point format.
- Set this property equal to 0 to convert input data to single-precision floating-point format.
- If you do not specify a value for this property, no data type conversion will be performed, and the data will be stored with its original precision.

Setting this property may be desirable if the data consists of large integers that cannot be accurately represented in single-precision floating-point arithmetic. This property is also automatically set to 1 if any of the input arguments are stored using a variable of type DOUBLE.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value indicating whether this region should be drawn:

- 0 = draw the region (the default)
- 1 = do not draw the region

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LINestyle

An integer that indicates the line style to be used to draw the region. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern. The valid values for the pre-defined line styles are:

- 0 = solid (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

Property Type	ENUMLIST		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the [IDLgrPalette](#) object class) that defines the color palette of this object. This property is only used for Object Graphics destinations using the RGB color model. In this case, if the color value for the region is specified as a color index value, this palette is used to look up the color for the region. If the PALETTE property is not set, the destination object PALETTE property is used, which defaults to a gray scale ramp.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

STYLE

An integer value that indicates the geometrical primitive to use to represent the region when displayed. Valid values include:

- 0 = points
- 1 = open polyline
- 2 = closed polyline (the default)

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SYMBOL

An object reference to an [IDLgrSymbol](#) object for the symbol used for display when STYLE = 0 (points). By default, a dot is used.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0, specifying the size of the points, or the thickness of the lines, measured in points. The default is 1.0 points.

In a property sheet, this property appears as follows:



Property Type	THICKNESS		
Name String	Thickness		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max} - X_{min}), 1.0/(X_{max} - X_{min})]$$

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Double-precision floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form $[xmin, xmax]$ that specifies the range of x data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Y = s_0 + s_1 * \text{Data}Y$$

Recommended values are:

$$[(-Y_{min})/(Y_{max} - Y_{min}), 1.0/(Y_{max} - Y_{min})]$$

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max} - Z_{min}), 1.0/(Z_{max} - Z_{min})]$$

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

ALPHA_CHANNEL added: 6.1

IDLgrROI::Cleanup

The IDLgrROI::Cleanup procedure method performs all cleanup for a region of interest object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrROI::]Cleanup (In a subclass' Cleanup method only.)

Arguments

None

Keywords

None

Version History

Introduced: 5.3

IDLgrROI::GetProperty

The IDLgrROI::GetProperty procedure method retrieves the value of a property or group of properties for the Object Graphics region.

Syntax

Obj->[IDLgrROI::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “IDLgrROI Properties” on page 3929 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.3

IDLgrROI::Init

The IDLgrROI::Init function method initializes an Object Graphics region of interest.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW( 'IDLgrROI' [, X[, Y[, Z]]] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrROI::]Init([X[, Y[, Z]]] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

X

A vector providing the X components of the vertices for the region. If the Y and Z arguments are not specified, X must be a two-dimensional array with the leading dimension either 2 or 3 ([2, *] or [3, *]), in which case, X[0, *] represents the X values, X[1, *] represents the Y values, and X[2, *] represents the Z values. This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero. Otherwise it is converted and stored as single precision floating point.

Y

A vector providing the Y components of the vertices. This argument is stored as double precision floating point values if the argument variable is of type `DOUBLE` or if the `DOUBLE` property is non-zero. Otherwise it is converted and stored as single precision floating point.

Z

A scalar or vector providing the Z components of the vertices. If not provided, Z values default to 0.0. This argument is stored as double precision floating point values if the argument variable is of type `DOUBLE` or if the `DOUBLE` property is non-zero. Otherwise it is converted and stored as single precision floating point.

Keywords

Any property listed under [“IDLgrROI Properties”](#) on page 3929 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.3

CLIP_PLANES keyword added: 5.6

IDLgrROI::PickVertex

The IDLgrROI::PickVertex function method picks a vertex of the region which, when projected onto the given destination device, is nearest to the given 2-D device coordinate.

Syntax

Result = *Obj*->[IDLgrROI::]PickVertex(*Dest*, *View*, *Point* [, **PATH**=*objref*])

Return Value

Returns the index of the nearest region vertex. If two or more vertices are equally nearest to the point, the smallest index of those vertices is returned.

Arguments

Dest

An object reference to an [IDLgrWindow](#) or [IDLgrBuffer](#) for which the pick is to occur.

View

An object reference to the [IDLgrView](#) containing this region.

Point

A two-element vector, [x, y], representing the device location used for picking a nearest vertex.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a location in the data space of the region. Each path object reference specified with this keyword must contain an alias. The selected vertex is computed for the version of the object falling within the specified path. If this keyword is not set, the parent properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

Version History

Introduced: 5.3

IDLgrROI:: SetProperty

The IDLgrROI::SetProperty procedure method sets the value of a property or group of properties for the Object Graphics region.

Syntax

Obj->[IDLgrROI::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrROI Properties](#)” on page 3929 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.3

IDLgrROIGroup

The IDLgrROIGroup object class is an Object Graphics representation of a group of regions of interest.

Superclasses

[IDLanROIGroup](#).

[IDLitComponent](#)

Creation

See “[IDLgrROIGroup::Init](#)” on page 3958.

Properties

Objects of this class have the following properties. See “[IDLgrROIGroup Properties](#)” on page 3948 for details on individual properties.

- [ALL](#)
- [CLIP_PLANES](#)
- [COLOR](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_TEST_FUNCTION](#)
- [DEPTH_WRITE_DISABLE](#)
- [HIDE](#)
- [PARENT](#)
- [XCOORD_CONV](#)
- [XRANGE](#)
- [YCOORD_CONV](#)
- [YRANGE](#)
- [ZCOORD_CONV](#)
- [ZRANGE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

The IDLgrROIGroup class has the following methods:

- [IDLgrROIGroup::Add](#)
- [IDLgrROIGroup::Cleanup](#)
- [IDLgrROIGroup::GetProperty](#)
- [IDLgrROIGroup::Init](#)
- [IDLgrROIGroup::PickRegion](#)
- [IDLgrROIGroup::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.3

IDLgrROIGroup Properties

IDLgrROIGroup objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrROIGroup::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrROIGroup::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrROIGroup::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure with the values of all of the properties associated with the state of this object.

Note

The fields in this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

CLIP_PLANES

A 4-by-*N* floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any x, y, or z coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

A color index or a three-element vector [*red, green, blue*] specifying the color used to drawing the region group. The default color is [0,0,0].

Property Type	COLOR		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this region group should be drawn:

- 0 = draw the region group (the default)
- 1 = do not draw the region group.

Property Type	BOOLEAN		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}X = s_0 + s_1 * \text{Data}X$$

Recommended values are:

$$[(-X_{min})/(X_{max} - X_{min}), 1.0/(X_{max} - X_{min})]$$

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form $[xmin, xmax]$ that specifies the range of x data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Y = s_0 + s_1 * \text{Data}Y$$

Recommended values are:

$$[(-Y_{min})/(Y_{max} - Y_{min}), 1.0/(Y_{max} - Y_{min})]$$

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max} - Z_{min}), 1.0/(Z_{max} - Z_{min})]$$

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLgrROIGroup::Add

The IDLgrROIGroup::Add procedure method adds a region to the region group. Only objects of the IDLgrROI class may be added to the group. The regions in the group must all be of the same type: all points, all paths, or all polygons.

Syntax

Obj->[IDLgrROIGroup:]Add, *ROI*

Arguments

ROI

A reference to an instance of the IDLgrROI object class representing the region of interest to add to the group.

Keywords

Accepts all keywords accepted by the [IDLAnROIGroup::Add](#) method.

Version History

Introduced: 5.3

IDLgrROIGroup::Cleanup

The IDLgrROIGroup::Cleanup procedure method performs all cleanup for an Object Graphics region of interest group object.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrROIGroup:]Cleanup (In a subclass' Cleanup method only.)

Arguments

None

Keywords

None

Version History

Introduced: 5.3

IDLgrROIGroup::GetProperty

The IDLgrROIGroup::Get Property procedure method retrieves the value of a property or group of properties for the region group.

Syntax

Obj->[IDLgrROIGroup:]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrROIGroup Properties](#)” on page 3948 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.3

IDLgrROIGroup::Init

The IDLgrROIGroup::Init function method initializes an Object Graphics region of interest group object.

Note

Init methods are special life cycle methods, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLgrROIGroup' [, *PROPERTY=value*])

or

Result = *Obj*->[IDLgrROIGroup::]Init([*PROPERTY=value*]) (*Only in a subclass' Init method.*)

Note

Keywords can be used in either form. They are omitted in the second form for brevity.

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrROIGroup Properties](#)” on page 3948 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.3

CLIP_PLANES keyword added: 5.6

IDLgrROIGroup::PickRegion

The IDLgrROIGroup::PickRegion function method picks a region within the group which, when projected onto the given destination device, is nearest to the given 2-D device coordinate.

Syntax

Result = *Obj*->[IDLgrROIGroup::]PickRegion(*Dest*, *View*, *Point* [, **PATH**=*objref*])

Return Value

Returns the object reference of the nearest region. If two or more regions are equally nearest to the point, the one that was added to the region group first is returned.

Arguments

Dest

An object reference to an [IDLgrWindow](#) or [IDLgrBuffer](#) for which the pick is to occur.

View

An object reference to the [IDLgrView](#) containing this region.

Point

A two-element vector, [x, y], representing the device location to use for picking a nearest region.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a location in the data space of the region. Each path object reference specified with this keyword must contain an alias. The selected region is computed for the version of the object falling within the specified path. If this keyword is not set, the parent properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

Version History

Introduced: 5.3

IDLgrROIGroup:: SetProperty

The IDLgrROIGroup::Set Property procedure method sets the value of a property or group of properties for the region group.

Syntax

Obj->[IDLgrROIGroup::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrROIGroup Properties](#)” on page 3948 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.3

IDLgrScene

A scene object represents the entire scene to be drawn and serves as a container of [IDLgrView](#) or [IDLgrViewgroup](#) objects.

Superclasses

[IDL_Container](#)

[IDLitComponent](#)

Creation

See “[IDLgrScene::Init](#)” on page 3973.

Properties

Objects of this class have the following properties. See “[IDLgrScene Properties](#)” on page 3965 for details on individual properties.

- [ALL](#)
- [COLOR](#)
- [HIDE](#)
- [REGISTER_PROPERTIES](#)
- [TRANSPARENT](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrScene::Add](#)
- [IDLgrScene::Cleanup](#)
- [IDLgrScene::GetByName](#)
- [IDLgrScene::GetProperty](#)
- [IDLgrScene::Init](#)
- [IDLgrScene::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrScene Properties

IDLgrScene objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via `IDLgrScene::GetProperty`. Properties with the word “Yes” in the “Init” column of the property table can be set via `IDLgrScene::Init`. Properties with the word “Yes” in the “Set” column in the property table can be set via `IDLgrScene::SetProperty`.

Note

For a discussion of the property description tables shown below, see “[About Object Property Descriptions](#)” on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR

The color to which the scene should be erased before drawing. The color may be specified as a color lookup table index or a three-element vector [*red, green, blue*].

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Background color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: No	Set: No	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

For IDLgrScene, the available properties (and their iTool data types) are:

- HIDE (Boolean)

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

TRANSPARENT

A Boolean value that determines whether to disable window clearing. If this property is not set, the destination object in use by the scene is automatically erased when the scene is initialized.

Property Type	BOOLEAN		
Name String	Transparent		
Get: Yes	Set: Yes	Init: No	Registered: Yes

IDLgrScene::Add

The IDLgrScene::Add function method verifies that the added item is an instance of an [IDLgrView](#) or IDLgrViewgroup object. If it is, IDLgrScene:Add adds the view or viewgroup to the specified scene.

Syntax

Obj->[IDLgrScene::]Add, *View* [, **POSITION**=*index*]

Arguments

View

An instance of the [IDLgrView](#) or [IDLgrViewgroup](#) object class.

Keywords

POSITION

Set this keyword equal to the zero-based index of the position within the container at which the new object should be placed.

Version History

Introduced: 5.0

IDLgrScene::Cleanup

The IDLgrScene::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrScene::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrScene::GetByName

The IDLgrScene::GetByName function method finds contained objects by name and returns the object reference to the named object. If the named object is not found, the GetByName function returns a null object reference.

Note

The GetByName function does *not* perform a recursive search through the object hierarchy. If a fully qualified object name is not specified, only the contents of the current container object are inspected for the named object.

Syntax

Result = *Obj*->[IDLgrScene::]GetByName(*Name*)

Return Value

Returns the object reference to the named object. If the named object is not found, returns a null object reference.

Arguments

Name

A string containing the name of the object to be returned.

Object naming syntax is very much like the syntax of a UNIX filesystem. Objects contained by other objects can include the name of their parent object; this allows you to create a fully qualified name specification. For example, if `object1` contains `object2`, which in turn contains `object3`, the string specifying the fully qualified object name of `object3` would be `'object1/object2/object3'`.

Object names are specified relative to the object on which the `GetByName` method is called. If used at the beginning of the name string, the `/` character represents the top of an object hierarchy. The string `'..'` represents the object one level “up” in the hierarchy.

Keywords

None

Version History

Introduced: 5.0

IDLgrScene::GetProperty

The IDLgrScene::GetProperty procedure method retrieves the value of a property or group of properties for the contour.

Syntax

Obj->[IDLgrScene::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrScene Properties](#)” on page 3965 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrScene::Init

The IDLgrScene::Init function method initializes the scene object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrScene' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrScene:]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrScene Properties](#)” on page 3965 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrScene:: SetProperty

The IDLgrScene::SetProperty procedure method sets the value of a property or group of properties for the buffer.

Syntax

Obj->[IDLgrScene::]SetProperty[, *PROPERTY=**value*]

Arguments

None

Keywords

Any property listed under “[IDLgrScene Properties](#)” on page 3965 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrSurface

A surface object represents a shaded or vector representation of a mesh grid.

An IDLgrSurface object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrSurface::Init](#)” on page 4007.

Properties

Objects of this class have the following properties. See “[IDLgrSurface Properties](#)” on page 3978 for details on individual properties.

- [ALL](#)
- [ALPHA_CHANNEL](#)
- [AMBIENT](#)
- [BOTTOM](#)
- [CLIP_PLANES](#)
- [COLOR](#)
- [DATA](#)
- [DATA_X](#)
- [DATA_Y](#)
- [DATA_Z](#)
- [DEPTH_OFFSET](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_TEST_FUNCTION](#)
- [DEPTH_WRITE_DISABLE](#)
- [DIFFUSE](#)
- [DOUBLE](#)
- [EMISSION](#)
- [EXTENDED_LEGO](#)
- [HIDDEN_LINES](#)
- [HIDE](#)
- [LINESTYLE](#)
- [MAX_VALUE](#)
- [MIN_VALUE](#)
- [PALETTE](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [RESET_DATA](#)
- [SHADE_RANGE](#)

- SHADING
- SHININESS
- SKIRT
- STYLE
- TEXTURE_HIGHRES
- TEXTURE_MAP
- USE_TRIANGLES
- XCOORD_CONV
- YCOORD_CONV
- ZCOORD_CONV
- ZRANGE
- SHARE_DATA
- SHOW_SKIRT
- SPECULAR
- TEXTURE_COORD
- TEXTURE_INTERP
- THICK
- VERT_COLORS
- XRANGE
- YRANGE
- ZERO_OPACITY_SKIP

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrSurface::Cleanup](#)
- [IDLgrSurface::GetCTM](#)
- [IDLgrSurface::GetProperty](#)
- [IDLgrSurface::Init](#)
- [IDLgrSurface::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

ALPHA_CHANNEL, AMBIENT, DIFFUSE, EMISSION, SHININESS, and SPECULAR added: 6.1

IDLgrSurface Properties

IDLgrSurface objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrSurface::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrSurface::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrSurface::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

Set this property to a floating-point value in the range [0.0, 1.0] to specify the opacity of the surface. The default value of 1.0 causes IDL to draw the surface completely opaque. If the value of this property is less than 1.0, then the pixels of the surface are blended with the pixels already on the screen. The color of the surface is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha.

Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results. This also applies to the individual facets of the surface that IDL generates to represent the surface geometry. Since you do not have control over the rendering order of the facets that form the surface, there is little you can do to change the ordering so that the facets draw in a back-to-front order. Therefore, using this

property to draw semi-transparent surfaces is not recommended unless the surface geometry and orientation is such that the surface can be viewed without self-overlapping areas.

This property has no effect on devices using indexed color mode.

When used with textures, the `ALPHA_CHANNEL` value controls the opacity of the polygon in the same manner as when not using a texture. If the texture contains alpha information, the effective alpha of each texel is the texel's alpha multiplied by the polygon's alpha value (`ALPHA_CHANNEL` property).

Note

For more detailed information on alpha channel support for IDL Object Graphics, see [“Alpha Channel and Objects”](#) in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

AMBIENT

Set this property to a three-element vector [*red, green, blue*] to specify the ambient reflective color of the object. The default is black or [0,0,0], which specifies that the value of the `COLOR` property determines the ambient reflective color of the object. Setting an ambient color other than black makes the object use that color for ambient reflectance. Ambient reflectance affects the overall color of the object and is most noticeable where the object is not directly lit by a light source. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Ambient reflective color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

BOTTOM

A three-element vector [*red, green, blue*] specifying the color used to draw the bottom of the surface. Set this property to a scalar to draw the bottom with the same

color as the top. Setting a bottom color is only supported when the destination device uses RGB color mode.

Property Type	COLOR		
Name String	Bottom color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by- N floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form $[A, B, C, D]$, where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any x , y , or z coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

The color to be used as the foreground color for this model. The color may be specified as a color lookup table index or as a three-element vector [*red*, *green*, *blue*]. The default is $[0, 0, 0]$.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DATA

An array of any type that specifies the surface data. The array format is based upon the following:

- If the DATAZ property (or Data argument) was set to an $M \times N$ array (via the `::Init` or `::SetProperty` methods), then the value returned via the DATA property is a $3 \times M \times N$ array, `[[[x,y,z],[x,y,z],...],[x,y,z],[x,y,z],...],...]`.
- If the DATAZ keyword (or Data argument) was never set, then the value returned via the DATA property is an undefined variable.

The value of the DATA array represents the combination of the DATAZ, DATAY, and DATAZ values defined using `::SetProperty`. If DATAZ was set, and DATAZ and DATAY were not, the DATA value returned includes the implied X and Y grid locations.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

DATAZ

A floating-point vector or a two-dimensional array specifying the X coordinates of the surface grid. This property is the same as the X argument described for [“IDLgrSurface::Init”](#) on page 4007. This property is stored as double precision floating point values if the property is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Property Type	Floating-point vector or array		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

DATAY

A floating-point vector or a two-dimensional array specifying the Y coordinates of the surface grid. This property is the same as the Y argument described for “[IDLgrSurface::Init](#)” on page 4007. This property is stored as double precision floating point values if the property is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Property Type	Floating-point vector or array		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

DATAZ

A two-dimensional floating-point array to display as a surface. This property is the same as the Z argument described for “[IDLgrSurface::Init](#)” on page 4007. This property is stored as double precision floating point values if the property is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

DEPTH_OFFSET

An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.

The units are “Z-Buffer units,” where a value of 1 is used to specify a distance that corresponds to a single step in the device’s Z-Buffer.

Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms. This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for “layering” complex scenes with multiple DEPTH_OFFSET values. It is safest to

use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value, such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because a set of DEPTH_OFFSET values may produce better results on one machine than on another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.

Note

DEPTH_OFFSET has no effect unless the value of the STYLE property is 2 or 6 (Filled or LegoFilled).

Property Type	INTEGER		
Name String	Depth offset		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DIFFUSE

Set this property to a three-element vector [*red*, *green*, *blue*] to specify the diffuse reflectance color of the object. The default is black or [0,0,0], which specifies that the value of the COLOR property determines the diffuse reflective color of the object. Setting a diffuse color other than black makes the object use that color for diffuse reflectance. Diffuse reflectance is an important factor in determining the color of the object and is affected by the color and angle of the light sources in a scene. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Diffuse reflective color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DOUBLE

A Boolean value that indicates whether data provided by any of the input arguments will be stored in this object as using double-precision floating-point format.

- Set this property equal to 1 to convert input data to double-precision floating-point format.
- Set this property equal to 0 to convert input data to single-precision floating-point format.

- If you do not specify a value for this property, no data type conversion will be performed, and the data will be stored with its original precision.

Setting this property may be desirable if the data consists of large integers that cannot be accurately represented in single-precision floating-point arithmetic. This property is also automatically set to 1 if any of the input arguments are stored using a variable of type DOUBLE.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

EMISSION

Set this property to a three-element vector [*red, green, blue*] to specify the emissive color of the object. The default is black or [0,0,0], which specifies that the object contributes no additional color in an emissive sense. Setting an emission color other than black makes the object appear as if it were emitting that color. This property is used to simulate light sources or other objects that emit light in a scene. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Emissive color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

EXTENDED_LEGO

A Boolean value that determines whether to force the IDLgrSurface object to display the last row and column of data when lego display styles are selected.

Property Type	BOOLEAN		
Name String	Show last lego row/column		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDDEN_LINES

A Boolean value that determines whether to draw point and wireframe surfaces using hidden line (point) removal. By default, hidden line removal is disabled.

Property Type	BOOLEAN		
Name String	Remove hidden lines		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LINestyle

An integer value that indicates the line style to use to draw the surface lines. The value can be either an integer value specifying a pre-defined line style, or a two-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINestyle property equal to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot

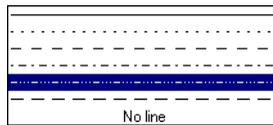
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector [*repeat*, *bitmask*], where *repeat* indicates the number of times consecutive runs of 1's or 0's in the *bitmask* should be repeated. (That is, if three consecutive 0's appear in the *bitmask* and the value of *repeat* is 2, then the line that is drawn will have six consecutive bits turned off.) The value of *repeat* must be in the range $1 \leq \textit{repeat} \leq 255$.

The *bitmask* indicates which pixels are drawn and which are not along the length of the line. *Bitmask* is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

In a property sheet, this property appears as follows:



Property Type	LINESTYLE		
Name String	Line style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MAX_VALUE

A double-precision floating-point value that determines the maximum value to be plotted. If this property is present, data values greater than the value of `MAX_VALUE` are treated as missing data and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Maximum value		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MIN_VALUE

A double-precision floating-point value that determines the minimum value to be plotted. If this property is present, data values less than the value of MIN_VALUE are treated as missing data and are not plotted. Note that the IEEE floating-point value NaN is also treated as missing data. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Minimum value		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class) that defines the color palette of this object. This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the PALETTE property on this object is not set, the destination object PALETTE property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESET_DATA

A Boolean value that determines whether to treat the data provided via one of the DATA[XYZ] properties as a new data set unique to this object, rather than overwriting data that is shared by other objects. There is no reason to use this property if the object on which the property is being set does not currently share data with another object (that is, if the SHARE_DATA property is not in use). This property has no effect if no new data is provided via a DATA[XYZ] property. .

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHADE_RANGE

A two-element integer array that specifies the range of pixel values (color indices) to use for shading. The first element is the color index for the darkest pixel. The second element is the color element for the brightest pixel. This value is ignored when the polygons are drawn to a graphics destination that uses the RGB color model.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SHADING

An integer representing the type of shading to use if STYLE is set to 2 (Filled).

- 0 = Flat (default): The color has a constant intensity for each face of the surface, based on the normal vector.
- 1 = Gouraud: The colors are interpolated between vertices, and then along scanlines from each of the edge intensities.

Gouraud shading may be slower than flat shading, but results in a smoother appearance.

In a property sheet, this property appears as an enumerated list with the following options:

- Flat
- Gouraud

Property Type	ENUMLIST		
Name String	Fill shading		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SHARE_DATA

An object reference to an object whose data is to be shared by this surface. A surface may only share data with another surface. The `SHARE_DATA` property is intended for use when data values are not set via an argument to the object's `Init` method or by setting the object's `DATA` property.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: No	Set: Yes	Init: Yes	Registered: No

SHININESS

Set this property to a floating-point value between 0.0 and 128.0, inclusive. The default value is 25.0. Higher values of shininess concentrate specular highlights into

smaller and brighter areas, while lower values will spread out specular highlights over a wider area and make them appear dimmer.

Property Type	FLOAT		
Name String	Shininess		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SHOW_SKIRT

A Boolean value that determines whether to enable skirt drawing. The default is to disable skirt drawing. .

Property Type	BOOLEAN		
Name String	Show skirt		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SKIRT

A floating-point value that determines the Z value at which a skirt is to be defined around the array. The Z value is expressed in data units; the default is 0.0. If a skirt is defined, each point on the four edges of the surface is connected to a point on the skirt which has the given Z value, and the same X and Y values as the edge point. In addition, each point on the skirt is connected to its neighbor. The skirt value is ignored if skirt drawing is disabled (see SHOW_SKIRT above). IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Skirt bottom height		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SPECULAR

Set this property to a three-element vector [*red*, *green*, *blue*] to specify the color of the specular highlights of an object. The default is white or [255,255,255]. This

property is used to simulate and control reflective highlights on an object. This property has no effect on devices using indexed color mode.

Property Type	COLOR		
Name String	Specular highlight color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

STYLE

An integer value that indicates the style to be used to draw the surface. Valid values are:

- 0 = Points
- 1 = Wire mesh (the default)
- 2 = Filled
- 3 = RuledXZ
- 4 = RuledYZ
- 5 = Lego
- 6 = LegoFilled: for outline or shaded and stacked histogram-style plots.

Note

Stitching effects may appear when combining an unfilled style (STYLE=0, 1, 3, 4, or 5) and hidden lines (HIDDEN_LINES=1). In such a case, missing lines may be caused by depth buffer resolution, limited depth offset support of the graphics card, or invisible parts of a surface occluding line visibility.

In a property sheet, this property appears as an enumerated list with the following options:

- Points
- Wire mesh
- Filled
- Ruled XZ
- Ruled YZ
- Lego
- Lego filled

Property Type	ENUMLIST		
Name String	Surface style		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTURE_COORD

A 2-by- n floating-point array of texture map coordinate pairs that are internally mapped to surface vertex coordinates. The number of data points in the surface and the value of n in the 2-by- n array should be equal.

Each texture map coordinate pair is mapped to a surface vertex coordinate pair in array order. For example, with a 2-by-9 element array, texture map coordinates will be mapped to a 3-by-3 surface as follows:

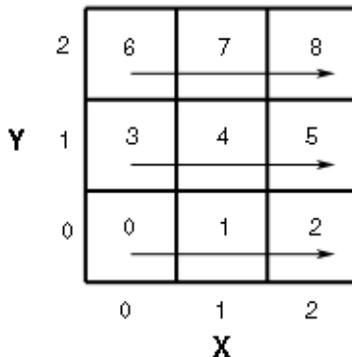


Figure A-1: Mapping Texture Coordinates

Use this property in conjunction with the TEXTURE_MAP property to warp images over the surface. To stretch (or shrink) the texture map to cover the surface mesh completely, set TEXTURE_COORD to a scalar. By default, IDL stretches (or shrinks) the texture map to cover the surface mesh completely, and sets TEXTURE_COORD to a scalar (-1).

Texture coordinates are normalized. This means that the $m \times n$ image object specified via the TEXTURE_MAP property is mapped into the range [0.0, 0.0] to [1.0, 1.0]. As the texture coordinates run from 0.0 to 1.0 across a texture, a texture coordinate of [0.5, 0.5] at a vertex specifies that the image pixel at the exact center of the image is

mapped to the surface at that vertex. If texture coordinates outside the range [0.0, 0.0] to [1.0, 1.0] are specified, the image object is tiled into the larger range.

For example, suppose the image object specified via `TEXTURE_MAP` is a 256 x 256 array, and we want to map the image into a surface two units on each side. To completely fill the 2 x 2 surface with a single copy of the image:

```
TEXTURE_COORD = [[0,0], [1,0], [0,1], [1,1]]
```

To fill the 2 x 2 surface with four tiled copies of the image:

```
TEXTURE_COORD = [[0,0], [2,0], [0,2], [2,2]]
```

Because of the way in which high-resolution textures require modified texture coordinates, if the `TEXTURE_COORD` property is used, `TEXTURE_HIGHRES` will be disabled.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TEXTURE_HIGHRES

An integer value that determines the type of texture tiling, if any, that will be used to maintain the full pixel resolution of the original texture image. Set this property to one of the following options:

- 0 = If the texture map is larger than the maximum resolution supported by the 3-D hardware, the texture is scaled down to this resolution.
- 1 = If the texture map is larger than the maximum resolution supported by the hardware, texture tiling is used to ensure no loss of data. As a large image could easily consume all available video memory, this mode employs texture Level of Detail (LoD). Rather than loading the entire image into the texture tiles at the full resolution, the image tiles are downsampled based on the level of zoom. If the entire image is visible the texture tiles will contain highly downsampled data from the image to conserve video memory. As a portion of the image is zoomed in on, progressively higher resolution image data is loaded into the visible tiles, until the maximum resolution is reached. The tiles that are not visible during zooming are not loaded with high resolution data.

Note

This mode is recommended if IDL is running on modern 3-D hardware and resolution loss due to downscaling becomes problematic when `TEXTURE_HIGHRES=0`.

- 2 = If the texture map is larger than the maximum resolution supported by the hardware, texture tiling is used to ensure no loss of data. However, the highest resolution data is loaded into all tiles. LoD tiling is disabled. This mode should only be used if the video memory is sufficient for the image size.

If not set, and the texture map is larger than the maximum resolution supported by the 3-D hardware, the texture is scaled down to the maximum resolution supported by the 3-D hardware on your system. The default value is 0.

Note

`TEXTURE_HIGHRES` is supported only when `STYLE=2` (Filled).

Note

Because of the way in which high-resolution textures require modified texture coordinates, if you specify the `TEXTURE_COORD` property, high resolution textures (`TEXTURE_HIGHRES`) will be disabled.

In a property sheet, this property appears as an enumerated list with the following options:

- No tiling
- LOD tiling
- Tiling

Property Type	ENUMLIST		
Name String	Texture hires		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTURE_INTERP

A Boolean value that indicates whether bilinear sampling is to be used with texture mapping. The default method is nearest-neighbor sampling.

In a property sheet, this property appears as an enumerated list with the following options:

- Nearest neighbor
- Bilinear

Property Type	ENUMLIST		
Name String	Texture interpolation		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

TEXTURE_MAP

An object reference to an instance of the [IDLgrImage](#) object class to be texture mapped onto the surface. If this property is omitted or set to a null object reference, no texture map is applied and the surface is filled with the color specified by the `COLOR` or `VERT_COLORS` property. If both `TEXTURE_MAP` and `COLORS` or `VERT_COLORS` properties exist, the color of the texture is modulated by the base color of the object. (This means that for the clearest display of the texture image, the `COLOR` property should be set equal to [255,255,255].) By default, the texture map will be stretched (or shrunk) to cover the surface mesh completely.

Setting `TEXTURE_MAP` to the object reference of an `IDLgrImage` that contains an alpha channel allows you to create a transparent `IDLgrSurface` object. If an alpha channel is present in the `IDLgrImage` object, IDL blends the texture using the blend function `src = alpha` and `dst = 1 - alpha`, which corresponds to a `BLEND_FUNCTION` of (3,4) as described for the `IDLgrImage` object.

If a texture is provided without texture coordinates, `IDLgrSurface` generates its own texture mapping coordinates to map the texture onto the surface without resampling artifacts, even if the provided texture image does not have dimensions that are an exact power of two. If texture coordinates are provided, the image is resampled to the nearest larger dimensions that are exact powers of two.

Note

Texture mapping is disabled when rendering to a destination object that uses Indexed color mode.

Note

Texture mapping is applied to all styles that are set by the `STYLE` property except `Lego` and `LegoFilled`.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Texture map		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

THICK

A floating-point value between 1.0 and 10.0, specifying the line thickness to use to draw surface lines, in points. The default is 1.0 points.

In a property sheet, this property appears as follows:



Property Type	THICKNESS		
Name String	Line thickness		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

USE_TRIANGLES

A Boolean value that determines whether to force the IDLgrSurface object to use triangles instead of quads to draw the surface and skirt.

In a property sheet, this property appears as an enumerated list with the following options:

- Quads
- Triangles

Property Type	BOOLEAN		
Name String	Draw method		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VERT_COLORS

A vector of colors to be used to specify the color of a surface vertex. The vector may be of the form $[n]$ where each entry is a color index, or of the form $[3,n]$ where each 3-element row is an RGB color, or of the form $[4,n]$ where each 4-element row is an RGBA color. To remove vertex colors after they have been set, set VERT_COLORS to a scalar.

If VERT_COLORS is not specified, the entire surface is drawn in the single color provided by the COLOR property, which is the default action.

If SHADING is set to 0 (Flat, the default) the color of a surface polygon is the color of the first vertex of that surface polygon. Color is interpolated between vertices if SHADING is set to 1 (Gouraud).

If there are more vertices than elements in VERT_COLORS, the elements of VERT_COLORS are cyclically repeated.

Note

If the surface object is being rendered on a destination device that uses the Indexed color model, and the view that contains the surface also contains one or more light objects, the VERT_COLORS property is ignored and the SHADE_RANGE property is used instead.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Vertex colors		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}X = s_0 + s_1 * \text{Data}X$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form [*xmin*, *xmax*] that specifies the range of *x* data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, [*s*₀, *s*₁], of scaling factors used to convert *Y* coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{Normalized}Y = s_0 + s_1 * \text{Data}Y$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is $[0.0, 1.0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZERO_OPACITY_SKIP

A Boolean value that determines whether to gain finer control over the rendering of textured surface pixels (texels) by setting an opacity of 0 in the texture map. Texels with zero opacity do not affect the color of a screen pixel since they have no opacity. If this property is set to 1, any texels are “skipped” and not rendered at all. If this property is set to zero, the Z-buffer is updated for these pixels and the display image is not affected as noted above. By updating the Z-buffer without updating the display image, the surface can be used as a *clipping* surface for other graphics primitives drawn after the current graphics object. The default value for this property is 1.

Note

This property has no effect if no texture map is used or if the texture map in use does not contain an opacity channel.

Property Type	BOOLEAN		
Name String	Skip zero opacity		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ZRANGE

A two-element floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

ALPHA_CHANNEL, AMBIENT, DIFFUSE, EMISSION, SHININESS, and SPECULAR properties added: 6.1

IDLgrSurface::Cleanup

The IDLgrSurface::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrSurface:]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrSurface::GetCTM

The IDLgrSurface::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrSurface::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the surface object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrSurface::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrSurface::GetProperty

The IDLgrSurface::GetProperty procedure method retrieves the value of a property or group of properties for the surface.

Syntax

Obj->[IDLgrSurface::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrSurface Properties](#)” on page 3978 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrSurface::Init

The IDLgrSurface::Init function method initializes the surface object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrSurface' [, Z [, X, Y]] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrSurface::]Init([Z [, X, Y]] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

X

A vector or two-dimensional array specifying the X coordinates of the grid. If this argument is a vector, each element of X specifies the X coordinates for a column of Z (e.g., X[0] specifies the X coordinate for Z[0, *]). If X is a two-dimensional array, each element of X specifies the X coordinate of the corresponding point in Z (X_{ij} specifies the X coordinate of Z_{ij}). This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Y

A vector or two-dimensional array specifying the Y coordinates of the grid. If this argument is a vector, each element of Y specifies the Y coordinates for a column of Z (e.g., Y[0] specifies the Y coordinate for Z[*], 0]). If Y is a two-dimensional array, each element of Y specifies the Y coordinate of the corresponding point in Z (Y_{ij} specifies the Y coordinate of Z_{ij}). This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Z

The two-dimensional array to be displayed. If X and Y are provided, the surface is defined as a function of the (X, Y) locations specified by their contents. Otherwise, the surface is generated as a function of the array indices of each element of Z. This argument is stored as double precision floating point values if the argument variable is of type DOUBLE or if the DOUBLE property is non-zero, otherwise it is stored as single precision floating point.

Keywords

Any property listed under [“IDLgrSurface Properties”](#) on page 3978 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES keyword added: 5.6

IDLgrSurface::SetProperty

The IDLgrSurface::SetProperty procedure method sets the value of a property or group of properties for the surface.

Syntax

Obj->[IDLgrSurface::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrSurface Properties](#)” on page 3978 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrSymbol

A symbol object represents a graphical element that is plotted relative to a particular position.

Note

Seven predefined symbols are provided by IDL.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrSymbol::Init](#)” on page 4018.

Properties

Objects of this class have the following properties. See “[IDLgrSymbol Properties](#)” on page 4012 for details on individual properties.

- [ALL](#)
- [ALPHA_CHANNEL](#)
- [COLOR](#)
- [DATA](#)
- [SIZE](#)
- [THICK](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrSymbol::Cleanup](#)
- [IDLgrSymbol::GetProperty](#)
- [IDLgrSymbol::Init](#)
- [IDLgrSymbol::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrSymbol Properties

IDLgrSymbol objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrSymbol::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrSymbol::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrSymbol::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

A floating-point value in the range [0.0, 1.0] to specify the opacity of the symbol. The default value of 1.0 causes IDL to draw the symbol completely opaque. If the value of this property is less than 1.0, then the pixels of the symbol are blended with the pixels already on the screen, where the color of the symbol is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha.

Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results.

This property has no effect on devices using indexed color mode.

Some of the objects that use symbols have properties that control the alpha value that is ultimately used to render the symbol. This ultimate alpha value might come from the parent object's ALPHA_CHANNEL property, an alpha value from a vertex color in the parent object, or the value of this property, depending on the settings of properties in the parent object. Refer to the reference guide for the parent properties for information on these controls.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see “[Alpha Channel and Objects](#)” in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

COLOR

The color used to draw the symbol. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default color is the color of the object for which this symbol is being used.

Property Type	Color		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DATA

Either an integer value (from the following list), or an object reference to either an IDLgrModel object, or atomic graphic object. Setting this property is equivalent to setting the *Data* argument.

Use one of the following scalar-represented internal default symbols:

- 0 = No symbol
- 1 = Plus sign, '+' (default)
- 2 = Asterisk
- 3 = Period (Dot)

- 4 = Diamond
- 5 = Triangle
- 6 = Square
- 7 = X
- 8 = “Greater-than” Arrow Head (>)
- 9 = “Less-than” Arrow Head (<)

If an instance of the IDLgrModel object class or an atomic graphic object is used, the object tree is used as the symbol. For best results, the object should fill the domain from -1 to +1 in all dimensions. The pre-defined symbols listed above are all defined in the domain -1 to +1.

Property Type	Integer or object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

SIZE

A one-, two-, or three-element floating-point vector describing the X, Y, and Z scaling factors to be applied to the symbol. The default is [1.0, 1.0, 1.0].

- If SIZE is specified as a scalar, then the X, Y, and Z scale factors are all equal to the scalar value.
- If SIZE is specified as a 2-element vector, then the X and Y scale factors are as specified by the vector, and the Z scale factor is 1.0.
- If SIZE is specified as a 3-element vector, then the X, Y, and Z scale factors are as specified by the vector.

IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

THICK

A floating-point value between 1.0 and 10.0, specifying the line thickness to used to draw any lines that make up the symbol, in points. The default is 1.0 points.

Property Type	Floating-point		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Version History

ALPHA_CHANNEL added: 6.1

IDLgrSymbol::Cleanup

The IDLgrSymbol::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrSymbol::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrSymbol::GetProperty

The IDLgrSymbol::GetProperty procedure method retrieves the value of a property or group of properties for the symbol.

Syntax

Obj->[IDLgrSymbol::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrSymbol Properties](#)” on page 4012 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrSymbol::Init

The IDLgrSymbol::Init function method initializes the plot symbol.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrSymbol' [, Data] [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrSymbol::]Init([Data] [, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

Data

Either an integer value from the list shown below, or an object reference to either an IDLgrModel object or atomic graphic object.

Use one of the following scalar-represented internal default symbols:

- 0 = No symbol
- 1 = Plus sign, '+' (default)
- 2 = Asterisk
- 3 = Period (Dot)
- 4 = Diamond

- 5 = Triangle
- 6 = Square
- 7 = X
- 8 = “Greater-than” Arrow Head (>)
- 9 = “Less-than” Arrow Head (<)

If an instance of the `IDLgrModel` object class or an atomic graphic object is used, the object tree is used as the symbol. For best results, the object should fill the domain from -1 to +1 in all dimensions. The pre-defined symbols listed above are all defined in the domain -1 to +1.

Keywords

Any property listed under [“IDLgrSymbol Properties”](#) on page 4012 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

Data argument added: 5.6

IDLgrSymbol:: SetProperty

The IDLgrSymbol::SetProperty procedure method sets the value of a property or group of properties for the symbol.

Syntax

Obj->[IDLgrSymbol::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrSymbol Properties](#)” on page 4012 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrTessellator

A tessellator object decomposes a polygon description into a set of triangles. Use the tessellator object to convert complex and/or concave polygons into a form suitable for drawing with the IDLgrPolygon object. IDLgrPolygon can draw only convex polygons correctly.

The polygon contours may intersect each other and may be self-intersecting. The contours may be disjoint, overlapping, or contained within other contours. The contours may also be degenerate, may contain repeated points, and may or may not be closed. The order of the vertices may be either counter-clockwise or clockwise. For best results, the polygon contours should be coplanar.

The tessellator object uses the "odd-winding rule" to determine if a point is in the interior of the polygon and therefore contained in a triangle output by the tessellator. With this rule, a point is in the interior if it is circled an odd number of times as we travel around each of the contours.

Note

The INTERIOR keyword for the AddPolygon method is no longer needed to inform the tessellator that the polygon defines an exterior or interior boundary. This keyword is ignored by the tessellator because it now performs the interior testing using the odd-winding rule.

Specify polygon contours with calls to the [IDLgrTessellator::AddPolygon](#) method. After adding all the contours, use the [IDLgrTessellator::Tessellate](#) method to perform the tessellation and retrieve the resulting list of vertices and connectivity array.

If the polygon contours contain intersecting or self-intersecting contours, the tessellator may return vertices that were not in the original set of vertices specified with the [IDLgrTessellator::AddPolygon](#) method. These vertices are created by the intersecting contours.

If your vertex data also includes other information, such as a color for each vertex, then you may wish for this extra information to be created for any new vertices generated by the tessellator. Use the AUXDATA keywords for [IDLgrTessellator::AddPolygon](#) and [IDLgrTessellator::Tessellate](#) to pass in and retrieve your per-vertex data. The tessellator object interpolates the per-vertex data from neighboring vertices to create new per-vertex data for the new vertices it generates.

In the following example of handling per-vertex data with generated vertices, the polygon is a simple self-intersecting "bow-tie" polygon. It is submitted to the

tessellator with four vertices, but the tessellator returns a fifth at the point of self-intersection. Color data for the four original vertices is also supplied, and the tessellator returns a fifth color.

```

PRO tessaux
  oTess = OBJ_NEW('IDLgrTessellator')
  colors = [[0,255,0],[0,255,0],[0,64,0],[0,64,0]]
  oTess->AddPolygon, [0,1,0,1], [0,0,1,1], AUXDATA=colors
  result = oTess->Tessellate(v, c, AUXDATA=aux)
  PRINT, v[* ,4]
  PRINT, aux[* ,4]
  oPoly = OBJ_NEW('IDLgrPolygon', v, POLYGONS=c, $
VERT_COLORS=aux, SHADING=1)
  XOBJVIEW, oPoly, /BLOCK
  OBJ_DESTROY, [oTess, oPoly]
END

```

The generated output will be:

```

0.500000  0.500000
0  159  0

```

Superclasses

None.

Creation

See “[IDLgrTessellator::Init](#)” on page 4028.

Properties

Objects of this class have no properties of their own.

Methods

This class has the following methods:

- [IDLgrTessellator::AddPolygon](#)
- [IDLgrTessellator::Cleanup](#)
- [IDLgrTessellator::Init](#)
- [IDLgrTessellator::Reset](#)
- [IDLgrTessellator::Tessellate](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrTessellator Properties

Objects of this class have no properties of their own.

IDLgrTessellator::AddPolygon

The IDLgrTessellator::AddPolygon procedure method adds a polygon to the tessellator object.

Syntax

Obj->[IDLgrTessellator::]AddPolygon, X [, Y[, Z]] [, **AUXDATA**=array of auxiliary data] [, **INTERIOR**] [, **POLYGON**=array of polygon descriptions]

Arguments

X

A $1 \times n$, $2 \times n$, or $3 \times n$ array of polygon vertices.

Y

A vector of Y values. If X and Y are both specified, they must be one-dimensional vectors of the same length.

Z

A vector of Z values. If X, Y, and Z are all specified, they must all three be one-dimensional vectors of the same length. If no Z values are specified, the Z value for the polygon is set to 0.

Keywords

AUXDATA

Set this keyword to an array of auxiliary per-vertex data. This array must have dimensions $[m,n]$ where m is the number of auxiliary data items per vertex and n is the number of vertices specified in the X, Y, and Z arguments. If you specify AUXDATA in any invocation of the AddPolygon method, you must specify it on all invocations of the method for the polygons to be tessellated together with the Tessellate method. Further, the value of m in the dimensions must be the same for all polygons. That is, all polygons must have the same number of auxiliary data items for each vertex.

INTERIOR

Set this keyword to set a polygon to be an interior polygon, which is treated as a hole in the exterior polygons.

Note

The INTERIOR keyword for the AddPolygon method is no longer needed to inform the tessellator that the polygon defines an exterior or interior boundary. This keyword is ignored by the tessellator because it now performs the interior testing using the odd-winding rule.

POLYGON

Set this keyword to an array of polygon descriptions. A polygon description is an integer or long word array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0..i_{n-1}$ are indices into the X, Y, and Z arguments that represent the polygon vertices. To ignore an entry in the POLYGON array, set the vertex count, n , to 0. To end the drawing list, even if additional array space is available, set n to -1. If this keyword is not specified, a single polygon will be generated.

Note

The connectivity array described by POLYGONS allows you to add multiple polygons to the tessellator object with a single AddPolygon operation.

Version History

Introduced: 5.0

AUXDATA keyword added: 5.6

IDLgrTessellator::Cleanup

The IDLgrTessellator::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY,*Obj*

or

Obj->[IDLgrTessellator::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrTessellator::Init

The IDLgrTessellator::Init function method initializes the tessellator object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLgrTessellator')

or

Result = *Obj*->[IDLgrTessellator::]Init() (*Only in a subclass' Init method.*)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrTessellator::Reset

The IDLgrTessellator::Reset procedure method resets the object's internal state. All previously added polygons are removed from memory and the object is prepared for a new tessellation task.

Syntax

Obj->[IDLgrTessellator::]Reset

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrTessellator::Tessellate

The IDLgrTessellator::Tessellate function method performs the actual tessellation.

Syntax

```
Result = Obj->[IDLgrTessellator::]Tessellate( Vertices, Poly [, AUXDATA=variable]  
[, /QUIET] )
```

Return Value

Returns 1, and the contents of Vertices and Poly are set to the results of the tessellation. Returns 0 if tessellation fails.

Arguments

If the tessellation succeeds, IDLgrTessellator::Tessellate returns 1 and the contents of Vertices and Poly are set to the results of the tessellation. If the tessellation fails, the function returns 0.

Vertices

A $2 \times n$ array if all the input polygons were 2-D. A $3 \times n$ array if all the input polygons were 3-D.

Poly

An array of polygon descriptions. A polygon description is an integer or long word array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0..i_{n-1}$ are indices into the X , Y , and Z arguments that represent the polygon vertices.

Note

On output, the *Vertices* array can be used as the value of the DATA property, and the *Poly* array can be used as the value of the POLYGON property, of an IDLgrPolygon object.

Keywords

AUXDATA

Set this keyword to a named variable that receives the auxiliary data associated with each vertex returned in the Vertices argument. The data is an $[m, n]$ array where m is the number of per-vertex auxiliary data items specified in the call(s) to the AddPolygon method, and n is the number of vertices returned in the Vertices argument. The type of the returned auxiliary data is the same as the type of the data supplied with the AUXDATA keyword in the last call to AddPolygon.

QUIET

Set this keyword to suppress warning and error message generation due to tessellation errors. !ERROR_STATE is not updated in the case of the return value being '0' when the QUIET keyword is specified.

Version History

Introduced: 5.0

AUXDATA keyword added: 5.6

IDLgrText

A text object represents one or more text strings that share common rendering attributes. An IDLgrText object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrText::Init](#)” on page 4053.

Properties

Objects of this class have the following properties. See “[IDLgrText Properties](#)” on page 4034 for details on individual properties.

- [ALIGNMENT](#)
- [ALPHA_CHANNEL](#)
- [CHAR_DIMENSIONS](#)
- [COLOR](#)
- [DEPTH_TEST_FUNCTION](#)
- [ENABLE_FORMATTING](#)
- [FILL_COLOR](#)
- [HIDE](#)
- [LOCATIONS](#)
- [PALETTE](#)
- [RECOMPUTE_DIMENSIONS](#)
- [RENDER_METHOD](#)
- [UPDIR](#)
- [XCOORD_CONV](#)
- [ALL](#)
- [BASELINE](#)
- [CLIP_PLANES](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_WRITE_DISABLE](#)
- [FILL_BACKGROUND](#)
- [FONT](#)
- [KERNING](#)
- [ONGLASS](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [STRINGS](#)
- [VERTICAL_ALIGNMENT](#)
- [XRANGE](#)

- [YCOORD_CONV](#)
- [ZCOORD_CONV](#)
- [YRANGE](#)
- [ZRANGE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrText::Cleanup](#)
- [IDLgrText::GetCTM](#)
- [IDLgrText::GetProperty](#)
- [IDLgrText::Init](#)
- [IDLgrText::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrText Properties

IDLgrText objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrText::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrText::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrText::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALIGNMENT

A floating-point value between 0.0 and 1.0 to indicate the requested horizontal alignment of the text baseline. An alignment of 0.0 (the default) left-justifies the text at the given position; an alignment of 1.0 right-justifies the text, and an alignment of 0.5 centers the text over the given position.

Property Type	FLOAT		
Name String	Horizontal alignment		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

A floating-point value that determines the opacity of the text with respect to the background. Set this property to a value in the range [0.0, 1.0] (1.0 is the default) to draw the text foreground and background with the specified blending factor. A value of 1.0 draws the text opaquely without blending the text with objects already drawn on the destination. Edges of the glyphs are always blended. A value of 0.0 draws no text at all. A value in the middle of the range draws the text semi-transparently, which provides a way of creating labels that are visible while allowing features blocked by the labels to still be seen. This property is used only when the `RENDER_METHOD` in effect is 0 (Texture).

Note

For more detailed information on alpha channel support for IDL Object Graphics, see [“Alpha Channel and Objects”](#) in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

BASELINE

A two- or three-element floating-point vector describing the direction in which the baseline is to be oriented. Use this property in conjunction with the `UPDIR` property to specify the plane on which the text lies. The default `BASELINE` is [1.0,0,0] (i.e., parallel to the x-axis).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Baseline direction		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CHAR_DIMENSIONS

A two-element floating-point vector [*width*, *height*] indicating the dimensions (measured in data units) of a bounding box for each character, to be used when scaling text projected in three dimensions. If either *width* or *height* is zero, the text will be scaled such that if it were positioned halfway between the near and far

clipping planes, it will appear at the point size associated with this text object's font. The default value is [0, 0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Note

If you set the CHAR_DIMENSIONS property to [0,0] (using the SetProperty method), indicating that IDL should calculate the text size, the value (returned by the GetProperty method) will not be updated to reflect the calculated size until you call either the Draw method or the GetTextDimensions method.

For example, if the VIEWPLANE_RECT of the view the text object is being rendered in is set equal to [0,0,10,10] (that is, it spans ten data units in each of the X and Y directions), setting the CHAR_DIMENSIONS property equal to [2, 3] will scale the text such that each character fills 20% of the X range and 30% of the Y range.

This property has no effect if the ONGLASS property is set equal to one.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

CLIP_PLANES

A 4-by-*N* floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any *x*, *y*, or *z* coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the `MAX_NUM_CLIP_PLANES` property of the `GetDeviceInfo` method for the `IDLgrBuffer`, `IDLgrClipboard`, `IDLgrWindow`, and `IDLgrVRML` objects.

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COLOR

The color to be used as the foreground color for the text. The color may be specified as a color lookup table index or as a three-element vector [*red*, *green*, *blue*]. The default is [0, 0, 0].

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ENABLE_FORMATTING

A Boolean value that determines whether the text object should honor embedded Hershey-style formatting codes within the strings. (Formatting codes are described in [Appendix H, “Fonts”](#).) The default is not to honor the formatting codes.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FILL_BACKGROUND

Set this property to zero (the default) to render the text with a transparent bitmap background, allowing graphics behind the text to show through between the glyphs. Set this property to non-zero to draw the text bitmap background with the color specified by the FILL_COLOR property. This property can only be used when RENDER_METHOD is set to 0 (Texture).

Property Type	BOOLEAN		
Name String	Fill background		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

FILL_COLOR

An integer vector that determines whether to use a text background color other than the view color. Set this property to -1 (the default) to specify that the text background should be drawn using the view color. Set this property to a three-element vector [*red*, *green*, *blue*] or color index value to specify that the text bitmap background should be drawn using the specified color. This property is used only when the FILL_BACKGROUND property has a non-zero value and the RENDER_METHOD in effect is 0 (Texture).

In a property sheet, this property appears as a color property

Property Type	COLOR		
Name String	Fill background		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

FONT

An object reference to an instance of an IDLgrFont object class to describe the font to use to draw this string. The default is 12 point Helvetica. See [IDLgrFont](#) for details.

Note

If the default font is in use, retrieving the value of the FONT property (using the GetProperty method) will return a null object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

KERNING

A Boolean value that determines whether to enable kerning while rendering characters. Kerning reduces the amount of space between glyphs if the shape of each glyph allows it, according to the font information stored in the font's file (e.g., AV). Set this property to a non-zero value (the default is zero) to enable kerning. Enabling kerning may not necessarily result in rendering glyphs more closely together because some fonts do not contain the required kerning information. This property is used only when the RENDER_METHOD in effect is 0 (Texture).

Property Type	BOOLEAN		
Name String	Kerning		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LOCATIONS

A floating-point array of one or more two- or three-element vectors specifying the coordinates (measured in data units) used to position the string(s). The default location for each string is [0,0,0]. Each vector is of the form [x, y] or [x, y, z]; if z is not provided, it is assumed to be zero. Each location corresponds to a string in the *String* argument; the number of locations should be equal the number of strings. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Text locations		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ONGLASS

A Boolean value that indicates whether the text should be displayed “on the glass”. The default is projected 3-D text.

Property Type	BOOLEAN		
Name String	On glass		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class) that defines the color palette of this object. This property is only used if the destination device is using the RGB color model. If so, and a color value for the object is specified as a color index value, the palette set by this property is used to translate the color to RGB space. If the PALETTE property on this object is not set, the destination object PALETTE property is used (which defaults to a grayscale ramp).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

RECOMPUTE_DIMENSIONS

An integer value that indicates when this text object's character dimensions (refer to the CHAR_DIMENSIONS property) are to be recomputed automatically:

- 0 = The physical size of the text is affected by model and view transforms. The size of the text in terms of data units is obtained from CHAR_DIMENSIONS. Since the character dimensions are specified in data units, the text will maintain the data space size specified by CHAR_DIMENSIONS as the transforms change. In other words, the physical text size changes along with other primitives. If the value of this property is [0, 0], the text font's point size is used to compute the physical size of the text in terms of data units using the transforms in effect for the first draw. This setting is the default value for this property.
- 1 = The physical size of the text is only affected by model transforms. The CHAR_DIMENSIONS property is ignored. The size of the text is computed from the font's point size the first time it is drawn, and IDL does not try to keep the size of the text constant with respect to changes in the model transforms.
- 2 = The physical size of the text is held constant, even as the model and view change. The CHAR_DIMENSIONS property is ignored and the text is always drawn with a physical size equal to the text font's point size. IDL adjusts its internal text transforms to maintain the physical size of the text.

In a property sheet, this property appears as an enumerated list with the following options:

- Never
- Previous
- Always

Property Type	ENUMLIST		
Name String	Recompute dimensions		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RENDER_METHOD

A Boolean value that determines how the text object will be rendered. Set this property to one of the following values:

- 0 = TEXTURE. IDL renders the text by placing a bitmap representation of a glyph into a texture map and then rendering a polygon with the texture map. The `FILL_BACKGROUND`, `ALPHA_CHANNEL`, and `FILL_COLOR` properties control the drawing of the background portions of the texture map and how the entire texture map is blended into the scene. Leaving these three properties set to their default values produces a result that closely approximates the `TRIANGLES` rendering method. One important difference is that the glyph bitmaps are generated by the FreeType font rendering library, producing glyphs that are more accurately rendered and anti-aliased than those drawn with the `TRIANGLES` method. The `TEXTURE` method cannot be used on indexed color destinations. The text is rendered with the `TRIANGLES` method if the destination color model is indexed.
- 1 = TRIANGLES. IDL renders the text by tessellating the glyph outline into a set of small triangles that are then drawn to produce the solid glyph. IDL also draws a blended line around the edge of the glyph to approximate anti-aliasing. This setting forces IDL to use the process it used as the default before IDL 6.0.

In a property sheet, this property appears as an enumerated list with the following options:

- Texture
- Triangles

Property Type	ENUMLIST		
Name String	Render method		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

STRINGS

The string (or vector of strings) associated with the text object. This property is the same as the *String* argument described for “IDLgrText::Init” on page 4053.

If the number of strings matches the number of locations (as specified by the LOCATIONS property), the existing locations are used.

Note

If the number of strings *does not* match the number of locations, the number of locations is modified to match the number of strings, and the location value for each string is reset to [0,0,0].

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Text strings		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

UPDIR

A two- (or three-) element floating-point vector describing the vertical direction for the string. The *upward direction* is the direction defined by a vector pointing from the origin to the point specified. Use this property in conjunction with the BASELINE property to specify the plane on which the text lies; the direction specified by UPDIR should be orthogonal to the direction specified by BASELINE. The default UPDIR is [0.0, 1.0, 0.0] (i.e., parallel to the *Y* axis).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Up direction		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VERTICAL_ALIGNMENT

A floating-point value between 0.0 and 1.0 to indicate the requested vertical alignment of the text. An alignment of 0.0 (the default) bottom-justifies the text at the given location; an alignment of 1.0 top-justifies the text at the given location.

Property Type	FLOAT		
Name String	Vertical alignment		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element floating-point vector of the form $[xmin, xmax]$ that specifies the range of x data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is $[0.0, 1.0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is $[0.0, 1.0]$. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZRANGE

A two-element floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object. IDL maintains and returns this property in double-precision floating-point.

Note

Until the text is drawn to the destination object, the $[XYZ]$ RANGE properties will only report the locations of the text. Use the `GetTextDimensions` method of the destination object to get the data dimensions of the text prior to a draw operation.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

IDLgrText::Cleanup

The IDLgrText::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrText::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrText::GetCTM

The IDLgrText::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrText::]GetCTM( [, DESTINATION=objref] [, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the text object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrText::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrText::GetProperty

The IDLgrText::GetProperty procedure method retrieves the value of a property or group of properties for the text.

Syntax

Obj->[IDLgrText::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrText Properties](#)” on page 4034 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrText::Init

The IDLgrText::Init function method initializes the text object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDLgrText' [, *String or vector of strings*] [, *PROPERTY=value*])

or

Result = *Obj*->[IDLgrText::]Init([*String or vector of strings*] [, *PROPERTY=value*])
(Only in a subclass' Init method.)

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

String

The string (or vector of strings) to be created. If this argument is not a string, it is converted prior to creation of the text object using the default formatting rules.

Note

Strings have a default location of [0,0,0]. Use the LOCATIONS property to provide a different location for each string.

Keywords

Any property listed under [“IDLgrText Properties”](#) on page 4034 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES keyword added: 5.6

IDLgrText:: SetProperty

The IDLgrText::SetProperty procedure method sets the value of a property or group of properties for the text.

Syntax

Obj->[IDLgrText::]SetProperty[, *PROPERTY*=*value*]

Arguments

None

Keywords

Any property listed under “[IDLgrText Properties](#)” on page 4034 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrView

A view object represents a rectangular area in which graphics objects are drawn. It is a container for objects of the [IDLgrModel](#) class.

Superclasses

[IDL_Container](#)

[IDLitComponent](#)

Creation

See “[IDLgrView::Init](#)” on page 4070.

Properties

Objects of this class have the following properties. See “[IDLgrView Properties](#)” on page 4058 for details on individual properties.

- [ALL](#)
- [COLOR](#)
- [DEPTH_CUE](#)
- [DIMENSIONS](#)
- [DOUBLE](#)
- [EYE](#)
- [HIDE](#)
- [LOCATION](#)
- [PARENT](#)
- [PROJECTION](#)
- [REGISTER_PROPERTIES](#)
- [TRANSPARENT](#)
- [UNITS](#)
- [VIEWPLANE_RECT](#)
- [ZCLIP](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrView::Add](#)
- [IDLgrView::Cleanup](#)
- [IDLgrView::GetByName](#)
- [IDLgrView::GetProperty](#)
- [IDLgrView::Init](#)
- [IDLgrView::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrView Properties

IDLgrView objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via `IDLgrView::GetProperty`. Properties with the word “Yes” in the “Init” column of the property table can be set via `IDLgrView::Init`. Properties with the word “Yes” in the “Set” column in the property table can be set via `IDLgrView::SetProperty`.

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR

The color for the view. This is the color to which the view area will be erased before its contents are drawn. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default is [255, 255, 255] (white).

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_CUE

A two-element floating-point array [*zbright*, *zdim*] specifying the near and far Z planes between which depth cueing is in effect. Depth cueing is only honored when drawing to a destination object that uses the RGB color model.

Depth cueing causes an object to appear to fade into the background color of the view object with changes in depth. If the depth of an object is further than *zdim* (that is, if the object's location in the Z direction is farther from the origin than the value specified by *zdim*), the object will be painted in the background color. Similarly, if the object is closer than the value of *zbright*, the object will appear in its "normal" color. Anywhere in-between, the object will be a blend of the background color and the object color. For example, if the DEPTH_CUE property is set to [-1,1], an object at the depth of 0.0 will appear as a 50% blend of the object color and the view color.

The relationship between Z_{bright} and Z_{dim} determines the result of the rendering:

- $Z_{bright} < Z_{dim}$: Rendering darkens with depth.
- $Z_{bright} > Z_{dim}$: Rendering brightens with depth.
- $Z_{bright} = Z_{dim}$: Disables depth cueing.

You can disable depth cueing by setting $z_{bright} = z_{dim}$. The default is [0.0, 0.0].

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Depth cue		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DIMENSIONS

A two-element integer vector of the form [*width*, *height*] specifying the dimensions of the viewport (the rectangle in which models are displayed on a graphics destination). By default, the viewport dimensions are set to [0, 0], which indicates that it will match the dimensions of the graphics destination to which it is drawn. The dimensions are measured in the units specified by the UNITS property.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DOUBLE

A Boolean value that controls the precision used for rendering the entire contents of the view. If set, IDL calculates the transformations used for the modeling and view transforms using double-precision floating-point arithmetic. This allows the values specified for the VIEWPLANE_RECT, modeling transforms in IDLgrModel objects, and coordinate data in atomic graphic objects to be used as double-precision before mapping to device coordinates.

Note

If this property is set to 0, IDL uses single-precision floating-point arithmetic for these values, which can cause loss of significance and incorrect rendering of data. Setting this property to 1 may impact graphics performance and should only be used when handling data requiring double precision.

Property Type	BOOLEAN		
Name String	Double precision		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

EYE

A floating-point value that specifies the distance from the eyepoint to the viewplane (Z=0). The default is 4.0. The eyepoint is always centered within the viewplane rectangle. (That is, if the VIEWPLANE_RECT property is set equal to [0,0,1,1], the eyepoint will be at X=0.5, Y=0.5.) IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	FLOAT		
Name String	Eye distance		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LOCATION

A two-element floating-point vector of the form $[x, y]$ specifying the position of the lower left corner of the view. The default is $[0, 0]$, measured in device units.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Viewport location		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set:	Init:	Registered: No

PROJECTION

An integer value indicating the type of projection to use within this view. All models displayed within this view will be projected using this type of projection. Valid values are described below.

- 1 = Orthogonal projection (default).
- 2 = Perspective: Indicates that all models are projected toward the eye (located at the origin), which is the apex of the viewing frustum. With a perspective

projection, models that are farther away from the eye will appear smaller in the view than models that are nearer to the eye.

In a property sheet, this property appears as an enumerated list with the following options:

- None
- Orthogonal
- Perspective

Property Type	ENUMLIST		
Name String	Projection		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

TRANSPARENT

A Boolean value that determines whether to disable the viewport erase, making the viewport transparent. If this value is false, the viewport is erased and the depth buffer initialized. If this value is true the depth buffer is not initialized.

Property Type	BOOLEAN		
Name String	Transparent		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

UNITS

An integer value that specifies the units of measure for this view. Valid values are:

- 0 = Device (default)
- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to the graphics destination's *rect*.

Note

If you set the UNITS property (using the SetProperty method) of a view without also setting the LOCATION and DIMENSIONS properties, IDL will use the existing size and location values in the new units, *without conversion*. This means that if your view's location and dimensions were previously measured in centimeters, and you change the value of UNITS to 1 (measurement in inches), the actual size of the view object will change.

In a property sheet, this property appears as an enumerated list with the following options:

- Device
- Inches
- Centimeters
- Normalization

Property Type	ENUMLIST		
Name String	Units		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VIEWPLANE_RECT

A four-element floating-point vector of the form $[x, y, width, height]$ to describe the bounds in x and y of the view volume. Objects within the view volume are projected into the viewport. These values are measured in normalized space. The default is $[-1.0, -1.0, 2.0, 2.0]$ IDL converts, maintains, and returns this data as double-precision floating-point.

Note

The z bounds of the view volume are set via the ZCLIP property. The viewplane rectangle is always located at Z=0.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Viewplane rectangle		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ZCLIP

A two element floating-point vector representing the near and far clipping planes to be applied to the objects in this view. The vector should take the form [*near*, *far*]. By default, these values are [1, -1]. IDL converts, maintains, and returns this data as double-precision floating-point.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Z clipping		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

IDLgrView::Add

The IDLgrView::Add procedure method adds a child to this view.

Syntax

Obj->[IDLgrView::]Add, *Model* [, POSITION=*index*]

Arguments

Model

An instance of the [IDLgrModel](#) object class.

Keywords

POSITION

Set this keyword equal to the zero-based index of the position within the container at which the new object should be placed.

Version History

Introduced 5.0

IDLgrView::Cleanup

The IDLgrView::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrView::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced 5.0

IDLgrView::GetByName

The IDLgrView::GetByName function method finds contained objects by name. If the named object is not found, the GetByName function returns a null object reference.

Note

The GetByName function does *not* perform a recursive search through the object hierarchy. If a fully qualified object name is not specified, only the contents of the current container object are inspected for the named object.

Syntax

Result = *Obj*->[IDLgrView::]GetByName(*Name*)

Return Value

Returns contained objects by name. If the named object is not found, the GetByName function returns a null object reference.

Arguments

Name

A string containing the name of the object to be returned.

Object naming syntax is very much like the syntax of a UNIX filesystem. Objects contained by other objects can include the name of their parent object; this allows you to create a fully qualified name specification. For example, if `object1` contains `object2`, which in turn contains `object3`, the string specifying the fully qualified object name of `object3` would be `'object1/object2/object3'`.

Object names are specified relative to the object on which the GetByName method is called. If used at the beginning of the name string, the `/` character represents the top of an object hierarchy. The string `'..'` represents the object one level “up” in the hierarchy.

Keywords

None

Version History

Introduced 5.0

IDLgrView::GetProperty

The IDLgrView::GetProperty procedure method retrieves the value of the property or group of properties for the view.

Syntax

Obj->[IDLgrView::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrView Properties](#)” on page 4058 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced 5.0

IDLgrView::Init

The IDLgrView::Init function method initializes the view object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrView' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrView::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrView Properties](#)” on page 4058 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced 5.0

IDLgrView:: SetProperty

The IDLgrView::SetProperty procedure method sets the value of the property or group of properties for the view.

Syntax

Obj->[IDLgrView::]SetProperty[, *PROPERTY*=*value*]

Arguments

None

Keywords

Any property listed under “[IDLgrView Properties](#)” on page 4058 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced 5.0

IDLgrViewgroup

The IDLgrViewgroup object is a simple container object, very similar to the IDLgrScene object. It contains one or more IDLgrView objects and an IDLgrScene can contain one or more of these objects. This object is special in that it can also contain objects which do not have a Draw method (e.g. IDLgrPattern and IDLgrFont). An IDLgrViewgroup object cannot be returned by a call to the IDLgrWindow::Select method or the IDLgrBuffer::Select method.

Superclasses

[IDL_Container](#)

[IDLitComponent](#)

Creation

See “[IDLgrViewgroup::Init](#)” on page 4082.

Properties

Objects of this class have the following properties. See “[IDLgrViewgroup Properties](#)” on page 4075 for details on individual properties.

- [ALL](#)
- [HIDE](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [IDLgrViewgroup::Add](#)
- [IDLgrViewgroup::Cleanup](#)
- [IDLgrViewgroup::GetByName](#)
- [IDLgrViewgroup::GetProperty](#)

- [IDLgrViewgroup::Init](#)
- [IDLgrViewgroup::SetProperty](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrViewgroup Properties

IDLgrViewgroup objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrViewgroup::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrViewgroup::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrViewgroup::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the retrievable properties associated with this object.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

HIDE

A Boolean value to indicate whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

PARENT

An object reference to the object that contains this viewgroup.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLgrViewgroup::Add

The IDLgrViewgroup::Add function method verifies that the added item is not an instance of the IDLgrScene or IDLgrViewgroup object. If it is not, IDLgrViewgroup:Add adds the object to the specified viewgroup.

Syntax

Obj->[IDLgrViewgroup::]Add, *Object* [, POSITION=*index*]

Arguments

Object

An instance of an object or a list of objects. Objects which subclass IDLgrScene or IDLgrViewgroup can not be added (avoiding circularity constraints). All other objects are allowed.

Keywords

POSITION

Set this keyword equal to the zero-based index of the position within the container at which the new object should be placed.

Version History

Introduced: 5.1

IDLgrViewgroup::Cleanup

The IDLgrViewgroup::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrViewgroup::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrViewgroup::GetByName

The IDLgrViewgroup::GetByName function method finds contained objects by name. If the named object is not found, the GetByName function returns a null object reference.

Note

The GetByName function does *not* perform a recursive search through the object hierarchy. If a fully qualified object name is not specified, only the contents of the current container object are inspected for the named object.

Syntax

Result = *Obj*->[IDLgrViewgroup::]GetByName(*Name*)

Return Value

Returns contained objects by name. If the named object is not found, the GetByName function returns a null object reference.

Arguments

Name

A string containing the name of the object to be returned.

Object naming syntax is very much like the syntax of a UNIX filesystem. Objects contained by other objects can include the name of their parent object; this allows you to create a fully qualified name specification. For example, if `object1` contains `object2`, which in turn contains `object3`, the string specifying the fully qualified object name of `object3` would be `'object1/object2/object3'`.

Object names are specified relative to the object on which the GetByName method is called. If used at the beginning of the name string, the `/` character represents the top of an object hierarchy. The string `'..'` represents the object one level “up” in the hierarchy.

Keywords

None

Version History

Introduced: 5.1

IDLgrViewgroup::GetProperty

The IDLgrViewgroup::GetProperty procedure method retrieves the value of a property or group of properties for the viewgroup object.

Syntax

Obj->[IDLgrViewgroup::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under [“IDLgrViewgroup Properties”](#) on page 4075 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrViewgroup::Init

The IDLgrViewgroup::Init function method initializes the viewgroup object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrViewgroup' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrViewgroup::]Init([, PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrViewgroup Properties](#)” on page 4075 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

IDLgrViewgroup:: SetProperty

The IDLgrViewgroup::SetProperty procedure method sets the value of a property or group of properties for the viewgroup.

Syntax

Obj->[IDLgrViewgroup::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under [“IDLgrViewgroup Properties”](#) on page 4075 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrVolume

A volume object represents a mapping from a three-dimensional array of data to a three-dimensional array of voxel colors, which, when drawn, are projected to two dimensions.

An IDLgrVolume object is an *atomic graphic object*; it is one of the basic drawable elements of the IDL Object Graphics system, and it is not a container for other objects.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrVolume::Init](#)” on page 4109.

Properties

Objects of this class have the following properties. See “[IDLgrVolume Properties](#)” on page 4087 for details on individual properties.

- [ALL](#)
- [ALPHA_CHANNEL](#)
- [AMBIENT](#)
- [BOUNDS](#)
- [CLIP_PLANES](#)
- [COMPOSITE_FUNCTION](#)
- [DATA0](#)
- [DATA1](#)
- [DATA2](#)
- [DATA3](#)
- [DEPTH_CUE](#)
- [DEPTH_TEST_DISABLE](#)
- [DEPTH_TEST_FUNCTION](#)
- [DEPTH_WRITE_DISABLE](#)
- [HIDE](#)
- [HINTS](#)
- [INTERPOLATE](#)
- [LIGHTING_MODEL](#)
- [NO_COPY](#)
- [OPACITY_TABLE0](#)
- [OPACITY_TABLE1](#)
- [PARENT](#)
- [REGISTER_PROPERTIES](#)
- [RENDER_STEP](#)

- `RGB_TABLE0`
- `TWO_SIDED`
- `VOLUME_SELECT`
- `XRANGE`
- `YRANGE`
- `ZCOORD_CONV`
- `RGB_TABLE1`
- `VALID_DATA`
- `XCOORD_CONV`
- `YCOORD_CONV`
- `ZBUFFER`
- `ZERO_OPACITY_SKIP`

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrVolume::Cleanup`
- `IDLgrVolume::ComputeBounds`
- `IDLgrVolume::GetCTM`
- `IDLgrVolume::GetProperty`
- `IDLgrVolume::Init`
- `IDLgrVolume::PickVoxel`
- `IDLgrVolume::SetProperty`

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrVolume Properties

IDLgrVolume objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrVolume::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrVolume::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrVolume::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ALPHA_CHANNEL

Apply to the volume only when the volume is rendered with a destination rendering quality setting of Low (0). In the case, the volume is rendered as a wireframe cube. Set this property to a floating-point value in the range [0.0, 1.0] to specify the opacity of the wireframe cube. The default value of 1.0 causes IDL to draw the wireframe cube completely opaque. If the value of this property is less than 1.0, then the pixels of the wireframe cube are blended with the pixels already on the screen, where the color of the wireframe cube is multiplied by the alpha value and the pixels already on the screen are multiplied by 1.0-alpha.

Because an object can only be blended with objects already drawn on the screen, the drawing order of the objects must be considered carefully in order to obtain the desired results.

This property has no effect on devices using indexed color mode.

Note

For more detailed information on alpha channel support for IDL Object Graphics, see “Alpha Channel and Objects” in Chapter 23 of the *Using IDL* manual.

Property Type	Float-point value		
Name String	Transparency		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

AMBIENT

The color and intensity of the volume’s base ambient lighting. Color is specified as a three-element vector [*red*, *green*, *blue*]. The default is [255, 255, 255]. AMBIENT is applicable only when LIGHTING_MODEL is set.

In a property sheet, this property appears as a color property.

Property Type	COLOR		
Name String	Ambient color		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

BOUNDS

A six-element floating-point vector of the form [x_{min} , y_{min} , z_{min} , x_{max} , y_{max} , z_{max}], which represents the sub-volume to be rendered. .

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Subvolume bounds		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

CLIP_PLANES

A 4-by- N floating-point array that specifies the coefficients of the clipping planes to be applied to this object. The four coefficients for each clipping plane are of the form [A, B, C, D], where $Ax + By + Cz + D = 0$. Portions of this object that fall in the half

space $Ax + By + Cz + D > 0$ will be clipped. By default, the value of this property is a scalar (-1) indicating that no clipping planes are to be applied.

Note

The clipping planes specified via this property are applied in addition to the near and far clipping planes associated with the IDLgrView in which this object appears.

Note

Clipping planes are applied in the data space of this object (prior to the application of any x , y , or z coordinate conversion).

Note

To determine the maximum number of clipping planes supported by the device, use the MAX_NUM_CLIP_PLANES property of the GetDeviceInfo method for the IDLgrBuffer, IDLgrClipboard, IDLgrWindow, and IDLgrVRML objects.

Note

Clipping planes are equivalent to the obsolete cutting planes. If defined, CUTTING_PLANES will be applied first, then the CLIP_PLANES (until a maximum number of planes is reached).

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

COMPOSITE_FUNCTION

An integer value that determines the composite function to be used to measure the value of a pixel on the viewing plane by analyzing the voxels falling along the corresponding ray. Specify one of the following compositing functions:

- 0 = Alpha (default): Alpha-blending. The recursive equation $dest' = src * sr calpha + dest * (1 - sr calpha)$ is used to compute the final pixel color.
- 1 = MIP: Maximum intensity projection. The value of each pixel on the viewing plane is set to the brightest voxel, as determined by its opacity. The most opaque voxel's color appropriation is then reflected by the pixel on the viewing plane.

- 2 = Alpha sum: Alpha-blending. The recursive equation

$$\text{dest}' = \text{src} + \text{dest} * (1 - \text{srcalpha})$$

is used to compute the final pixel color. This equation assumes that the color tables have been pre-multiplied by the opacity tables. The accumulated values can be no greater than 255.

- 3 = Average: Average-intensity projection. The resulting image is the average of all voxels along the corresponding ray. Disables lighting and only works with grey scale palettes. Will not work with four-channel volumes.

In a property sheet, this property appears as an enumerated list with the following options:

- Alpha blending
- Maximum intensity projection
- Alpha sum
- Average intensity

Property Type	ENUMLIST		
Name String	Composite function		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DATA0

A three-element array of any type of the format (d_x, d_y, d_z) , which specifies a data volume. Setting this property is the same as including the *vol₀* argument at creation time. If the data volume dimensions do not match those of any pre-existing data in DATA1, DATA2, or DATA3, all existing data is removed from the object.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DATA1

A three-dimensional array of any type of the format (d_x, d_y, d_z) , which specifies a data volume. Setting this property is the same as including the *vol₁* argument at creation

time. If the data volume dimensions do not match those of any pre-existing data in DATA0, DATA2, or DATA3, all existing data is removed from the object.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DATA2

A three-dimensional array of any type of the format (d_x, d_y, d_z) , which specifies a data volume. Setting this property is the same as including the vol_2 argument at creation time. If the data volume dimensions do not match those of any pre-existing data in DATA0, DATA1, or DATA3, all existing data is removed from the object.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DATA3

A three-dimensional array of any type of the format (d_x, d_y, d_z) , which specifies a data volume. Setting this property is the same as including the vol_3 argument at creation time. If the data volume dimensions do not match those of any pre-existing data in DATA0, DATA1, or DATA2, all existing data is removed from the object.

Note

DATA0, DATA1, DATA2, and DATA3 sizes are dynamic.

Property Type	Array of any type		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

DEPTH_CUE

A two-element floating-point array [$zbright, zdim$] specifying the near and far Z planes between which depth cueing is in effect. Depth cueing is only honored when drawing to a destination object that uses the RGB color model.

Depth cueing causes an object to appear to fade into the background color of the view object with changes in depth. If the depth of an object is further than z_{dim} (that is, if the object's location in the Z direction is farther from the origin than the value specified by z_{dim}), the object will be painted in the background color.

Similarly, if the object is closer than the value of z_{bright} , the object will appear in its "normal" color. Anywhere in-between, the object will be a blend of the background color and the object color. For example, if the DEPTH_CUE property is set to [-1,1], an object at the depth of 0.0 will appear as a 50% blend of the object color and the view color.

The relationship between Z_{bright} and Z_{dim} determines the result of the rendering:

- $Z_{bright} < Z_{dim}$: Rendering darkens with depth.
- $Z_{bright} > Z_{dim}$: Rendering brightens with depth.
- $Z_{bright} = Z_{dim}$: Disables depth cueing.

You can disable depth cueing by setting $z_{bright} = z_{dim}$. The default is [0.0, 0.0].

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Depth cue range		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_DISABLE

An integer value that determines whether depth testing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth testing. A model may also enable or disable depth testing.
- Set this property to 1 to explicitly disable depth buffer testing while drawing this object.
- Set this property to 2 to explicitly enable depth testing for this object.

Disabling depth testing allows an object to draw itself on top of other objects already on the screen, even if the object is located behind them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_TEST_FUNCTION

An integer value that determines the depth test function. Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always sets a depth test function of LESS. A model may also set a depth test function value. The graphics device compares the object's depth at a particular pixel location with the depth stored in the depth buffer at that same pixel location. If the comparison test passes, the object's pixel is drawn at that location on the screen and the depth buffer is updated (if depth writing is enabled).

Set this property to any of the following values to use the desired function while rendering this object.

- 0 = INHERIT - use value from parent model or view.
- 1 = NEVER - never passes.
- 2 = LESS - passes if the object's depth is less than the depth buffer's value.
- 3 = EQUAL - passes if the object's depth is equal to the depth buffer's value.
- 4 = LESS OR EQUAL - passes if the object's depth is less than or equal to the depth buffer's value.
- 5 = GREATER - passes if the object's depth is greater than or equal to the depth buffer's value.
- 6 = NOT EQUAL - passes if the object's depth is not equal to the depth buffer's value.
- 7 = GREATER OR EQUAL - passes if the object's depth is greater than or equal to the depth buffer's value.
- 8 = ALWAYS - always passes

Less means closer to the viewer.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Test Enable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DEPTH_WRITE_DISABLE

An integer value that determines whether depth writing is disabled.

- Set this property to 0 (the default) to inherit the value set by the parent model or view. The parent view always enables depth writing. A model may also enable or disable depth writing.
- Set this property to 1 to explicitly disable depth buffer writing while rendering this object.
- Set this property to 2 to explicitly enable depth writing for this object.

Disabling depth writing allows an object to be overdrawn by other objects, even if the object is located in front of them.

This property is registered as an enumerated list, but it is hidden by default.

Property Type	ENUMLIST		
Name String	Depth Write Disable		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HIDE

A Boolean value indicating whether this object should be drawn:

- 0 = Draw graphic (the default)
- 1 = Do not draw graphic

In a property sheet, this property appears as an enumerated list with the following options:

- True = Draw graphic (the default)
- False = Do not draw graphic.

Property Type	ENUMLIST		
Name String	Show		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

HINTS

An integer value that specifies one of the following acceleration hints:

- 0 = Disables all acceleration hints (default).
- 1 = Enables Euclidean distance map (EDM) acceleration. This option generates a volume map containing the distance from any voxel to the nearest non-zero opacity voxel. The map is used to speed ray casting by allowing the ray to jump over open spaces. It is most useful with sparse volumes. After setting the EDM hint, the draw operation generates the volume map; this process can take some time. Subsequent draw operations will reuse the generated map and may be much faster, depending on the volume's sparseness. A new map is not automatically generated to match changes in opacity tables or volume data (for performance reasons). The user may force recomputation of the EDM map by setting the HINTS property to 1 again.
- 2 = Enables the use of multiple CPUs for volume rendering if the platforms used support such use. If HINTS is set to 2, IDL will use all the available (up to 8) CPUs to render portions of the volume in parallel.
- 3 = Selects the two acceleration options described above.

In a property sheet, this property appears as an enumerated list with the following options:

- Disable
- Euclidean distance map (EDM)
- Multiple CPU
- EDM and Multiple CPU

Property Type	ENUMLIST		
Name String	Acceleration hints		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

INTERPOLATE

A Boolean value that determines whether to use Trilinear interpolation to determine the data value for each step on a ray. Setting this property improves the quality of images produced, at the cost of more computing time, especially when the volume has low resolution with respect to the size of the viewing plane. Nearest neighbor sampling is used by default.

In a property sheet, this property appears as an enumerated list with the following options:

- Nearest neighbor
- Trilinear

Property Type	ENUMLIST		
Name String	Interpolation		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

LIGHTING_MODEL

A Boolean value that determines whether to use the current lighting model during rendering in conjunction with a local gradient evaluation.

Note

Only DIRECTIONAL light sources are honored by the volume object. Because normals must be computed for all voxels in a lighted view, enabling light sources increases the rendering time.

Property Type	BOOLEAN		
Name String	Use lighting		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

NO_COPY

A Boolean value that determines whether to relocate volume data from the input variables to the volume object, leaving the input variables undefined. Only the

DATA0 property and the *vol0* argument are affected. If this property is omitted, the input volume data will be duplicated and a copy will be stored in the object.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

OPACITY_TABLE0

A 256-element byte array that specifies the opacity table for DATA0. The default table is the linear ramp.

Property Type	Byte array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

OPACITY_TABLE1

A 256-element byte array that specifies the opacity table for DATA1. The default table is the linear ramp. This table is used only when VOLUME_SELECT is set equal to 1.

Property Type	Byte array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

PARENT

An object reference to the object that contains this object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RENDER_STEP

A three-element floating-point vector of the form $[x, y, z]$ that specifies the stepping factor through the voxel matrix.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	XYZ render step		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

RGB_TABLE0

A 256 x 3-element byte array that specifies the RGB color table for DATA0. The default table is the linear ramp. .

Property Type	Byte array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

RGB_TABLE1

A 256 x 3-element byte array that specifies the RGB color table for DATA1. The default table is the linear ramp. This table is used only when VOLUME_SELECT is set equal to 1.

Property Type	Byte array		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

TWO_SIDED

A Boolean value that determines whether the lighting model uses a two-sided voxel gradient. The two-sided gradient is different from the one-sided gradient (default) in that the absolute value of the inner product of the light direction and the surface gradient is used instead of clamping to 0.0 for negative values. .

In a property sheet, this property appears as an enumerated list with the following options:

- One-sided
- Two-sided

Property Type	ENUMLIST		
Name String	Voxel gradient		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VALID_DATA

An integer array (one per volume, DATA0, DATA1, etc.) which have the value 1 if volume data has been loaded for that volume and 0 if that volume data is currently undefined.

Property Type	Integer array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

VOLUME_SELECT

An integer value that selects the form of the volume to be rendered. The `VOLUME_SELECT` property is used to modify the `src` and `srcalpha` parameters for the `COMPOSITE_FUNCTION` property.

- 0 = render voxels from the 8bit `DATA0` volume (the default)

```
src = RGB_TABLE0[DATA0]
srcalpha = OPACITY_TABLE0[DATA0]
```

- 1 = render voxels formed by modulating the RGBA components from `DATA0` and `DATA1` (after `RGB` and `OPACITY` table lookups).

```
src = (RGB_TABLE0[DATA0]*RGB_TABLE1[DATA1])/256
srcalpha=(OPACITY_TABLE0[DATA0]*OPACITY_TABLE1[DATA1])/256
```

- 2 = render voxels formed using a byte from `DATA0` (red), `DATA1` (green), `DATA2`(blue) and `DATA3`(alpha). The keywords `OPACITY_TABLE0` and `RGB_TABLE0`, described above, are used to indirect the data from each volume before forming the RGBA pixel.

```
src=(RGB_TABLE[DATA0,0],RGB_TABLE[DATA1,1],RGB_TABLE[DATA2,2])/256
srcalpha = (OPACITY_TABLE0[DATA3])/256.
```

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert X coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedX} = s_0 + s_1 * \text{DataX}$$

Recommended values are:

$$[(-X_{min})/(X_{max}-X_{min}), 1/(X_{max}-X_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

XRANGE

A two-element double-precision floating-point vector of the form [x_{min} , x_{max}] that specifies the range of x data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

YCOORD_CONV

A floating-point vector, [s_0 , s_1], of scaling factors used to convert Y coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedY} = s_0 + s_1 * \text{DataY}$$

Recommended values are:

$$[(-Y_{min})/(Y_{max}-Y_{min}), 1/(Y_{max}-Y_{min})]$$

The default is [0.0, 1.0]. IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

YRANGE

A two-element double-precision floating-point vector of the form $[ymin, ymax]$ that specifies the range of y data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZBUFFER

A Boolean value that determines whether to clip the rendering to the current Z-buffer and then update the buffer. The default is to not modify the current Z-buffer.

Property Type	BOOLEAN		
Name String	Update Z buffer		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ZCOORD_CONV

A floating-point vector, $[s_0, s_1]$, of scaling factors used to convert Z coordinates from data units to normalized units. The formula for the conversion is as follows:

$$\text{NormalizedZ} = s_0 + s_1 * \text{DataZ}$$

Recommended values are:

$$[(-Z_{min})/(Z_{max}-Z_{min}), 1/(Z_{max}-Z_{min})]$$

The default is [0.0, 1.0]IDL converts, maintains, and returns this data as double-precision floating-point.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZERO_OPACITY_SKIP

A Boolean value that determines whether to skip voxels with an opacity of 0. This property can increase the output contrast of MIP (MAXIMUM_INTENSITY) projections by allowing the background to show through. If this property is set,

voxels with an opacity of zero will not modify the Z-buffer. The default (not setting the property) continues to render voxels with an opacity of zero.

Property Type	BOOLEAN		
Name String	Skip zero opacity		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ZRANGE

A two-element double-precision floating-point vector of the form $[zmin, zmax]$ that specifies the range of z data coordinates covered by the graphic object.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Obsolete Properties

The following properties are obsolete:

- CUTTING_PLANES

For information on obsolete keywords, see [Appendix I, “Obsolete Features”](#).

Version History

Obsoleted CUTTING_PLANES: 6.0

Added ALPHA_CHANNEL: 6.1

IDLgrVolume::Cleanup

The IDLgrVolume::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrVolume::]Cleanup (*Only in subclass' Cleanup method.*)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrVolume::ComputeBounds

The IDLgrVolume::ComputeBounds procedure method computes the smallest bounding box that contains all voxels whose opacity lookup is greater than a given opacity value. The BOUNDS property is updated to the computed bounding box.

Syntax

```
Obj->[IDLgrVolume::]ComputeBounds [, OPACITY=value] [, /RESET]  
[, VOLUMES=int array]
```

Arguments

None

Keywords

OPACITY

Set this keyword to the opacity value to be used to determine which voxels are included within the bounding box. All voxels whose opacity lookup is greater than this value will be included. The default value is zero.

RESET

Set this keyword to cause the BOUNDS keyword of IDLgrVolume::Init to be reset to contain the entire volume.

VOLUMES

Set this keyword to an array of integers which select which volumes to consider when computing the bounding box. A non-zero value selects a volume to be searched. The default is to search all loaded volumes. For example: VOLUMES=[0,1] will cause ComputeBounds to search only the volume loaded in DATA1.

Version History

Introduced: 5.0

IDLgrVolume::GetCTM

The IDLgrVolume::GetCTM function method returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Syntax

```
Result = Obj->[IDLgrVolume::]GetCTM( [, DESTINATION=objref]  
[, PATH=objref(s)] [, TOP=objref to IDLgrModel object] )
```

Return Value

Returns the 4-by-4 double-precision floating-point graphics transform matrix from the current object upward through the graphics tree.

Arguments

None

Keywords

DESTINATION

Set this keyword to the object reference of a destination object to specify that the projection matrix for the View object in the current tree be included in the returned transformation matrix. The resulting matrix will transform a point in the data space of the object on which the GetCTM method is called into a normalized coordinate system (-1 to +1 in X, Y, and Z), relative to the View object that contains the volume object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the transformation matrix. Each path object reference specified with this keyword must contain an alias. The transformation matrix is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If IDLgrVolume::GetCTM is called from within a Draw method, with the DESTINATION keyword set and the PATH keyword not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

TOP

Set this keyword equal to the object reference to an [IDLgrModel](#) object to specify that the returned matrix accumulate from the object on which the `GetCTM` method is called up to but not including the specified model object.

Version History

Introduced: 5.0

IDLgrVolume::GetProperty

The IDLgrVolume::GetProperty procedure method retrieves the value of a property or group of properties for the volume.

Syntax

Obj->[IDLgrVolume::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrVolume Properties](#)” on page 4087 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrVolume::Init

The IDLgrVolume::Init function method initializes the volume object. At least one volume must be specified, via arguments or keywords.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrVolume' [, vol0 [, vol1 [, vol2 [, vol3]]]]
      [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrVolume::]Init([vol0 [, vol1 [, vol2 [, vol3]]]]
      [, PROPERTY=value]) (Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

vol₀

A three-dimensional array (d_x, d_y, d_z) which specifies a data volume.

vol₁

A three-dimensional array (d_x, d_y, d_z) which specifies a data volume.

vol₂

A three-dimensional array (d_x, d_y, d_z) which specifies a data volume.

vol₃

A three-dimensional array (d_x, d_y, d_z) which specifies a data volume.

Note

If two or more of the above arguments are specified, they must have matching dimensions.

Keywords

Any property listed under [“IDLgrVolume Properties”](#) on page 4087 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

CLIP_PLANES keyword added: 5.6

IDLgrVolume::PickVoxel

The IDLgrVolume::PickVoxel function method computes the coordinates of the voxel projected to a location specified by the 2-D device coordinates point, $[x_i, y_i]$, and the current Z-buffer. The function returns the volume indices as a vector of three long integers. If the selected point is not within the volume, this function returns $[-1,-1,-1]$.

Syntax

Result = *Obj*->[IDLgrVolume::]PickVoxel (*Win*, *View*, *Point* [, **PATH**=*objref(s)*])

Return Value

Returns the volume indices as a vector of three long integers. If the selected point is not within the volume, this function returns $[-1,-1,-1]$.

Arguments

Win

The [IDLgrWindow](#) object from which the Z-buffer is to be used.

View

The IDLgrView object that contains the volume.

Point

The $[x, y]$ viewport coordinates of the point chosen.

Keywords

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a voxel coordinate. Each path object reference specified with this keyword must contain an alias. The voxel coordinate is computed for the version of the object falling within the specified path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in IDLgrModel::Add.

Version History

Introduced: 5.0

IDLgrVolume:: SetProperty

The IDLgrVolume::SetProperty procedure method sets the value of a property or group of properties for the volume.

Syntax

Obj->[IDLgrVolume::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrVolume Properties](#)” on page 4087 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrVRML

The IDLgrVRML object allows you to save the contents of an Object Graphics hierarchy into a VRML 2.0 format file. The graphics tree can only contain a single view due to limitations in the VRML specification. The resulting VRML file is interactive and allows you to explore the geometry interactively using a VRML browser.

Note

Objects or subclasses of this type can not be saved or restored.

Aspect ratios are difficult to duplicate as they can be browser dependent. The object is limited to the primitives supported by VRML. Texture maps (and images) will be inlined into the output file. While this will generate large VRML files, the files are fully self-contained.

Several entities cannot be translated perfectly. These include:

IDLgrImage objects

Rotation and Z buffer behavior are not completely supported. Image objects will be converted into texture mapped polygons. BLEND_FUNCTION is not completely supported (only binary srcAlpha,1-srcAlpha) This function is applied automatically if an alpha channel is present. It is also very browser dependent. Channel masks are not supported.

IDLgrPolygon and IDLgrSurface objects

Hidden line/hidden point display, color and vertex color blending with texture colors, and bottom color are not supported. Shading may be browser dependent. Front face culling is not supported and back face culling is only supported at the browser's discretion.

IDLgrLight objects

Lighting scope and intensity may be browser dependent.

IDLgrText objects

Text using the ONGLASS property is only supported for the initial view.

IDLgrViewgroup, IDLgrScene, IDLgrVolume objects

These objects are not supported.

IDLgrPalette objects

Palette objects are simulated using an RGB color model.

IDLgrPattern objects

Only solid or clear patterns are supported.

IDLgrFont, IDLgrSymbol objects

The THICK property is not supported.

IDLgrPolyline, IDLgrSymbol, IDLgrSurface, IDLgrPolygon and IDLgrPlot objects

Line attributes (thickness, linestyle) are not supported.

IDLgrView objects

Z-clipping control, aspect ratio preservation, the LOCATION property, and orthographic projections are not supported.

Destination objects

The COLOR_MODEL property is not fully supported in Indexed Color mode, when using a SHADER_RANGE (an RGB model will be substituted instead). The QUALITY property is not supported.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrVRML::Init](#)” on page 4132.

Properties

Objects of this class have the following properties. See “[IDLgrVRML Properties](#)” on page 4117 for details on individual properties.

- [ALL](#)
- [COLOR_MODEL](#)

- DIMENSIONS
- FILENAME
- GRAPHICS_TREE
- N_COLORS
- PALETTE
- QUALITY
- REGISTER_PROPERTIES
- RESOLUTION
- SCREEN_DIMENSIONS
- UNITS
- WORLDINFO
- WORLDTITLE

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- `IDLgrVRML::Cleanup`
- `IDLgrVRML::Draw`
- `IDLgrVRML::GetDeviceInfo`
- `IDLgrVRML::GetFontnames`
- `IDLgrVRML::GetProperty`
- `IDLgrVRML::GetTextDimensions`
- `IDLgrVRML::Init`
- `IDLgrVRML::SetProperty`

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.1

IDLgrVRML Properties

IDLgrVRML objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrVRML::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrVRML::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrVRML::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the retrievable properties associated with this object.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR_MODEL

An integer that determines whether to use the indexed color model for the buffer:

- 0 = RGB (the default)
- 1 = Color indexed.

In a property sheet, this property appears as an enumerated list with the following options:

- RGB
- Indexed

Property Type	ENUMLIST		
Name String	Color model		
Get: Yes	Set: No	Init: Yes	Registered: Yes

DIMENSIONS

A two-element integer vector of the form *[width, height]* to specify the dimensions of the window in units specified by the UNITS property. The default is [640,480].

Note

The only use of this property is to support the use of normalized coordinates for the dimensions of the IDLgrView object passed to the IDLgrVRML::Draw method.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

FILENAME

A string that specifies the name of a file into which the vector data will be saved. The default is `idl.wrl`.

Property Type	String		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

GRAPHICS_TREE

An object reference of type IDLgrView. If this property is set to a valid object reference, calling the Draw method on the destination object with no arguments will cause the object reference associated with this property to be drawn. If this object is valid and the destination object is destroyed, this object reference will be destroyed as well. By default the GRAPHICS_TREE property is set equal to the null-object. .

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_COLORS

An integer that specifies the number of colors (between 2 and 256) to be used if COLOR_MODEL is set to indexed.

Property Type	INTEGER		
Name String	Number of colors		
Get: Yes	Set: No	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the IDLgrPalette object class) that specifies the red, green, and blue values that are to be loaded into the buffer's color lookup table.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

QUALITY

An integer indicating the rendering quality at which graphics are to be drawn to the buffer. Valid values are:

- 0=Low
- 1=Medium
- 2=High (the default).
- In a property sheet, this property appears as an enumerated list with the following options:
 - Low
 - Medium
 - High

Property Type	ENUMLIST		
Name String	Quality		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RESOLUTION

A two-element floating-point vector of the form $[xres, yres]$ specifying the device resolution in centimeters per pixel.

Note

This property is used for text scaling and partial aspect ratio preservation only. The default value is $[0.0352778, 0.0352778]$ (72 DPI).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Resolution		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

SCREEN_DIMENSIONS

A two-element integer vector of the form [*width, height*] specifying the dimensions of the overall screen dimensions for the screen with which this object associated. The screen dimensions are measured in device units.

Property Type	Integer vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

UNITS

An integer that indicates the units of measure for the DIMENSIONS property. Valid values are:

- 0 = Device (the default)
- 1 = Inches
- 2 = Centimeters
- 3 = Normalized (relative to 1600 x 1200)

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

WORLDINFO

A list of strings for the info field of the VRML WorldInfo node. The default is the null string, "".

Property Type	String		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

WORLDTITLE

A string containing the title for the VRML WorldInfo node, TITLE field. The default is 'IDL VRML file'.

Property Type	String		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

IDLgrVRML::Cleanup

The IDLgrVRML::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrVRML::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.1

IDLgrVRML::Draw

The IDLgrVRML::Draw procedure method draws the given picture to this graphics destination.

Syntax

Obj->[IDLgrVRML::]Draw [, *Picture*]

Arguments

Picture

The view (an instance of an [IDLgrView](#) object) to be drawn. If the view has a LOCATION property, it is ignored.

Keywords

None

Version History

Introduced: 5.1

IDLgrVRML::GetDeviceInfo

The IDLgrVRML::GetDeviceInfo procedure method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=1 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Obj->[IDLgrVRML::]GetDeviceInfo [, ALL=variable]  
    [, MAX_NUM_CLIP_PLANES=variable]  
    [, MAX_TEXTURE_DIMENSIONS=variable]  
    [, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
    [, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable]
```

Arguments

None

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_NUM_CLIP_PLANES

Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, *MAX_TEXTURE_DIMENSIONS* contains a two element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, *MAX_VIEWPORT_DIMENSIONS* contains a two element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, *NAME* contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, *NUM_CPUS* contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The *NUM_CPUS* keyword accurately returns the number of CPUs for the SGI IRIX, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, *VENDOR* contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, *VERSION* contains the version of the rendering device driver as a string.

Version History

Introduced: 5.1

MAX_NUM_CLIP_PLANES keyword added: 5.6

IDLgrVRML::GetFontnames

The IDLgrVRML::GetFontnames function method returns the list of available fonts that can be used in [IDLgrFont](#) objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned; see [Appendix H, “Fonts”](#) for more information.

Syntax

```
Result = Obj->[IDLgrVRML::]GetFontnames( FamilyName [, IDL_FONTS={0 | 1 | 2}] [, STYLES=string] )
```

Return Value

Returns the list of available fonts that can be used in [IDLgrFont](#) objects.

Arguments

FamilyName

A string representing the name of the font family to which all of the returned fonts must belong. The string may be a fully specified family name, such as “Helvetica”. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. To return all available family names, use “*”.

Keywords

IDL_FONTS

Set this keyword to specify where to search for fonts that IDL may use. Set IDL_FONT to 1 to select only fonts installed by IDL and to 2 to select only fonts detected in the host operating system. The default value is 0, specifying that both IDL and operating system fonts should be returned.

STYLES

Set this keyword to a string specifying the styles that are to be matched by the returned font names. You can set STYLES to a fully specified style string, such as “Bold Italic”. If you set STYLES to the null string, ' ', only fontnames without style modifiers will be returned. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. The default value is

the string, “*”, which returns all fontnames containing the *FamilyName* argument, with or without style modifiers.

Version History

Introduced: 5.1

IDLgrVRML::GetProperty

The IDLgrVRML::GetProperty procedure method retrieves the value of a property or group of properties for the VRML object.

Syntax

Obj->[IDLgrVRML::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrVRML Properties](#)” on page 4117 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.1

IDLgrVRML::GetTextDimensions

The `IDLgrVRML::GetTextDimensions` function method retrieves the dimensions of a text or axis object that will be rendered in a window. The result is a 3-element double-precision floating-point vector [$xDim$, $yDim$, $zDim$] representing the dimensions of the text or axis object, measured in data units. If the object specified is an axis object, the result encompasses the tick labels and the title of the axis (if any).

Syntax

```
Result = Obj->[IDLgrVRML::]GetTextDimensions( TextObj [, DESCENT=variable]  
[, PATH=objref(s)] )
```

Return Value

Returns a 3-element double-precision floating-point vector [$xDim$, $yDim$, $zDim$] representing the dimensions of the text or axis object, measured in data units.

Arguments

TextObj

The object reference to a text or axis object for which the text dimensions are requested.

Keywords

DESCENT

Set this keyword equal to a named variable that will contain an array of double-precision floating-point values (one for each string in the `IDLgrText` object). The values represent the distance to travel (parallel to the `UPDIR` vector) from the text baseline to reach the bottom of the lowest descender in the string. All values will be negative numbers, or zero. This keyword is valid only if `TextObj` is an `IDLgrText` object.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the text dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this

keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued. If `IDLgrVRML::GetTextDimensions` is called from within a Draw method and the PATH keyword is not set, the alias path used to find the object during the draw is used, rather than the PARENT path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

Version History

Introduced: 5.1

IDLgrVRML::Init

The IDLgrVRML::Init function method initializes the VRML object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDLgrVRML' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrVRML::]Init([PROPERTY=value])  
(Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrVRML Properties](#)” on page 4117 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.1

IDLgrVRML:: SetProperty

The IDLgrVRML::SetProperty procedure method sets the value of a property or group of properties for the VRML world.

Syntax

Obj->[IDLgrVRML::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrVRML Properties](#)” on page 4117 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.1

IDLgrWindow

A window object is a representation of an on-screen area on a display device that serves as a graphics destination.

Note

Objects or subclasses of this type can not be saved or restored.

Note on Window Size Limits

The OpenGL libraries IDL uses impose limits on the maximum size of a drawable area. The limits are device-dependent — they depend both on your graphics hardware and the setting of the `RENDERER` property. Currently, the smallest maximum drawable area on any IDL platform is 1280 x 1024 pixels; the limit on your system may be larger.

Superclasses

[IDLitComponent](#)

Creation

See “[IDLgrWindow::Init](#)” on page 4164.

Properties

Objects of this class have the following properties. See “[IDLgrWindow Properties](#)” on page 4138 for details on individual properties.

- [ALL](#)
- [CURRENT_ZOOM](#)
- [DISPLAY_NAME \(X Only\)](#)
- [IMAGE_DATA](#)
- [MINIMUM_VIRTUAL_DIMENSIONS](#)
- [PALETTE](#)
- [REGISTER_PROPERTIES](#)
- [RESOLUTION](#)
- [COLOR_MODEL](#)
- [DIMENSIONS](#)
- [GRAPHICS_TREE](#)
- [LOCATION](#)
- [N_COLORS](#)
- [QUALITY](#)
- [RENDERER](#)
- [RETAIN](#)

- SCREEN_DIMENSIONS
- UNITS
- VISIBLE_LOCATION
- ZOOM_BASE
- TITLE
- VIRTUAL_DIMENSIONS
- ZBUFFER_DATA
- ZOOM_NSTEP

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- IDLgrWindow::Cleanup
- IDLgrWindow::Draw
- IDLgrWindow::Erase
- IDLgrWindow::GetContiguousPixels
- IDLgrWindow::GetDeviceInfo
- IDLgrWindow::GetDimensions
- IDLgrWindow::GetFontnames
- IDLgrWindow::GetProperty
- IDLgrWindow::GetTextDimensions
- IDLgrWindow::Iconify
- IDLgrWindow::Init
- IDLgrWindow::PickData
- IDLgrWindow::Read
- IDLgrWindow::Select
- IDLgrWindow::SetCurrentCursor
- IDLgrWindow::SetCurrentZoom
- IDLgrWindow::SetProperty
- IDLgrWindow::Show
- IDLgrWindow::ZoomIn
- IDLgrWindow::ZoomOut

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDLgrWindow Properties

IDLgrWindow objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via [IDLgrWindow::GetProperty](#). Properties with the word “Yes” in the “Init” column of the property table can be set via [IDLgrWindow::Init](#). Properties with the word “Yes” in the “Set” column in the property table can be set via [IDLgrWindow::SetProperty](#).

Note

For a discussion of the property description tables shown below, see [“About Object Property Descriptions”](#) on page 2671.

ALL

An anonymous structure containing the values of all of the properties associated with the *state* of this object. State information about the object includes things like color, range, tick direction, etc., but not image, vertex, or connectivity data, or user values.

Note

The fields of this structure may change in subsequent releases of IDL.

Property Type	Structure		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

COLOR_MODEL

An integer value that determines whether to use indexed color as the color model for the window:

- 0 = RGB (default)
- 1 = Color Index

Note

For some X11 display situations, IDL may not be able to support a color index model destination object in object graphics. We do, however, guarantee that an RGB color model destination will be available for all display situations.

In a property sheet, this property appears as an enumerated list with the following options:

- RGB
- Indexed color

Property Type	ENUMLIST		
Name String	Color model		
Get: Yes	Set: No	Init: Yes	Registered: Yes

CURRENT_ZOOM

A floating-point value that represents the current zoom factor associated with this window.

Property Type	FLOAT		
Name String	Zoom factor		
Get: Yes	Set: No	Init: No	Registered: Yes

DIMENSIONS

A two-element integer vector of the form [*width*, *height*] to specify the dimensions of the window in units specified by the UNITS property. By default, if no value is specified for DIMENSIONS, IDL uses the value of the “Default Window Width” and “Default Window Height” preferences set in the IDL Development Environment’s (IDLDE) Preferences dialog. If there is no preference file for the IDLDE, the DIMENSIONS property is set equal to one quarter of the screen size. There are limits on the maximum size of an IDLgrWindow object; see “[Note on Window Size Limits](#)” on page 4135 for details.

Note

Changing the DIMENSIONS property is merely a request to the windowing system and may be ignored.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Dimensions		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

DISPLAY_NAME (X Only)

A string that specifies the name of the X Windows display on which the window is to appear.

Property Type	STRING		
Name String	Display name		
Get: Yes	Set: Yes	Init: No	Registered: Yes

GRAPHICS_TREE

An object reference of type IDLgrScene, IDLgrViewgroup, or IDLgrView describing the graphics tree of this window object. If this property is set to a valid object reference, calling the Draw method on the destination object with no arguments will cause the object reference associated with this property to be drawn. If this object is valid and the destination object is destroyed, this object reference will be destroyed as well. By default the GRAPHICS_TREE property is set equal to the null-object.

Property Type	Object reference		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

IMAGE_DATA

A byte array representing the image that is currently displayed in the window. If the window object uses an RGB color model, the returned array will have dimensions (3, *winXSize*, *winYSize*), or (4, *winXSize*, *winYSize*) if an alpha channel is included. If the window object uses an Indexed color model, the returned array will have dimensions

(*winXSize*, *winYSize*). See “[IDLgrWindow::Read](#)” on page 4169 for more information.

Property Type	Byte array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

LOCATION

A two-element floating-point vector of the form $[x, y]$ to specify the location of the upper lefthand corner of the window relative to the display screen, in units specified by the UNITS property. By default, the window is positioned at one of four quadrants on the display screen, and the location is measured in device units.

Note

Changing LOCATION properties is merely a request and may be ignored for various reasons. LOCATION may be adjusted to take into account window decorations.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Location		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

MINIMUM_VIRTUAL_DIMENSIONS

A two-element vector, $[width, height]$, specifying the minimum dimensions allowed for the virtual canvas of this window. The unit of measure is specified by the UNITS property. In a scrolling window, the virtual canvas represents the full canvas, of which only a smaller visible portion is displayed at any given time. The default value of this property is $[0, 0]$, a value indicating that the minimum virtual dimensions should match the current virtual dimensions.

Property Type	Vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

N_COLORS

An integer value that specifies the number of colors (between 2 and 256) to be used if COLOR_MODEL is set to Indexed (1). This property is ignored if COLOR_MODEL is set to RGB (0).

Note

If COLOR_MODEL is set to Color Index (1), setting N_COLORS is treated as a request to your operating system. You should always check the actual number of available colors for any Color Indexed destination with the [IDLgrWindow::GetProperty](#) method. The actual number of available colors depends on your system and also on how you have used IDL.

Property Type	INTEGER		
Name String	Number of colors		
Get: Yes	Set: No	Init: Yes	Registered: Yes

PALETTE

An object reference to a palette object (an instance of the [IDLgrPalette](#) object class) to specify the red, green, and blue values that are to be loaded into the graphics destination's color lookup table, applicable if the Indexed color model is used.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Color palette		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

QUALITY

An integer indicating the rendering quality at which graphics are to be drawn to this destination. Valid values are:

- 0 = Low
- 1 = Medium
- 2 = High (default).

In a property sheet, this property appears as an enumerated list with the following options:

- Low
- Medium
- High

Property Type	ENUMLIST		
Name String	Draw quality		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

REGISTER_PROPERTIES

A Boolean value that determines whether to register properties available for this object. Set this property to automatically register the following properties of the object for display in a property sheet. This property is useful mainly when creating iTools. By default, no properties are registered.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

RENDERER

An integer value indicating which graphics renderer to use when drawing objects within the window. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL's software implementation

By default, IDL uses the value specified in the Graphics tab of the IDLDE Preferences dialog. If the preference is not explicitly set in the IDLDE and your platform has a native OpenGL implementation, IDL uses that implementation as the default. If your platform does not have a native OpenGL implementation, IDL uses its own software implementation regardless of the value of this property. See [“Hardware vs. Software Rendering”](#) in Chapter 34 of the *Using IDL* manual for details. Your choice of renderer may also affect the maximum size of an IDLgrWindow object; see [“Note on Window Size Limits”](#) on page 4135 for details.

In a property sheet, this property appears as an enumerated list with the following options:

- OpenGL
- Software.

Property Type	ENUMLIST		
Name String	Renderer		
Get: Yes	Set: No	Init: Yes	Registered: Yes

RESOLUTION

A floating-point vector of the form $[xres, yres]$ reporting the pixel resolution, measured in centimeters per pixel. This value is stored in double precision.

Property Type	FLOAT		
Name String	Resolution		
Get: Yes	Set: No	Init: No	Registered: Yes

RETAIN

An integer value that specifies how backing store should be handled for the window. By default, if no value is specified for RETAIN, IDL uses the value of the “Backing Store” preference set in the IDL Development Environment’s (IDLDE) Preferences dialog. If there is no preference file for the IDLDE (that is, if you always use IDL in plain tty mode), the RETAIN property is set equal to 0 by default.

- 0 = No backing store.
- 1 = The server or window system is requested to provide the backing store. Note that requesting backing store from the server is only a request; backing store may not be provided in all situations.
- 2 = Requests that IDL provide the backing store directly. In some situations, IDL can not provide this backing store in Object Graphics. To see if IDL provided backing store, query the RETAIN property of [IDLgrWindow::GetProperty](#). IDL may also alter the RENDERER property while attempting to provide backing store.

In IDL Object Graphics, it is almost always best to disable backing store (that is, set the `RETAIN` property equal to zero). This is because drawing to an off-screen pixmap (which is what happens when backing store is enabled) almost always bypasses any hardware graphics acceleration that may be available, causing all rendering to be done in software. To ensure that windows are redrawn properly, enable the generation of expose events on the `WIDGET_DRAW` window and redraw the window explicitly when an expose event is received.

Note

If you are using software rendering (that is, the `RENDERER` property is set equal to one), IDL will refresh the window automatically regardless of the setting of the `RETAIN` property.

In a property sheet, this property appears as an enumerated list with the following options:

- No backing
- Server/Window
- IDL

Property Type	ENUMLIST		
Name String	Retain		
Get: Yes	Set: No	Init: Yes	Registered: Yes

SCREEN_DIMENSIONS

A two-element floating-point vector of the form $[width, height]$ specifying the dimensions of the overall screen dimensions for the screen with which this window is associated. The screen dimensions are measured in device units.

Note

The maximum screen dimension size depends on the graphics device.

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

TITLE

A string that represents the title of the window.

Property Type	STRING		
Name String	Title		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

UNITS

An integer value that indicates the units of measure for the LOCATION and DIMENSIONS properties. Valid values are:

- 0 = Device (default)
- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to the dimensions of the screen.

Note

If you set the value of the UNITS property (using the SetProperty method) without also setting the value of the LOCATION and DIMENSIONS properties, IDL will convert the current size and location values into the new units.

In a property sheet, this property appears as an enumerated list with the following options:

- Device
- Inches
- Centimeters
- Normalized

Property Type	ENUMLIST		
Name String	Units		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VIRTUAL_DIMENSIONS

A two-element integer vector, [width, height], specifying the dimensions of the *virtual canvas* for this window. In a scrolling window, the virtual canvas represents the full canvas, of which only a smaller visible portion is displayed at any given time. The default is [0,0], indicating that the virtual canvas dimensions should match the visible dimensions (as specified via the DIMENSIONS property).

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Virtual dimensions		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

VISIBLE_LOCATION

A two-element integer vector, [x,y], specifying the lower left location of the visible portion of the canvas (relative to the virtual canvas). In a scrolling window, the virtual canvas represents the full canvas, of which only a smaller visible portion is displayed at any given time.

This property is registered as a user-defined property, but it is hidden by default.

Property Type	USERDEF		
Name String	Visible location		
Get: Yes	Set: Yes	Init: Yes	Registered: Yes

ZBUFFER_DATA

A floating-point array representing the zbuffer that is currently within the buffer. The returned array will have dimensions (*xdim*, *ydim*).

Property Type	Floating-point array		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

ZOOM_BASE

A floating point value representing the base value by which the window's current zoom factor will be multiplied or divided for zooming in or zooming out. Allowable values are those greater than 1.0. The default is 2.0.

Property Type	Float		
Name String	<i>not displayed</i>		
Get: Yes	Set: Yes	Init: Yes	Registered: No

ZOOM_NSTEP

A named variable that, upon return, will contain an integer that indicates the number of times this window's ZOOM_BASE property has been applied to achieve the current zoom factor. A positive value indicates that the current zoom factor represents the result of zooming in ZOOM_NSTEP times. A negative value indicates that the current zoom factor represents the result of zooming out ZOOM_NSTEP times.

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Version History

MINIMUM_VIRTUAL_DIMENSIONS, ZOOM_BASE, and ZOOM_NSTEP
added: 6.1

IDLgrWindow::Cleanup

The IDLgrWindow::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDLgrWindow::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrWindow::Draw

The IDLgrWindow::Draw procedure method draws the specified scene or view object to this graphics destination.

Note

Objects are drawn to the destination device in the order that they are added to the model, view, viewgroup, or scene object that contains them.

Syntax

```
Obj->[IDLgrWindow::]Draw [, Picture] [, CREATE_INSTANCE={1 | 2}]
[, /DRAW_INSTANCE]
```

Arguments

Picture

The view (an instance of an [IDLgrView](#) object), viewgroup (an instance of an [IDLgrViewgroup](#) object), or scene (an instance of an [IDLgrScene](#) object) to be drawn. If *Picture* is not supplied, the graphics hierarchy specified by the [GRAPHICS_TREE](#) property of the IDLgrWindow object will be drawn, if it exists. If neither the *Picture* argument or the [GRAPHICS_TREE](#) property are specified, nothing is drawn.

Keywords

CREATE_INSTANCE

Set this keyword equal to one specify that this scene or view is the unchanging part of a drawing. Some destinations can make an instance from the current window contents without having to perform a complete redraw. If the view or scene to be drawn is identical to the previously drawn view or scene, this keyword can be set equal to 2 to hint the destination to create the instance from the current window contents if it can.

DRAW_INSTANCE

Set this keyword to specify that this scene or view is the changing part of a drawing. It is overlaid on the result of the most recent [CREATE_INSTANCE](#) draw.

Version History

Introduced: 5.0

IDLgrWindow::Erase

The IDLgrWindow::Erase procedure method erases the entire contents of the window and initializes the depth buffer.

Syntax

Obj->[IDLgrWindow::]Erase [, **COLOR**=*index or RGB vector*]

Arguments

None

Keywords

COLOR

Set this keyword to the color to be used for the erase. The color may be specified as a color lookup table index or as a three-element vector [*red, green, blue*]. The default erase color is white.

Version History

Introduced: 5.0

IDLgrWindow::GetContiguousPixels

The `IDLgrWindow::GetContiguousPixels` function method returns an array of long integers whose length is equal to the number of colors available in the index color mode (that is, the value of the `N_COLORS` property).

The returned array marks contiguous pixels with the ranking of the range's size. This means that within the array, the elements in the largest available range are set to zero, the elements in the second-largest range are set to one, etc. Use this range to set an appropriate colormap for use with the `SHADE_RANGE` property of the [IDLgrSurface](#) and [IDLgrPolygon](#) object classes.

To get the largest contiguous range, you could use the following IDL command:

```
result = obj->GetContiguousPixels()  
Range0 = WHERE(result EQ 0)
```

A contiguous region in the colormap can be increasing or decreasing in values. The following would be considered contiguous:

```
[0,1,2,3,4]  
[4,3,2,1,0]
```

Syntax

```
Result = Obj->[IDLgrWindow::]GetContiguousPixels()
```

Return Value

Returns an array of long integers whose length is equal to the number of colors available in the index color mode.

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrWindow::GetDeviceInfo

The IDLgrWindow::GetDeviceInfo procedure method returns information which allows IDL applications to intelligently make decisions for optimal performance. For example, it allows an application to determine if RENDERER=0 is actually implemented in hardware. It also allows applications to make optimal quality decisions when dynamically building texture maps.

Syntax

```
Obj->[IDLgrWindow::]GetDeviceInfo [, ALL=variable]  
    [, MAX_NUM_CLIP_PLANES=variable]  
    [, MAX_TEXTURE_DIMENSIONS=variable]  
    [, MAX_VIEWPORT_DIMENSIONS=variable] [, NAME=variable]  
    [, NUM_CPUS=variable] [, VENDOR=variable] [, VERSION=variable]
```

Arguments

None

Keywords

ALL

Set this keyword to a named variable which, upon return, contains a structure with the values of all the device information keywords as fields.

MAX_NUM_CLIP_PLANES

Set this keyword to a named variable that upon return will contain an integer that specifies the maximum number of user-defined clipping planes supported by the device.

MAX_TEXTURE_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_TEXTURE_DIMENSIONS contains a two element integer array that specifies the maximum texture size supported by the device.

MAX_VIEWPORT_DIMENSIONS

Set this keyword equal to a named variable. Upon return, MAX_VIEWPORT_DIMENSIONS contains a two element integer array that specifies the maximum size of a graphics display supported by the device.

NAME

Set this keyword equal to a named variable. Upon return, NAME contains the name of the rendering device as a string.

NUM_CPUS

Set this keyword equal to a named variable. Upon return, NUM_CPUS contains an integer that specifies the number of CPUs that are known to, and available to IDL.

Note

The NUM_CPUS keyword accurately returns the number of CPUs for the SGI IRIX, SUN, and Microsoft Windows platforms. For platforms other than these, the number returned may not reflect the actual number of CPUs available to IDL in the current system.

VENDOR

Set this keyword equal to a named variable. Upon return, VENDOR contains the name of the rendering device creator as a string.

VERSION

Set this keyword equal to a named variable. Upon return, VERSION contains the version of the rendering device driver as a string.

Version History

Introduced: 5.0

MAX_NUM_CLIP_PLANES keyword added: 5.6

IDLgrWindow::GetDimensions

The IDLgrWindow::GetDimensions function method returns a two-element vector, [*width*, *height*], representing the visible dimensions (in pixel units) of this window.

Syntax

```
Result = Obj -> [IDLgrWindow:]GetDimensions
    ( [, MINIMUM_VIRTUAL_DIMENSIONS=variable]
      [, ORIGINAL_VIRTUAL_DIMENSIONS=variable]
      [, VIRTUAL_DIMENSIONS=variable] [, VISIBLE_LOCATION=variable])
```

Return Value

Returns the visible dimensions of the window (in pixel units) in a two-element vector [*width*, *height*].

Arguments

None

Keywords

MINIMUM_VIRTUAL_DIMENSIONS

Set this keyword to a named variable that will contain a two-element vector, [*width*, *height*], representing the minimum allowed virtual dimensions for this window.

Note

Returning MINIMUM_VIRTUAL_DIMENSIONS using GetDimensions provides the width and height of the window in pixel units. The property, “MINIMUM_VIRTUAL_DIMENSIONS” on page 4141, returns the dimensions of the window in units specified by the UNITS property.

ORIGINAL_VIRTUAL_DIMENSIONS

Set this keyword to a named variable that will contain a two-element vector, [*width*, *height*], representing the virtual dimensions in pixels for this window if the current zoom factor was 1.0.

VIRTUAL_DIMENSIONS

Set this keyword to a named variable that will contain a two-element vector, [*width*, *height*], representing the dimensions in pixels of the virtual canvas for this window.

VISIBLE_LOCATION

Set this keyword to a named variable that will contain a two-element vector, [*x*, *y*], representing the lower-left location of the visible portion of the window (relative to its virtual canvas).

Version History

Introduced: 6.1

IDLgrWindow::GetFontnames

The IDLgrWindow::GetFontnames function method returns the list of available fonts that can be used in [IDLgrFont](#) objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned; see [Appendix H, “Fonts”](#) for more information.

Syntax

```
Result = Obj->[IDLgrWindow::]GetFontnames(FamilyName [, IDL_FONTS={0 | 1 | 2}] [, STYLES=string ] )
```

Return Value

Returns the list of available fonts that can be used in [IDLgrFont](#) objects.

Arguments

FamilyName

A string representing the name of the font family to which all of the returned fonts must belong. The string may be a fully specified family name—such as “Helvetica”. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. To return all available family names, use “*”.

Keywords

IDL_FONTS

Set this keyword to specify where to search for fonts that IDL may use. Set IDL_FONT to 1 to select only fonts installed by IDL and to 2 to select only fonts detected in the host operating system. The default value is 0, specifying that both IDL and operating system fonts should be returned.

STYLES

Set this keyword to a string specifying the styles that are to be matched by the returned font names. You can set STYLES to a fully specified style string, such as “Bold Italic”. If you set STYLES to the null string, ' ', only fontnames without style modifiers will be returned. You can use both “*” and “?” as wildcard characters, matching any number of characters or one character respectively. The default value is

the string, “*”, which returns all fontnames containing the *FamilyName* argument, with or without style modifiers.

Version History

Introduced: 5.0

IDLgrWindow::GetProperty

The IDLgrWindow::GetProperty procedure method retrieves the value of a property or group of properties for the window.

Syntax

Obj->[IDLgrWindow::]GetProperty[, *PROPERTY=variable*]

Arguments

None

Keywords

Any property listed under “[IDLgrWindow Properties](#)” on page 4138 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Any keyword not recognized is passed to this object’s superclass.

Version History

Introduced: 5.0

IDLgrWindow::GetTextDimensions

The `IDLgrWindow::GetTextDimensions` function method retrieves the dimensions of a text or axis object that will be rendered in a window. The result is a 3-element double-precision floating-point vector $[xDim, yDim, zDim]$ representing the dimensions of the text or axis object, measured in data units. If the object specified is an axis object, the result encompasses the tick labels and the title of the axis (if any).

Syntax

```
Result = Obj->[IDLgrWindow::]GetTextDimensions( TextObj  
    [, DESCENT=variable] [, PATH=objref(s)] )
```

Return Value

Returns a 3-element double-precision floating-point vector $[xDim, yDim, zDim]$ representing the dimensions of the text or axis object, measured in data units.

Arguments

TextObj

The object reference to a text or axis object for which the text dimensions are requested.

Keywords

DESCENT

Set this keyword equal to a named variable that will contain an array of double-precision floating-point values (one for each string in the `IDLgrText` object). The values are the distance to travel (parallel to the `UPDIR` direction) from the baseline to reach the bottom of all the descenders for the string; the values will be negative or 0. This keyword is only valid if `TextObj` is of the class `IDLgrText`.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to compute the text dimensions. Each path object reference specified with this keyword must contain an alias. The text dimensions are computed for the version of the object falling within that path. If this keyword is not set, the `PARENT` properties determine the path from the current

object to the top of the graphics hierarchy and no alias paths are pursued. If `IDLgrWindow::GetTextDimensions` is called from within a Draw method and the `PATH` keyword is not set, the alias path used to find the object during the draw is used, rather than the `PARENT` path.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in `IDLgrModel::Add`.

Version History

Introduced: 5.0

IDLgrWindow::Iconify

The IDLgrWindow::Iconify procedure method iconifies or de-iconifies the window.

Note

Iconification under window systems is solely handled by the window manager; client applications, such as IDL, do not have the capability to manage icons. The Iconify method provides a hint to the window manager, which applies the information as it sees fit.

Syntax

Obj->[IDLgrWindow::]Iconify, *IconFlag*

Arguments

IconFlag

Set *IconFlag* to 1 (one) to iconify the window or to 0 (zero) to restore the window. If the window is already restored, it is brought to the front of the window stack.

Keywords

None

Version History

Introduced: 5.0

IDLgrWindow::Init

The IDLgrWindow::Init function method initializes the window object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Warning

When an IDLgrWindow object is created, its depth buffer is in an indeterminate state. This may cause unexpected behavior if the view being drawn into the window is transparent. In addition, parts of the window not covered by any view may contain random pixels. Calling the [IDLgrWindow::Erase](#) method will initialize the depth buffer to its proper value and write the proper color value to the background.

Syntax

```
Obj = OBJ_NEW('IDLgrWindow' [, PROPERTY=value])
```

or

```
Result = Obj->[IDLgrWindow::]Init([, PROPERTY=value]) (Only in a subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 if initialization was successful, or zero otherwise.

Arguments

None

Keywords

Any property listed under “[IDLgrWindow Properties](#)” on page 4138 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

All other keywords are passed to the superclass of this object.

Version History

Introduced: 5.0

IDLgrWindow::PickData

The IDLgrWindow::PickData function method maps a point in the two-dimensional device space of the window to a point in the three-dimensional data space of an object tree. The resulting 3-D data space coordinates are returned in a user-specified variable. The PickData function returns one if the specified location in the window's device space "hits" a graphic object, or zero if no object was "hit". PickData returns -1 if the point selected falls outside of the specified view or window.

Note

The PickData method relies on the contents of the depth buffer at the time it is called to compute and return its results. Be sure that the depth buffer contents are appropriate for getting the expected results from PickData. For example, you may use the DEPTH_WRITE_DISABLE property of an object to prevent an object from modifying the depth buffer as it is drawn. This prevents the object from affecting the result returned by the PickData method.

Syntax

```
Result = Obj->[IDLgrWindow:]PickData( View, Object, Location, XYZLocation  
[, DIMENSIONS=[w,h]] [, PATH=objref(s)] [, PICK_STATUS=variable])
```

Return Value

Returns 1 if the specified location in the window's device space "hits" a graphic object, or 0 if no object was hit. Returns -1 if the point selected falls outside the specified view or window.

Arguments

View

The object reference of an IDLgrView object that contains the object being picked.

Object

The object reference of a model or atomic graphic object from which the data space coordinates are being requested.

Note

Even though you can specify any object for this argument, the location that is returned may not lie on this object. If another object is in front of the specified

object then the returned location lies on the object in front. However, the coordinate system used is that of the object specified by this argument.

Location

A two-element vector $[x, y]$ specifying the location in the window's device space of the point to pick data from.

XYZLocation

A named variable that will contain the three-dimensional double-precision floating-point data space coordinates of the picked point. Note that the value returned in this variable is a location, not a data value.

Note

If the atomic graphic object specified as the target has been transformed using either the `LOCATION` or `DIMENSIONS` properties (this is only possible with `IDLgrAxis`, `IDLgrImage`, and `IDLgrText` objects), these transformations will *not* be included in the data coordinates returned by the `PickData` function. This means that you may need to re-apply the transformation accomplished by specifying `LOCATION` or `DIMENSIONS` once you have retrieved the data coordinates with `PickData`. This situation does not occur if you transform the axis, text, or image object using the `[XYZ]COORD_CONV` properties.

Keywords

DIMENSIONS

Set this keyword to a two-element array $[w, h]$ to specify data picking should occur for all device locations that fall within a *pick box* of these dimensions. The pick box will be centered about the coordinates $[x, y]$ specified in the *Location* argument, and will occupy the rectangle defined by:

$$(x-(w/2), y-(h/2)) - (x+(w/2), y+(h/2))$$

By default, the pick box covers a single pixel. The array returned via the *XYZLocation* argument will have dimensions $[3, w, h]$.

PATH

Set this keyword to a single object reference or a vector of object references. This keyword specifies the path in the graphics hierarchy to map the device position to a data space coordinate. Each path object reference specified with this keyword must

contain an alias. The data space coordinate is computed for the version of the object falling within that path. If this keyword is not set, the PARENT properties determine the path from the current object to the top of the graphics hierarchy and no alias paths are pursued.

Note

For more information on aliases, refer to the [ALIAS](#) keyword in IDLgrModel::Add.

PICK_STATUS

Set this keyword to a named variable that will contain “hit” information for each pixel in the pick box. If the DIMENSIONS keyword is not set, the PICK_STATUS will be a scalar value exactly matching the *Result* of the method call. If the DIMENSIONS keyword is set, the PICK_STATUS variable will be an array matching the dimensions of the pick box. Each value in the PICK_STATUS array corresponds to a pixel in the pick box, and will be set to one of the following values:

Value	Description
-1	The pixel falls outside of the window’s viewport.
0	No graphic object is “hit” at that pixel location.
1	A graphic object is “hit” at that pixel location.

Table 8-10: PICK_STATUS Keyword Values

Version History

Introduced: 5.0

PICK_STATUS keyword added: 5.6

IDLgrWindow::Read

The IDLgrWindow::Read function method reads an image from a window. The returned value is an instance of the [IDLgrImage](#) object class.

Syntax

Result = Obj->[IDLgrWindow:]Read()

Arguments

None

Keywords

None

Version History

Introduced: 5.0

IDLgrWindow::Select

The IDLgrWindow::Select function method returns a list of objects selected at a specified location. If no objects are selected, the Select function returns -1.

Note

IDL returns a maximum of 512 objects. This maximum may be smaller if any of the objects are contained in deep model hierarchies. Because of this limit, it is possible that not all objects eligible for selection will appear in the list.

Syntax

```
Result = Obj->[IDLgrWindow::]Select( Picture, XY [, DIMENSIONS=[width,
    height]] [, /ORDER] [, SUB_SELECTION=variable] [, UNITS={0 | 1 | 2 | 3} ] )
```

Return Value

Returns a list of objects selected at a specified location. Returns -1 if no objects are selected.

Arguments

Picture

The view or scene (an instance of the [IDLgrView](#), [IDLgrViewgroup](#), or [IDLgrScene](#) class) whose children are among the candidates for selection.

If the first argument is a scene or viewgroup, then the returned object list will contain one or more views. If the first argument is a view, the list will contain atomic graphic objects (or model objects which have their SELECT_TARGET property set). Objects are returned in order, according to their distance from the viewer. The closer an object is to the viewer, the lower its index in the returned object list. If multiple objects are at the same distance from the viewer (views in a scene or 2-D geometry), the first object drawn will appear at a lower index in the list. (The ORDER keyword can be used to change this behavior.)

XY

A two-element array defining the center of the selection box in device space. By default, the selection box is 3 pixels by 3 pixels, as defined by the DIMENSIONS keyword.

Keywords

DIMENSIONS

Set this keyword to a two-element array $[w, h]$ to specify that the selection box will have a width w and a height h , and will be centered about the coordinates $[x, y]$ specified in the XY argument. The box occupies the rectangle defined by:

$$(x-(w/2), y-(h/2)) - (x+(w/2), y+(h/2))$$

Any object that intersects this box is considered to be selected. By default, the selection box is 3 pixels by 3 pixels.

ORDER

Set this keyword to control how objects that are the same distance from the viewer are ordered in the selection list. If $ORDER=0$ (the default), the order of objects in the selection list will be the same as the order in which the objects are drawn. If $ORDER=1$, the order of objects in the selection list will be the reverse of the order in which they are drawn. This keyword has no affect on the ordering of objects that are not at the same distance from the viewer.

Tip

If you are using $DEPTH_TEST_FUNCTION=4$ (“less than or equal”) on your graphics objects, set $ORDER=1$ to return objects at the same depth in the order in which they appear visually.

SUB_SELECTION

Set this keyword to a named variable that upon return will contain a vector of references to the graphic objects that were sub-selected at the given location. This vector includes: all graphic objects that are:

- Graphic objects contained by the nearest selected object (the first object in the return value)
- Select targets (IDLgrModel objects with the $SELECT_TARGET$ property set)
- Graphic objects displayed at the given XY location

UNITS

Set this keyword to indicate the units of measure. Valid values are:

- 0 = Device (default)

- 1 = Inches
- 2 = Centimeters
- 3 = Normalized: relative to the dimensions of the graphics destination

Version History

Introduced: 5.0

ORDER keyword added: 6.0

SUB_SELECTION keyword added: 6.1

IDLgrWindow::SetCurrentCursor

The IDLgrWindow::SetCurrentCursor procedure method sets the current cursor image to be used while positioned over a drawing area.

Syntax

Obj->[IDLgrWindow:]SetCurrentCursor [, *CursorName*] [, **IMAGE**=16 x 16
bitmap] [, **MASK**=16 x 16 bitmap] [, **HOTSPOT**=[*x*, *y*]]

Unix-Only Keywords: [, **STANDARD**=*index*]

Arguments

CursorName

A string that specifies which built-in cursor to use. This argument is ignored if any keywords to this routine are set. This string can either be a name provided to the REGISTER_CURSOR routine or one of the following:

- ARROW
- CROSSHAIR
- ICON
- IBEAM
- MOVE
- ORIGINAL
- SIZE_NE
- SIZE_NW
- SIZE_SE
- SIZE_SW
- SIZE_NS
- SIZE_EW
- UP_ARROW

Keywords

IMAGE

Set this keyword to a 16x16 column bitmap, contained in a 16-element short integer vector, specifying the cursor pattern. The offset from the upper-left pixel to the point that is considered the “hot spot” can be provided via the HOTSPOT keyword.

MASK

When the IMAGE keyword is set, the MASK keyword can be used to simultaneously specify the mask that should be used. In the mask, bits that are set indicate bits in the IMAGE that should be seen and bits that are not are “masked out”.

HOTSPOT

Set this keyword to a two-element vector specifying the $[x, y]$ pixel offset of the cursor “hot spot”, the point which is considered to be the mouse position, from the upper left corner of the cursor image. This parameter is only applicable if IMAGE is provided. The cursor image is displayed top-down (the first row is displayed at the top).

STANDARD

This keyword is only available on UNIX platforms.

Set this keyword to an X11 cursor font index to change the appearance of the cursor in the IDL graphics window to a glyph in this font. On non-X platforms, setting this keyword displays the crosshair cursor.

Version History

Introduced: 5.0

CursorName argument added: 5.6

IDLgrWindow::SetCurrentZoom

The IDLgrWindow::SetCurrentZoom procedure method sets the current zoom factor for this window. The current zoom factor, the virtual canvas dimensions, and the visible dimensions of the window are updated to reflect the new zoom factor.

Syntax

Obj-> [IDLgrWindow::]SetCurrentZoom, *ZoomFactor* [, /RESET]

Arguments

ZoomFactor

A positive floating point value indicating the zoom factor to be applied to the window. Values less than 1.0 zoom out; values greater than 1.0 zoom in on the contents of the window.

Keywords

RESET

Set this keyword to indicate that the zoom factor should be reset to 1.0. If this keyword is present, the *ZoomFactor* argument is ignored.

Version History

Introduced: 6.1

IDLgrWindow:: SetProperty

The IDLgrWindow::SetProperty procedure method sets the value of a property or group of properties for the window.

Syntax

Obj->[IDLgrWindow::]SetProperty[, *PROPERTY=value*]

Arguments

None

Keywords

Any property listed under “[IDLgrWindow Properties](#)” on page 4138 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

IDLgrWindow::Show

The IDLgrWindow::Show procedure method exposes or hides a window.

Syntax

Obj->[IDLgrWindow::]Show, *Position*

Arguments

Position

Set this argument equal to a non-zero value to expose the window, or to 0 to hide the window.

Keywords

None

Version History

Introduced: 5.0

IDLgrWindow::ZoomIn

The IDLgrWindow::ZoomIn procedure method causes the current zoom factor for this window to be increased (that is, multiplied by the factor given by the window's [ZOOM_BASE](#) property). The current zoom factor, the virtual canvas dimensions, and the visible dimensions of the window are updated to reflect the new zoom factor.

Syntax

Obj->[IDLgrWindow::]ZoomIn

Arguments

None

Keywords

None

Version History

Introduced: 6.1

IDLgrWindow::ZoomOut

The IDLgrWindow::ZoomIn procedure method causes the current zoom factor for this window to be decreased (that is, divided by the factor given by the window's [ZOOM_BASE](#) property). The current zoom factor, the virtual canvas dimensions, and the visible dimensions of the window are updated to reflect the new zoom factor.

Syntax

Obj->[IDLgrWindow::]ZoomOut

Arguments

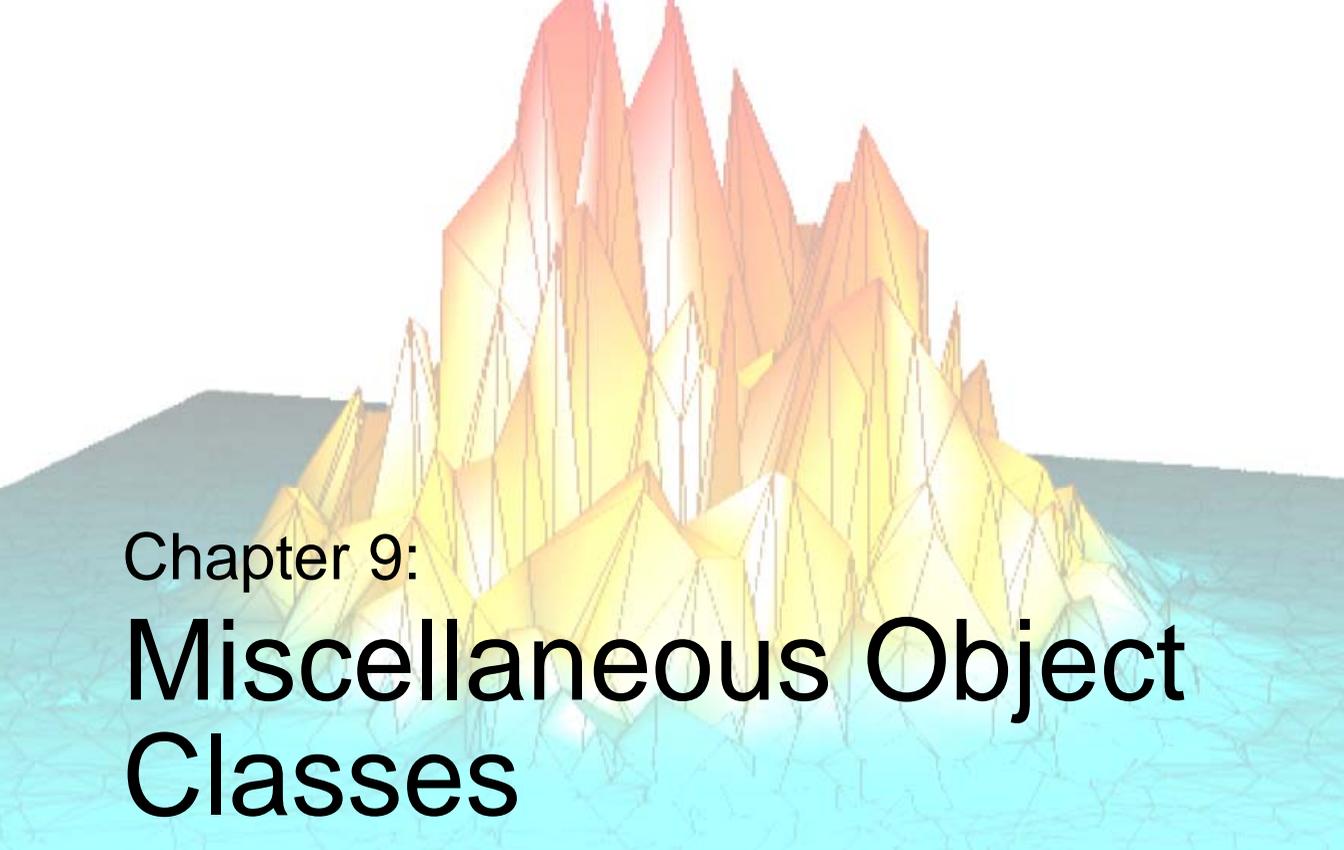
None

Keywords

None

Version History

Introduced: 6.1



Chapter 9: Miscellaneous Object Classes

This chapter describes IDL's Container, COM, Java, and Trackball class libraries.

IDL_Container	4182	IDLcomIDispatch	4217
IDL_Savefile	4193	IDLjavaObject	4223
IDLcomActiveX	4215	TrackBall	4231

IDL_Container

An IDL_Container object holds other objects. Destroying an IDL_Container object destroys any objects that have been added to the container via the Add method.

Superclasses

This class has no superclasses.

Creation

See “[IDL_Container::Init](#)” on page 4189.

Properties

Objects of this class have no properties of their own.

Methods

This class has the following methods:

- [IDL_Container::Add](#)
- [IDL_Container::Cleanup](#)
- [IDL_Container::Count](#)
- [IDL_Container::Get](#)
- [IDL_Container::Init](#)
- [IDL_Container::IsContained](#)
- [IDL_Container::Move](#)
- [IDL_Container::Remove](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

IDL_Container Properties

Objects of this class have no properties of their own.

IDL_Container::Add

The IDL_Container::Add procedure method adds one or more child objects to the container.

Syntax

Obj->[IDL_Container::]Add, *Objects* [**POSITION**=*index*]

Arguments

Objects

An object instance or array of object instances to be added to the container object.

Keywords

POSITION

Set this keyword equal to a scalar or array of zero-based index values. The number of elements specified must be equal to the number of object references specified by the *Objects* argument. Each index value specifies the position within the container at which a new object should be placed. The default is to add new objects at the end of the list of contained items.

Examples

If the container has three objects, a new object will be placed at the fourth position. Since positions begin at zero, this would be equivalent to setting POSITION=3.

Version History

Introduced: 5.0

IDL_Container::Cleanup

The IDL_Container::Cleanup procedure method performs all cleanup on the object.

Note

Cleanup methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->[IDL_Container::]Cleanup(*Only in subclass' Cleanup method.*)

Arguments

None.

Keywords

None.

Version History

Introduced: 5.0

IDL_Container::Count

The IDL_Container::Count function method returns the number of objects contained by the container object.

Syntax

Result = *Obj*->[IDL_Container::]Count()

Arguments

None.

Keywords

None.

Version History

Introduced: 5.0

IDL_Container::Get

The IDL_Container::Get function method returns an array of object references to objects in a container. Unless the ALL or POSITION keywords are specified, the first object in the container is returned. If no objects are found in the container, the Get function returns -1.

Syntax

```
Result = Obj->[IDL_Container::]Get ([, /ALL [, ISA=class_name(s)] | ,  
    POSITION=index] [COUNT=variable] )
```

Arguments

None.

Keywords

ALL

Set this keyword to return an array of object references to all of the objects in the container.

COUNT

Set this keyword equal to a named variable that will contain the number of objects selected by the function. If the ALL keyword is also specified, specifying this keyword is the same as calling the IDL_Container::Count method.

ISA

Set this keyword equal to a class name or vector of class names. This keyword is used in conjunction with the ALL keyword. The ISA keyword filters the array returned by the ALL keyword, returning only the objects that inherit from the class or classes specified by the ISA keyword.

Note

This keyword is ignored if the ALL keyword is not provided.

POSITION

Set this keyword equal to a scalar or array containing the zero-based indices of the positions of the objects to return.

Version History

Introduced: 5.0

IDL_Container::Init

The IDL_Container::Init function method initializes the container object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

Obj = OBJ_NEW('IDL_Container')

or

Result = *Obj*->[IDL_Container::]Init() (*Only in a subclass' Init method.*)

Arguments

None.

Keywords

None.

Version History

Introduced: 5.0

IDL_Container::IsContained

The IDL_Container::IsContained function method returns true (1) if the specified object is in the container, or false (0) otherwise.

Syntax

Result = *Obj*->[IDL_Container::]IsContained(*Object* [, POSITION=*variable*])

Arguments

Object

The object reference or vector of object references of the object(s) to search for in the container.

Keywords

POSITION

Set this keyword to a named variable that upon return will contain the position(s) at which (each of) the argument(s) is located within the container, or -1 if it is not contained.

Version History

Introduced: 5.0

IDL_Container::Move

The IDL_Container::Move procedure method moves an object from one position in a container to a new position. The order of the other objects in the container remains unchanged.

Positioning within a container controls the rendering order of the contained objects. The object whose location has the lowest index value is rendered first. If several objects are located at the same point in three-dimensional space, the object rendered first will occlude objects rendered later. Objects located “behind” other objects in three-dimensional space must be rendered before objects in front of them, even if the “front” objects are translucent.

Syntax

Obj->[IDL_Container::]Move, *Source*, *Destination*

Arguments

Source

The zero-based index of the current location of the object to be moved.

Destination

The zero-based index of the location in the container where the object will reside after being moved.

Keywords

None.

Version History

Introduced: 5.0

IDL_Container::Remove

The IDL_Container::Remove procedure method removes an object from the container.

Syntax

```
Obj->[IDL_Container:]Remove [, Child_object | , POSITION=index / , /ALL]
```

Arguments

Child_object

The object reference of the object to be removed from the container. If *Child_object* is not provided (and neither the ALL nor POSITION keyword are set), the first object in the container will be removed.

Keywords

ALL

Set this keyword to remove all objects from the container. If this keyword is set, the *Child_object* argument is not required.

POSITION

Set this keyword equal to the zero-based index of the object to be removed. If the *Child_object* argument is supplied, this keyword is ignored.

Version History

Introduced: 5.0

IDL_Savefile

The IDL_Savefile object provides complete query and restore capabilities for IDL SAVE files. IDL SAVE files are binary format files created with the SAVE procedure; they can contain data or IDL routines. Using IDL_Savefile, you can retrieve information about the user, machine, and system that created the SAVE file, as well as the number and size of the various items contained in the file (variables, common blocks, routines, *etc*). You can also selectively restore individual items from the SAVE file.

Use IDL_Savefile instead of the RESTORE procedure when you need to obtain detailed information on the items contained within a SAVE file without first restoring it, or when you wish to restore only selected items. Use RESTORE when you want to restore everything from the SAVE file using a simple interface.

Superclasses

This class has no superclasses.

Creation

See “[IDL_Savefile::Init](#)” on page 4201.

Properties

Objects of this class have no properties of their own.

Methods

This class has the following methods:

- [IDL_Savefile::Cleanup](#)
- [IDL_Savefile::Contents](#)
- [IDL_Savefile::Init](#)
- [IDL_Savefile::Names](#)
- [IDL_Savefile::Restore](#)
- [IDL_Savefile::Size](#)

In addition, this class inherits the methods of its superclasses (if any).

Example

Use the `IDL_Savefile` object to query the contents of a SAVE file containing data and selectively restore variables.

Typically, `IDL_Savefile::Contents` is the first method called on a savefile object; the information retrieved indicates the number of different types of items contained in the file (variables, routines, *etc.*). Next, a call to `IDL_Savefile::Names` is used to obtain the names of items of a specific type. `IDL_Savefile::Size` can be used to determine the size of variables in the file, and `IDL_Savefile::Restore` can be used to restore items for use within the current IDL session.

```

; Select a SAVE file to restore. The file contains volume data.
savefile = FILEPATH('cduskcD1400.sav', $
    SUBDIRECTORY=['examples', 'data'])

; Create a savefile object.
sObj = OBJ_NEW('IDL_Savefile', savefile)

; Query the contents of the SAVE file to determine the number of
; regular variables contained therein.
sContents = sObj->Contents()
PRINT, sContents.N_VAR

; IDL Prints:
; 3

; Retrieve the names of the variables in the SAVE file.
sNames = sObj->Names()
PRINT, sNames

; IDL Prints:
; DENSITY MASSFLUX VELOCITY

; Determine the size of the DENSITY variable.
sDensitySize = sObj->Size('density')
PRINT, sDensitySize

; IDL Prints:
; 3          30          30          15          1          13500

; Restore the DENSITY variable.
sObj->Restore, 'density'

; Display the volume data.
IVOLUME, density

```

Version History

Introduced: 6.1

See Also

[RESTORE](#), [SAVE](#)

IDL_Savefile Properties

Objects of this class have no properties of their own.

IDL_Savefile::Cleanup

The IDL_Savefile::Cleanup procedure method performs all necessary cleanup when the savefile object is destroyed. The SAVE file associated with the object is closed. In addition, all in-memory data structures created by IDL when cataloging the SAVE file are released.

Note

Cleanup methods are special life cycle methods, and as such cannot be called outside the context of object creation and destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Init or Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj->IDL_Savefile::Cleanup (Only in subclass' Cleanup method.)

Arguments

None.

Keywords

None.

Version History

Introduced: 6.1

IDL_Savefile::Contents

The IDL_Savefile::Contents function method returns a structure variable of type IDL_SAVEFILE_CONTENTS containing information about the associated SAVE file and its contents.

Syntax

Result = *Obj*->[IDL_Savefile::]Contents()

Return Value

Returns a structure variable of type IDL_SAVEFILE_CONTENTS containing information about the associated SAVE file and its contents. The structure contains the following fields:

Field	Type	Description
FILENAME	String	The file path to the open SAVE file.
DESCRIPTION	String	A user-supplied description string for the SAVE file, as specified using the DESCRIPTION keyword to the SAVE procedure. If the SAVE file lacks a user supplied description, this field contains a null string.
FILETYPE	String	A description of SAVE file type. For a standard SAVE file, this field will contain the string "Portable (XDR)". For a compressed SAVE file, this field will contain "Portable (XDR) compressed".
USER	String	The user name of the user who issued the SAVE command that created the SAVE file.
HOST	String	The name of the computer on which the SAVE file was created.
DATE	String	The date on which the SAVE file was created.

Table 9-1: Structure returned by IDL_Savefile::Contents.

Field	Type	Description
ARCH	String	The value of the <code>!VERSION.ARCH</code> system variable field of the IDL installation that created the SAVE file.
OS	String	The value of the <code>!VERSION.OS</code> system variable field of the IDL installation that created the SAVE file.
RELEASE	String	The value of the <code>!VERSION.RELEASE</code> system variable field of the IDL installation that created the SAVE file.
N_COMMON	Integer	The number of main-level common blocks contained in the SAVE file.
N_VAR	Integer	The number of variables contained in the SAVE file.
N_SYSVAR	Integer	The number of system variables contained in the SAVE file.
N_PROCEDURE	Integer	The number of procedures contained in the SAVE file.
N_FUNCTION	Integer	The number of functions contained in the SAVE file.
N_OBJECT_HEAPVAR	Integer	The number of object heap variables contained in the SAVE file.
N_POINTER_HEAPVAR	Integer	The number of pointer heap variables contained in the SAVE file.
N_STRUCTDEF	Integer	The number of named structure definitions contained in the SAVE file. Named structure definitions are saved when a variable that references them is saved.

Table 9-1: Structure returned by `IDL_Savefile::Contents`.

Arguments

None.

Keywords

None.

Example

See “[Example](#)” on page 4194.

Version History

Introduced: 6.1

IDL_Savefile::Init

The IDL_Savefile::Init function method initializes the savefile object. The SAVE file associated with the object is opened and scanned to construct an internal data structure representing the contents. The SAVE file remains open until the savefile object is destroyed — the user is not allowed to use the CLOSE procedure to close the SAVE file.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Obj = OBJ_NEW('IDL_Savefile' [, Filename ] [, FILENAME=string]  
      [, /RELAXED_STRUCTURE_ASSIGNMENT])
```

or

```
Result = Obj->IDL_Savefile::Init() (Only in subclass' Init method.)
```

Return Value

When this method is called indirectly, as part of the call to the OBJ_NEW function, the return value is an object reference to the newly-created object.

When called directly within a subclass Init method, the return value is 1 (one) if initialization was successful, or 0 (zero) otherwise.

Arguments

Filename

A scalar string that contains the path name of the SAVE file from which information or data is to be recovered. If *Filename* is not present but the FILENAME keyword is specified, the keyword value is used. If neither the argument or the keyword is present, the file `idlsave.dat` in IDL's current working directory is used.

Keywords

FILENAME

Set this keyword equal to a scalar string that contains the path name of the SAVE file from which information or data is to be recovered.

Note

The FILENAME keyword duplicates the functionality of the *Filename* argument. If both are provided, the value specified by the *Filename* argument is used.

RELAXED_STRUCTURE_ASSIGNMENT

Normally, the [IDL_Savefile::Restore](#) method is unable to restore a structure variable if the definition of its type has changed since the SAVE file was written. This commonly occurs when objects are saved and the class structures of the objects change before they are restored in another IDL session. In such cases, [IDL_Savefile::Restore](#) issues an error and does not restore the item.

Set the RELAXED_STRUCTURE_ASSIGNMENT keyword when creating a savefile object to cause the [IDL_Savefile::Restore](#) method to restore such incompatible values using “relaxed structure assignment,” in which all possible data are restored using a field-by-field copy.

See the description of the [STRUCT_ASSIGN](#) procedure for additional details. “[Relaxed Structure Assignment](#)” in Chapter 9 of the *Building IDL Applications* manual provides a more in-depth discussion of the structure-definition process.

Version History

Introduced: 6.1

IDL_Savefile::Names

The IDL_Savefile::Names function method returns names, or heap variable identifiers, of items contained within the SAVE file. By default, this method returns the names of variables in the file; keywords can be used to obtain identifiers for other items.

Syntax

```
Result = Obj->[IDL_SAVEFILE::]Names( [, COUNT=variable]  
    [, /COMMON_BLOCK | , COMMON_VARIABLE=string | , /FUNCTION |  
    , /OBJECT_HEAPVAR | , /POINTER_HEAPVAR | , /PROCEDURE |  
    , /STRUCTURE_DEFINITION | , /SYSTEM_VARIABLE] )
```

Return Value

If the SAVE file contains one or more items of the specified type, the return value is a string array containing the names of the items. If no items of the specified type exist, the return value is a scalar null string.

Note

If either the OBJECT_HEAPVAR or the POINTER_HEAPVAR keyword is specified, the return value is an integer array containing integer values that can be used to recover the heap variables stored in the SAVE file. If no heap variables of the specified type exist, the return value is 0 (zero).

Arguments

None.

Keywords

Note

The Names method will only return the names of one type of item in a single call. With the exception of the COUNT keyword, specifying more than one keyword may cause the method to return unexpected results (most often a null string).

COMMON_BLOCK

Set this keyword to retrieve the names of any main-level COMMON blocks contained in the SAVE file. A main-level COMMON block item in a SAVE file consists of the name of the block and the names of the variables contained within it. (The values of the variables are saved separately, as normal variables.)

COMMON_VARIABLE

Set this keyword to a scalar string that contains the name of a main-level COMMON block. The value returned by the Names method will be a string array containing the names of the variables contained in the specified COMMON block.

COUNT

Set this keyword equal to a named variable that will contain the number of identifiers returned. A value of 0 (zero) indicates that no identifiers of the specified type exist in the SAVE file, and that the result from the function is either a scalar null string or 0 (zero). A nonzero value indicates that the result is an array with that number of elements.

FUNCTION

Set this keyword to return the names of the compiled functions contained in the SAVE file.

OBJECT_HEAPVAR

Set this keyword to return the integer heap identifiers for the object heap variables contained in the SAVE file.

Note

It is rarely (if ever) necessary to restore heap variables directly; in almost all cases, restoring the regular variable that refers to the heap variable produces the desired effect. See [“Restoring Heap Variables Directly”](#) on page 4211 for a complete discussion of this topic.

Because the heap identifiers contained in the SAVE file being queried were assigned by the IDL session that saved the variables in the file, and not (in general) by the currently executing IDL session, the integers returned when OBJECT_HEAPVAR is set are *not* valid heap identifiers within the current IDL session. Because of this, the values are returned as plain integer values, and not as object references. The returned integer values can be supplied to the [IDL_Savefile::Restore](#) method (using the OBJECT_HEAPVAR keyword) to recover the object heap variables they reference.

POINTER_HEAPVAR

Set this keyword to return the integer heap identifiers for the pointer heap variables contained in the SAVE file.

Note

It is rarely (if ever) necessary to restore heap variables directly; in almost all cases, restoring the regular variable that refers to the heap variable produces the desired effect. See [“Restoring Heap Variables Directly”](#) on page 4211 for a complete discussion of this topic.

Because the heap identifiers contained in the SAVE file being queried were assigned by the IDL session that saved the variables in the file, and not (in general) by the currently executing IDL session, the integers returned when `POINTER_HEAPVAR` is set are *not* valid heap identifiers within the current IDL session. Because of this, the values are returned as plain integer values, and not as pointer references. The returned integer values can be supplied to the `IDL_Savefile::Restore` method (using the `POINTER_HEAPVAR` keyword) to recover the pointer heap variables they reference.

PROCEDURE

Set this keyword to return the names of the compiled procedures contained in the SAVE file.

STRUCTURE_DEFINITION

Structure definitions are saved by IDL as a side effect of saving structure variables that use the structure definition. Set this keyword to return the names of the structure definitions contained in the SAVE file.

SYSTEM_VARIABLE

Set this keyword to return the names of system variables contained in the SAVE file.

Example

See [“Example”](#) on page 4194.

Version History

Introduced: 6.1

IDL_Savefile::Restore

The `IDL_Savefile::Restore` procedure method selectively restores individual items from the associated SAVE file.

Given the *identifier* for an item within the SAVE file, this method restores the item along with any additional items (such as structure definitions or heap variables) upon which the specified item depends. Use the [IDL_Savefile::Names](#) method to retrieve valid item identifiers.

By default, this method restores variables from the SAVE file; keywords can be used to restore items of other types.

Warning

There are several issues of which you should be aware when using this method to restore pointers or object references. See “[Notes on Restored Heap Variables](#)” on page 4208 for details.

Syntax

```
Obj->[IDL_Savefile::]Restore, SaveItem [, /COMMON] [, /FUNCTION]
  [, /NEW_HEAPVAR=variable] [, /OBJECT_HEAPVAR]
  [, /POINTER_HEAPVAR] [, /PROCEDURE]
  [, /RESTORED_OBJECTS=variable] [, /STRUCTURE_DEFINITION]
  [, /VERBOSE]
```

Arguments

SaveItem

A scalar string or string array of identifiers for items to be restored from the SAVE file. Identifiers are retrieved from the SAVE file using the [IDL_Savefile::Names](#) method.

Note

If either the `OBJECT_HEAPVAR` keyword or the `POINTER_HEAPVAR` keyword is specified, *SaveItem* must be an integer or integer array.

Keywords

COMMON

Set this keyword to restore the main-level COMMON block definitions specified by *SaveItem*.

Note

To restore both a COMMON block definition and the variables it references, restore the definition first, then restore the variables.

FUNCTION

Set this keyword to restore the function(s) specified by *SaveItem*.

NEW_HEAPVAR

Set this keyword equal to a named variable that will contain valid references for heap variables.

Note

This keyword is only valid when either the OBJECT_HEAPVAR or the POINTER_HEAPVAR keyword is specified.

The returned variable will have the same structure as *SaveItem*, with each element in the returned array corresponding to the same element in the *SaveItem* array. If OBJECT_HEAPVAR is specified, the returned variable will contain an array of object references; if POINTER_HEAPVAR is specified, the returned variable will contain an array of pointers. See [“Notes on Restored Heap Variables”](#) on page 4208 for additional information on restoring heap variables.

OBJECT_HEAPVAR

Set this keyword to restore the object heap variable(s) specified by *SaveItem*. To obtain valid references to the newly restored heap variables, you must also specify the NEW_HEAPVAR keyword. See [“Notes on Restored Heap Variables”](#) on page 4208 for additional information on restoring heap variables.

POINTER_HEAPVAR

Set this keyword to restore the pointer heap variable(s) specified by *SaveItem*. To obtain valid references to the newly restored heap variables, you must also specify the

NEW_HEAPVAR keyword. See “[Notes on Restored Heap Variables](#)” on page 4208 for additional information on restoring heap variables.

PROCEDURE

Set this keyword to restore the procedure(s) specified by *SaveItem*.

RESTORED_OBJECTS

Set this keyword equal to a named variable that will contain an array of object references for any objects restored. The resulting list of objects is useful for programmatically calling the objects’ own restore methods (if any). If no objects are restored, the variable will contain a null object reference.

STRUCTURE_DEFINITION

Set this keyword to restore the named structure definition(s) specified by *SaveItem*. Named structure definitions are saved by IDL when a structure variable of that type is saved, and are restored when the variable is restored. The STRUCTURE_DEFINITION keyword allows you to restore just the definition, without the variable. This can be useful if you simply wish to create a new variable of the same type, and have no use for the saved variable data.

VERBOSE

Set this keyword to print an informative message for each item restored.

Note

Messages are only printed for top-level named variables. Heap variable references returned by the NEW_HEAPVAR keyword will not generate individual messages.

Notes on Restored Heap Variables

IDL variables can contain object or pointer references to *heap variables*, and these heap variables can contain other such references, either to themselves or to other heap variables. There is no limit to the number of references that can be made to a given heap variable. This flexible design allows the construction of arbitrary data structures. (See “[Heap Variables](#)” in Chapter 8 of the *Building IDL Applications* manual for a more complete discussion of heap variables.)

While the RESTORE procedure handles all details related to restoring heap variables in a way that is transparent to the IDL user, the IDL_Savefile::Restore method provides greater flexibility and thus requires greater care. In particular, you should be

aware of the following when using the `IDL_Savefile::Restore` method to restore heap variables:

1. An `IDL_Savefile` object will restore a given heap variable only once. This has implications for situations in which heap variables might be freed during the `IDL_Savefile` object's lifetime. See [“Caution Regarding Restored Heap Variables”](#) on page 4210 for a complete discussion of this issue.
2. The `IDL_Savefile` object makes it possible to restore a heap variable independently of any regular IDL variable that might refer to it. This feature is described in detail in [“Restoring Heap Variables Directly”](#) on page 4211.

Understanding How Heap Variables are Restored

When the `SAVE` procedure saves variables that contain object or pointer references, the referenced heap variables are also saved. Internally, heap variables are represented by unique integers called *heap identifiers*, which are not generally accessible (or useful) to the IDL user. The unique integers used to identify heap variables are assigned by IDL when the variables are created, and are therefore only unique within that IDL session. A given heap variable will typically have completely different identifiers in two different IDL sessions.

When `RESTORE` later restores variables containing object or pointer references, the heap variables are likewise restored. Note, however, that the unique identifiers for the restored heap variables will be different from the identifiers they had in the IDL session in which the `SAVE` file was created. `RESTORE` handles the mapping of the old identifiers to the new ones internally, in a manner that is completely transparent to the IDL user. This allows the user to save and restore very complex data structures using the simple interface provided by the `SAVE` and `RESTORE` procedures.

Notice that when using `SAVE` and `RESTORE`, the user always deals with regular named variables. Heap variables are never explicitly saved or restored; they are referenced only indirectly, through the regular variables that refer to them. Working with restored heap variables directly would be inconvenient, because heap variables lack mnemonic names that allow for easy identification, and because the original integer identifiers assigned by IDL when the heap variables were created have no meaning in the IDL session into which the variables are restored.

The `IDL_Savefile::Restore` method duplicates the behavior of the `RESTORE` procedure with regard to heap variables. That is, when a regular IDL variable that refers to a pointer or object heap variable is restored, the heap variable is also restored automatically. In addition, although it is rarely (if ever) useful, the `IDL_Savefile::Restore` method provides the ability to restore heap variables *without* restoring the IDL variables that refer to them. To accomplish this, the `IDL_Savefile`

object allows you to retrieve the integer heap identifiers stored in the SAVE file, using the `POINTER_HEAPVAR` and `OBJECT_HEAPVAR` keywords to the `Names` method. Note that it would be incorrect to treat these identifiers as object or pointer references, because they are not valid references in the restoring IDL session. For this reason, they are reported as simple integer values. They have no special meaning outside of that given to them by the `IDL_Savefile` object for a given object instance.

Note

Restoring heap variables independently of regular variables is not recommended; the mechanism is provided only for completeness. See [“Restoring Heap Variables Directly”](#) on page 4211, for additional information.

Caution Regarding Restored Heap Variables

A given instance of an `IDL_Savefile` object will never restore a given pointer or object heap variable more than once, regardless of the number of pointers or object references that refer to that heap variable. While this reflects the expected behavior of pointers and object references — references to the same underlying quantity should all point to the same heap variable — you must take care in situations in which heap variables can be destroyed during the lifetime of the `IDL_Savefile` object.

To make the implications of this behavior clear, consider an example in which a SAVE file contains two variables, A and B, each of which is a pointer reference to the same underlying pointer heap variable. After instantiating an `IDL_Savefile` object named `obj` that refers to the SAVE file, execute the following IDL statements:

```
obj->RESTORE, 'a'  
PTR_FREE, a  
obj->RESTORE, 'b'
```

The result is that both restored variables A and B are dangling references to a non-existent heap variable. IDL does not re-create the heap variable when B is restored, because the `IDL_Savefile` object knows that it already restored it once. The process of freeing the heap variable using the `PTR_FREE` procedure does not influence the behavior of the `IDL_Savefile` object — as far as the savefile object is concerned, the heap variable referred to by the variable B has already been restored, and should not be restored again.

Tip

To force IDL to create multiple instances of a single saved heap variable, you can create multiple instances of the `IDL_Savefile` object referring to the same SAVE file, and use each one to create a distinct and separate instance of the heap variable. We caution you that this sort of code can become confusing very quickly.

Restoring Heap Variables Directly

The IDL_Savefile object is designed to provide a way to query and restore *anything* contained within a SAVE file. As a result, the savefile object provides a mechanism for restoring heap variables independently of any regular IDL variables that may refer to them.

Note

The ability to directly restore heap variables independently of any referring regular variable is provided for completeness only. Restoring heap variables directly as described in this section is rarely (if ever) useful. The recommended way to restore a heap variable is to restore a regular IDL variable that refers to the heap variable.

To restore a heap variable independently of any regular variable, do the following:

1. Use either the POINTER_HEAPVAR or the OBJECT_HEAPVAR keyword (as appropriate) to the IDL_Savefile::Names method to retrieve the integer heap identifier of the heap variable.
2. Use either the POINTER_HEAPVAR or the OBJECT_HEAPVAR keyword (as appropriate) and the NEW_HEAPVAR keyword to the IDL_Savefile::Restore method to associate the restored heap variable with a new regular variable.

The following IDL code provides an example of how this might look:

```
; First create a pointer
myPtr = PTR_NEW('99.00')

; Create a SAVE file containing the pointer reference
SAVE, myPtr, FILE='ptr.sav'

; Reset the IDL session
.FULL_RESET_SESSION

; Create a savefile object from the SAVE file
sObj = OBJ_NEW('IDL_Savefile', 'ptr.sav')

; Retrieve the heap identifier of the saved pointer heap variable
ptrName = sObj->Names(/POINTER_HEAPVAR)

; Restore the heap variable, associating it with a new regular
; variable. Note that ptrName is (in this case) a one-element array.
sObj->Restore, ptrName[0], /POINTER_HEAPVAR, NEW_HEAPVAR=myNewPtr

; Display the contents of the new variable
PRINT, *myNewPtr
```

IDL Prints:

```
99.00
```

Tip

The above example is presented to depict the process of restoring a heap variable directly. It would have been much easier to simply restore the `myPtr` variable contained in the SAVE file.

Example

See “[Example](#)” on page 4194 for an example that restores variable data.

Version History

Introduced: 6.1

IDL_Savefile::Size

The IDL_Savefile::Size function method returns the size and type information for the specified variable, system variable, or heap variable in the SAVE file associated with the savefile object, if no keywords are set. If a keyword is set, IDL_Savefile::Size returns only the specified quantity.

Note

This method accepts the same keywords as the SIZE function, and returns the same information using the same formats. It differs only in that the argument is a string or integer identifier that specifies an item within a SAVE file, rather than an in-memory expression.

Syntax

Result = *Obj*->[IDL_Savefile:]Size(*SaveItem* [, *Keywords*])

Return Value

If no keywords are set, the Size method returns a vector of integer type. The first element is equal to the number of dimensions of *SaveItem*. This value is 0 (zero) if *SaveItem* is scalar or undefined. The next elements contain the size of each dimension, one element per dimension (none if *SaveItem* is scalar or undefined). After the dimension sizes, the last two elements contain the type code (zero if undefined) and the number of elements in *SaveItem*, respectively.

See “[SIZE](#)” on page 1926 for a listing of IDL type codes.

If a keyword is set, the Size method returns the specified information.

Arguments

SaveItem

A scalar value that specifies the item within the SAVE file for which size information is requested. If *SaveItem* is a string, it is the name of a variable or system variable contained in the SAVE file. If *SaveItem* is an integer, it is the unique heap variable identifier for a heap variable contained in the SAVE file.

Tip

Use the [IDL_Savefile::Names](#) function method to obtain the names or heap variable identifiers contained within a SAVE file before calling this method.

Keywords

IDL_Savefile::Size method accepts all of the keywords accepted by the [SIZE](#) function. See “[SIZE](#)” on page 1926 for details.

Example

See “[Example](#)” on page 4194.

Version History

Introduced: 6.1

See Also

[SIZE](#)

IDLcomActiveX

The IDLcomActiveX object class creates an IDL object that encapsulates an ActiveX control. IDL provides data type and other translation services, allowing IDL programs to access the ActiveX control's methods and properties using standard IDL syntax.

For detailed information on using the IDLcomActiveX object, see “[Using ActiveX Controls in IDL](#)” in Chapter 5 of the *External Development Guide* manual.

Superclasses

This class is a subclass of [IDLcomIDispatch](#).

Subclasses

When an ActiveX control is instantiated, IDL creates a dynamic subclass of the IDLcomActiveX class. The dynamic subclass is used to provide a unique name for each component type, based on the COM class or program identifier. See “[ActiveX Control Naming Scheme](#)” in Chapter 5 of the *External Development Guide* manual for details.

Creation

IDLcomActiveX objects are always created automatically by IDL, as a result of a call to the [WIDGET_ACTIVEX](#) function. They should never be created manually.

Properties

Objects of this class have no properties of their own.

Methods

The IDLcomActiveX object class is a direct subclass of the IDLcomIDispatch class, and relies entirely on the superclass methods, providing none of its own.

Version History

Introduced: 5.5

IDLcomActiveX Properties

Objects of this class have no properties of their own.

IDLcomIDispatch

The IDLcomIDispatch object class creates an IDL object that encapsulates a COM object. IDL provides data type and other translation services, allowing IDL programs to access the COM object's methods and properties using standard IDL syntax.

Note

COM objects encapsulated by IDLcomIDispatch objects must implement an IDispatch interface.

Note

You should be aware that COM, like C, stores array data in row-major format. IDL stores array data in column-major format. See “[Columns, Rows, and Array Majority](#)” in Chapter 7 of the *Building IDL Applications* manual for a detailed discussion of this issue and its implications for IDL application design.

For detailed information on using the IDLcomIDispatch object, see [Chapter 4](#), “[Using COM Objects in IDL](#)” in the *External Development Guide* manual.

Superclasses

This class has no superclasses.

Subclasses

When a COM object is instantiated, IDL creates a dynamic subclass of the IDLcomIDispatch class. The dynamic subclass is used to provide a unique name for each component type, based on the COM class or program identifier. See “[IDLcomIDispatch Object Naming Scheme](#)” in Chapter 4 of the *External Development Guide* manual for details.

ActiveX controls are instantiated within IDL as a special subclass of the IDLcomIDispatch class named IDLcomActiveX.

Creation

See [IDLcomIDispatch::Init](#)

Properties

Objects of this class have the following properties. See “[IDLcomIDispatch Properties](#)” on page 4219 for details on individual properties.

- *KEYWORD*

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

- [IDLcomIDispatch::Init](#)
- [IDLcomIDispatch::GetProperty](#)
- [IDLcomIDispatch::SetProperty](#)

In addition to these methods, you can call the underlying COM object’s methods directly. See “[Method Calls on IDLcomIDispatch Objects](#)” in Chapter 4 of the *External Development Guide* manual for details.

Note

The IDL object system uses method names to identify and call object lifecycle methods (Init and Cleanup). If the COM object underlying an IDLcomIDispatch object implements Init or Cleanup methods, they will be overridden by IDL’s lifecycle methods — the COM object’s methods will be inaccessible from IDL. Similarly, IDL implements the GetProperty and SetProperty methods for the IDLcomIDispatch object, so any methods of the underlying COM object that use these names will be inaccessible from IDL.

Version History

Introduced: 5.5

IDLcomIDispatch Properties

IDLcomIDispatch objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Get” column of the property table can be retrieved via “[IDLcomIDispatch::GetProperty](#)” on page 4220. Properties with the word “Yes” in the “Init” column of the property table can be set via “[IDLcomIDispatch::Init](#)” on page 4221. Properties with the word “Yes” in the “Set” column in the property table can be set via “[IDLcomIDispatch::SetProperty](#)” on page 4222.

KEYWORD

KEYWORD is a string containing the name of one of the underlying COM object’s properties, and *Variable* is the name of an IDL variable that will contain the retrieved property value. You can get multiple property values in a single statement by supplying multiple *KEYWORD=Variable* pairs.

Note

KEYWORD must map exactly to the full name of the underlying COM object’s property method. The partial keyword name functionality provided by IDL is not valid with IDLcomIDispatch objects.

IDLcomIDispatch::GetProperty

The IDLcomIDispatch::GetProperty function method is used to get properties from the COM object that underlies an IDLcomIDispatch object. The COM object's property names are represented as IDL keywords to the GetProperty method, and property values are treated as IDL keyword values. See “[Managing COM Object Properties](#)” in Chapter 4 of the *External Development Guide* manual.

Syntax

Obj->[IDLcomIDispatch::]GetProperty, *KEYWORD=Variable*, [*arg0*, *arg1*, ...]

Arguments

Because some of the underlying COM object's property methods may require arguments, the GetProperty method will accept optional arguments. The values of the arguments themselves will depend on the COM object that underlies the IDLcomIDispatch object.

Note

If arguments are required, you can only specify one property to retrieve in a given call to the GetProperty method.

Keywords

Any property listed under “[IDLcomIDispatch Properties](#)” on page 4219 that contains the word “Yes” in the “Get” column of the properties table can be retrieved using this method. To retrieve the value of a property, specify the property name as a keyword set equal to a named variable that will contain the value of the property.

Version History

Introduced: 5.5

IDLcomIDDispatch::Init

The IDLcomIDDispatch::Init function method is used to initialize a given COM object and establish a link between the resulting IDL object and the IDispatch interface of the underlying COM object.

Syntax

```
Obj = OBJ_NEW('IDLcomIDDispatch$ID_type$ID')
```

or

```
Result = Obj->[IDLcomIDDispatch$ID_type$ID::]Init( ) (Only in a subclass' Init method)
```

where *ID_type* is one of the following:

- CLSID if the object is identified by its COM class ID, or
- PROGID if the object is identified by its COM program ID,

and *ID* is the COM object's actual class or program identifier string. If the COM object's class identifier string is used to create the IDLcomIDDispatch object, the braces ({ }) must be removed and the hyphens replaced by underscores. For details on constructing the class name, see “[IDLcomIDDispatch Object Naming Scheme](#)” in Chapter 4 of the *External Development Guide* manual for details.

Note

While COM objects incorporated into IDL are instances of the dynamic subclass created when the COM object is instantiated, they still expose the functionality of the class IDLcomIDDispatch, which is the direct superclass of the dynamic subclass. All IDLcomIDDispatch methods are available to the dynamic subclass.

Arguments

None.

Keywords

None.

Version History

Introduced: 5.5

IDLcomIDispatch:: SetProperty

The IDLcomIDispatch::SetProperty function method is used to set properties for the COM object that underlies an IDLcomIDispatch object. The COM object's property names are represented as IDL keywords to the SetProperty method, and property values are treated as IDL keyword values. See [“Managing COM Object Properties”](#) in Chapter 4 of the *External Development Guide* manual for details.

Syntax

Obj->[IDLcomIDispatch::]SetProperty, *KEYWORD*=*Expression*

Arguments

None.

Keywords

Any property listed under [“IDLcomIDispatch Properties”](#) on page 4219 that contains the word “Yes” in the “Set” column of the properties table can be set using this method. To set the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.5

IDLjavaObject

The IDLjavaObject class creates an IDL object that encapsulates a Java object. IDL provides data type and other translation services, allowing IDL programs to access the Java object's methods and properties using standard IDL syntax.

For detailed information on using the IDLjavaObject class, see [Chapter 8, “Using Java Objects in IDL”](#) in the *External Development Guide* manual.

Superclasses

This class has no superclasses.

Creation

See [“IDLjavaObject::Init”](#) on page 4227.

Properties

The properties of this object depends on which Java object the IDLjavaObject class encapsulates. See [“IDLjavaObject Properties”](#) on page 4225 for more details.

Methods

This class has the following methods:

- [IDLjavaObject::GetProperty](#)
- [IDLjavaObject::Init](#)
- [IDLjavaObject::SetProperty](#)

Note

In addition to these methods, you can call the underlying Java object's methods directly. See [“Method Calls on IDL-Java Objects”](#) in Chapter 8 of the *External Development Guide* manual for details. The IDL object system uses method names to identify and call object *lifecycle methods* (Init and Cleanup). If the Java object underlying IDLjavaObject implements Init or Cleanup methods, they will be overridden by IDL's lifecycle methods — the Java object's methods will be inaccessible from IDL. Similarly, IDL implements the GetProperty and SetProperty methods for IDLjavaObject, so any methods of the underlying Java object that use

these names will be inaccessible from IDL. In Java, you can wrap these methods with different named methods to work around this limitation.

Examples

See [Chapter 8, “Using Java Objects in IDL”](#) in the *External Development Guide* manual.

Version History

Introduced: 6.0

IDLjavaObject Properties

The property name of an instance of `IDLjavaObject` is passed to the IDL Java subsystem and is used in conjunction with the Java reflection API to access the related data member on the underlying object. The data member (property) is identified through the arguments to the `IDLjavaObject::GetProperty` and `IDLjavaObject::SetProperty` methods.

IDLjavaObject::GetProperty

The IDLjavaObject::GetProperty procedure method retrieves properties (known as data members in Java) from the Java object that underlies the IDLjavaObject. The Java object's property names are represented as IDL keywords to the GetProperty method, and property values are treated as IDL keyword values. See “[Managing IDL-Java Object Properties](#)” in Chapter 8 of the *External Development Guide* manual.

Syntax

```
Obj->[IDLjavaObject::]GetProperty [, PROPERTY=variable]
```

Arguments

None

Keywords

PROPERTY

The Java object property names are mapped to IDL keywords. The underlying property values are treated as IDL keyword values, which is the same convention for other IDL objects.

Note

The provided keywords must map directly to a property name or IDL issues an error. Any keyword passed into either of the property routines is assumed to be a fully-qualified Java property name. As such, the partial keyword name functionality provided by IDL is not valid with IDL Java based objects.

The *variable* may be an IDL primitive type, an instance of IDLJavaObject, or an array of IDL primitive types. See “[IDL-Java Bridge Data Type Mapping](#)” in Chapter 8 of the *External Development Guide* manual for more information.

Note

Besides other Java based objects, no complex types (structures, pointers, etc.) are supported as parameters to method calls.

Version History

Introduced: 6.0

IDLjavaObject::Init

The IDLjavaObject::Init function method instantiates the given Java object and establishes a link between the resulting IDL object with the underlying Java object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: the Init method can be called from within the Init method of a subclass.

Syntax

Obj = OBJ_NEW('IDLjavaObject\$JAVACLASSNAME', *JavaClassName*[, *Arg1*, ...])

where *JAVACLASSNAME* is a case-insensitive string representing the Java class to be instantiated, see “[IDL-Java Bridge Architecture](#)” in Chapter 8 of the *External Development Guide* manual for more information. All periods in *JAVACLASSNAME* must be converted to underscores.

Return Value

Returns an object reference to the newly-created object if initialization is successful, or a null object if initialization fails.

Arguments

JavaClassName

A string representing the Java class to be instantiated. See “[Java Class Names in IDL](#)” in Chapter 8 of the *External Development Guide* manual for more details.

Arg1, ...

Additional arguments as required by the constructor. These arguments are passed to the underlying Java constructor. Arguments may be IDL primitive types, IDL strings, IDLJavaObjects, and arrays of the previous types. See “[IDL-Java Bridge Data Type Mapping](#)” in Chapter 8 of the *External Development Guide* manual for more information.

Keywords

None

Version History

Introduced: 6.0

IDLjavaObject:: SetProperty

The IDLjavaObject::SetProperty procedure method sets properties (known as data members in Java) for the Java object that underlies an instance of IDLjavaObject. The Java object's property names are represented as IDL keywords to the SetProperty method, and property values are treated as IDL keyword values. See “[Managing IDL-Java Object Properties](#)” in Chapter 8 of the *External Development Guide* manual for more details.

Syntax

```
Obj->[IDLjavaObject::]SetProperty [, PROPERTY=value]
```

Arguments

None

Keywords

PROPERTY

The Java object property names are mapped to IDL keywords. The underlying property values are treated as IDL keyword values, which is the same convention for other IDL objects.

Note

The provided keywords must map directly to a property name or IDL issues an error. Any keyword passed into either of the property routines is assumed to be a fully-qualified Java property name. As such, the partial keyword name functionality provided by IDL is not valid with IDL Java based objects.

The *value* may be an IDL primitive type, an instance of IDLJavaObject, or an array of IDL primitive types. See “[IDL-Java Bridge Data Type Mapping](#)” in Chapter 8 of the *External Development Guide* manual for more information.

Note

Besides other Java based objects, no complex types (structures, pointers, etc.) are supported as parameters to method calls.

Version History

Introduced: 6.0

TrackBall

A TrackBall object translates widget events from a draw widget (created with the `WIDGET_DRAW` function) into transformations that emulate a virtual trackball (for transforming object graphics in three dimensions).

This object class is implemented in the IDL language. Its source code can be found in the file `trackball__define.pro` in the `lib` subdirectory of the IDL distribution.

Superclasses

This class has no superclasses.

Creation

See [“TrackBall::Init”](#) on page 4234.

Properties

Objects of this class have the following properties. See [“TrackBall Properties”](#) on page 4232 for details on individual properties.

- [AXIS](#)
- [CONSTRAIN](#)
- [MOUSE](#)

In addition, objects of this class inherit the properties of all superclasses of this class.

Methods

This class has the following methods:

- [TrackBall::Init](#)
- [TrackBall::Reset](#)
- [TrackBall::Update](#)

In addition, this class inherits the methods of its superclasses (if any).

Version History

Introduced: 5.0

TrackBall Properties

TrackBall objects have the following properties in addition to properties inherited from any superclasses. Properties with the word “Yes” in the “Init” column of the property table can be set via “[TrackBall::Init](#)” on page 4234.

Note

For a discussion of the property description tables shown below, see “[About Object Property Descriptions](#)” on page 2671.

AXIS

An integer value that indicates the axis about which rotations are to be constrained if the CONSTRAIN keyword is set. Valid values include:

- 0 = Rotate only around the *X* axis.
- 1 = Rotate only around the *Y* axis.
- 2 = Rotate only around the *Z* axis (this is the default).

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

CONSTRAIN

A boolean value that indicates whether the trackball transformations are to be constrained about the axis specified by the AXIS keyword. The default is not to constrain the transformations.

Property Type	Boolean		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

MOUSE

An integer value that indicates which mouse button to honor for trackball events. Valid values include:

- 1 = Left mouse button (the default)
- 2 = Middle mouse button
- 4 = Right mouse button

Property Type	Integer		
Name String	<i>not displayed</i>		
Get: No	Set: No	Init: Yes	Registered: No

TrackBall::Init

The TrackBall::Init function method initializes the TrackBall object.

Syntax

```
Obj = OBJ_NEW('TrackBall', Center, Radius [, AXIS={0 | 1 | 2}] [, /CONSTRAIN]
  [, MOUSE={1 | 2 | 4}])
```

or

```
Result = Obj->[TrackBall::]Init( Center, Radius ) (Only in a subclass' Init method.)
```

Note

Keywords can be used in either form. They are omitted in the second form for brevity.

Arguments

Center

A two-element vector, [X, Y], specifying the center coordinates of the trackball. X and Y should be specified in device units.

Radius

The radius of the trackball, specified in device units.

Keywords

Any property listed under [“TrackBall Properties”](#) on page 4232 that contains the word “Yes” in the “Init” column of the properties table can be initialized during object creation using this method. To initialize the value of a property, specify the property name as a keyword set equal to the appropriate property value.

Version History

Introduced: 5.0

TrackBall::Reset

The TrackBall::Reset procedure method resets the state of the TrackBall object.

Syntax

```
Obj->[TrackBall::]Reset, Center, Radius [, AXIS={0 | 1 | 2}] [, /CONSTRAIN]  
[, MOUSE={1 | 2 | 4}]
```

Arguments

Center

A two-element vector, [X, Y], specifying the center coordinates of the trackball. X and Y should be specified in device units.

Radius

The radius of the trackball, specified in device units.

Keywords

AXIS

Set this keyword to an integer value to indicate the axis about which rotations are to be constrained if the CONSTRAIN keyword is set. Valid values include:

- 0 = Rotate only around the X axis.
- 1 = Rotate only around the Y axis.
- 2 = Rotate only around the Z axis (this is the default).

CONSTRAIN

Set this keyword to indicate that the trackball transformations are to be constrained about the axis specified by the AXIS keyword. The default is not to constrain the transformations.

MOUSE

Set this keyword to an integer value to indicate which mouse button to honor for trackball events. Valid values include:

- 1 = Left mouse button (the default)
- 2 = Middle mouse button
- 4 = Right mouse button

Version History

Introduced: 5.0

TrackBall::Update

The TrackBall::Update function method updates the state of the TrackBall object based on the information contained in the input widget event structure. The return value is nonzero if a transformation matrix is calculated as a result of the event, or zero otherwise.

Syntax

```
Result = Obj->[TrackBall::]Update( sEvent [, MOUSE={1 | 2 | 4}]  
    [, TRANSFORM=variable] [, /TRANSLATE] )
```

Arguments

sEvent

The widget event structure.

Keywords

MOUSE

Set this keyword to an integer value to indicate which mouse button to honor for trackball events. Valid values include:

- 1 = Left mouse button (the default)
- 2 = Middle mouse button
- 4 = Right mouse button

TRANSFORM

Set this keyword to a named variable that will contain a 4 x 4 element floating-point array if a new transformations matrix is calculated as a result of the widget event.

TRANSLATE

Set this keyword to indicate that the trackball movement should be constrained to translation in the X-Y plane rather than rotation about an axis.

Example

The example code below provides a skeleton for a widget-based application that uses the `TrackBall` object to interactively change the orientation of graphics.

Create a trackball centered on a 512x512 pixel drawable area, and a view containing the model to be manipulated:

```

xdim = 512
ydim = 512
wBase = WIDGET_BASE()
wDraw = WIDGET_DRAW(wBase, XSIZE=xdim, YSIZE=ydim, $
    GRAPHICS_LEVEL=2, /BUTTON_EVENTS, $
    /MOTION_EVENTS, /EXPOSE_EVENTS, RETAIN=0 )
WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wDraw, GET_VALUE=oWindow

oTrackball = OBJ_NEW('Trackball', [xdim/2.,ydim/2.], xdim/2.)
oView = OBJ_NEW('IDLgrView')
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel
XMANAGER, 'TrackEx', wBase

```

You must handle the trackball updates in the widget event-handling code. As the trackball transformation changes, update the transformation for the model object, and redraw the view:

```

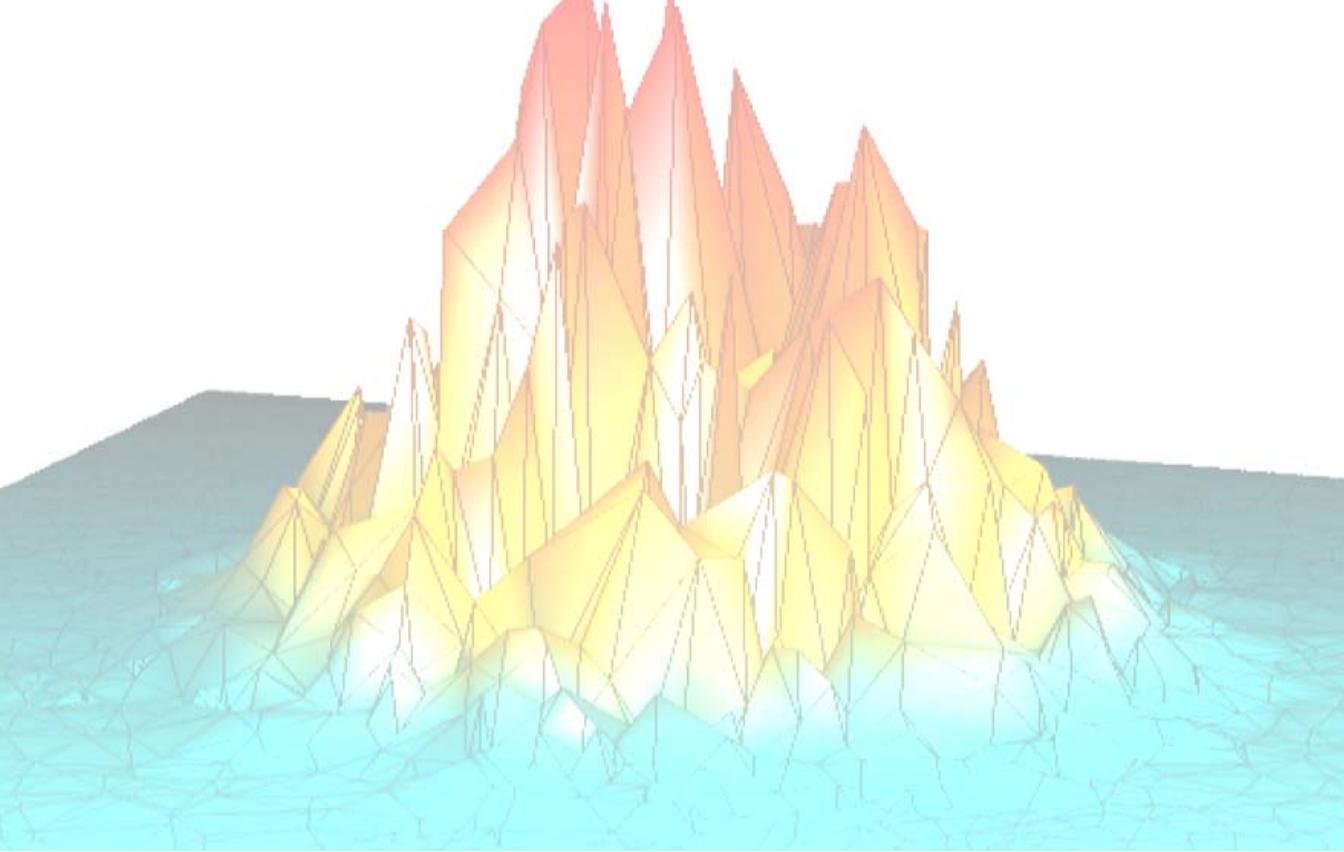
PRO TrackEx_Event, sEvent
...
bHaveXform = oTrackball->Update( sEvent, TRANSFORM=TrackXform )
IF (bHaveXform) THEN BEGIN
oModel->GetProperty, TRANSFORM=ModelXform
oModel->SetProperty, TRANSFORM=ModelXform # TrackXform
oWindow->Draw, oView
ENDIF
...
END

```

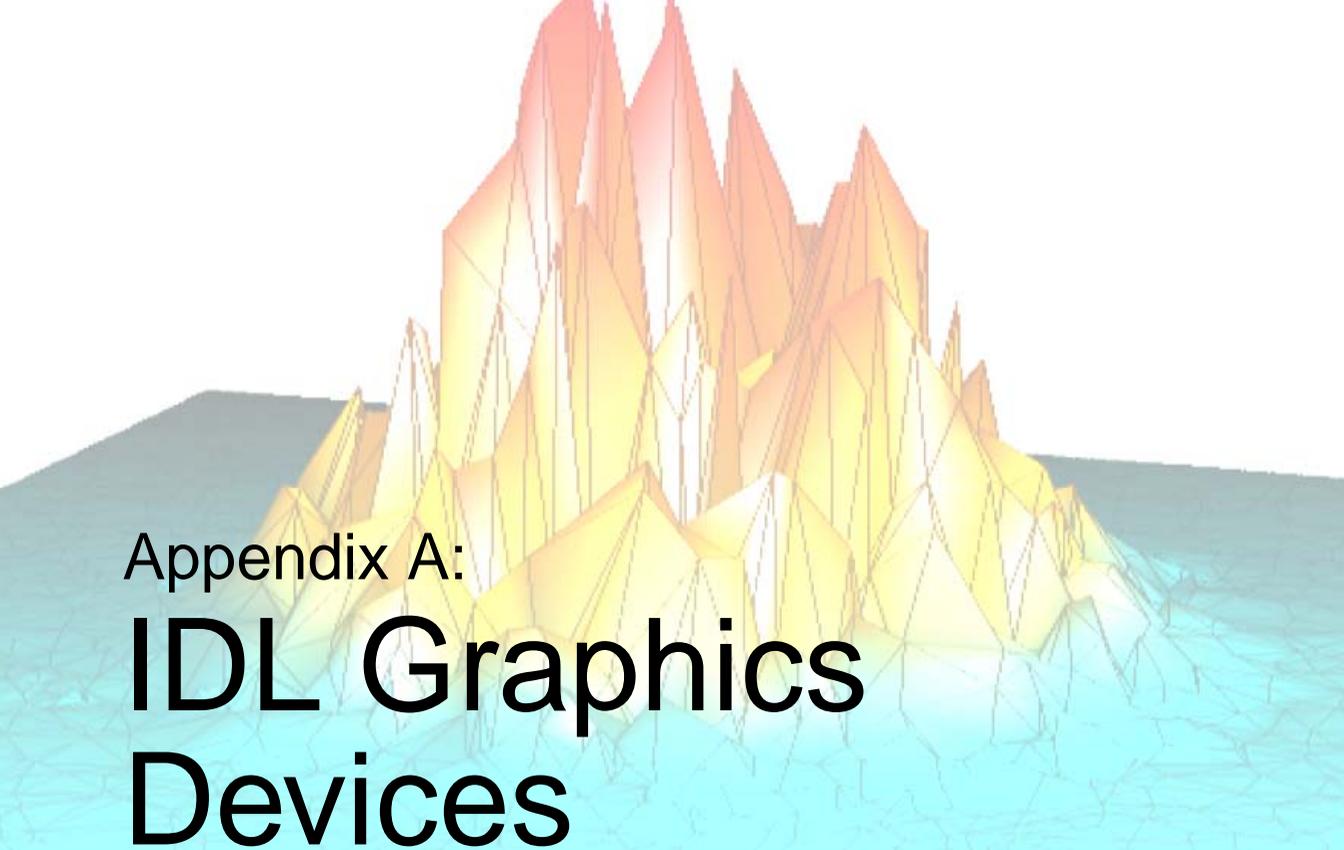
For a complete example, see the file `surf_track.pro`, located in the `examples/visual` subdirectory of the IDL distribution. The `SURF_TRACK` procedure uses IDL widgets to create a graphical user interface to an object tree, creates a surface object from user-specified data (or from default data, if none is specified), and places the surface object in an IDL draw widget. The `SURF_TRACK` interface allows the user to specify several attributes of the object hierarchy via pulldown menus.

Version History

Introduced: 5.0



Part III: Appendices



Appendix A:

IDL Graphics Devices

The following topics are covered in this appendix:

Supported Devices	4244	The PCL Device	4299
Keywords Accepted by the IDL Devices	4246	The Printer Device	4301
Window Systems	4286	The PostScript Device	4302
Printing Graphics Output Files	4289	The Regis Terminal Device	4315
The CGM Device	4292	The Tektronix Device	4316
The HP-GL Device	4294	The Microsoft Windows Device	4318
The Metafile Display Device	4296	The X Windows Device	4319
The Null Display Device	4298	The Z-Buffer Device	4328

Supported Devices

IDL Direct Graphics support graphic output to the devices listed below:

Device Name	Description
CGM	Computer Graphics Metafile
HP	Hewlett-Packard Graphics Language (HP-GL)
METAFILE	Windows Metafile Format (WMF)
NULL	No graphics output
PCL	Hewlett-Packard Printer Control Language (PCL)
PRINTER	System printer
PS	PostScript
REGIS	Regis graphics protocol (DEC systems only)
TEK	Tektronix compatible terminal
WIN	Microsoft Windows
X	X Window System
Z	Z-buffer pseudo device

Table A-2: IDL Graphics Output Devices

Each of these devices is described in a section of this chapter. The `SET_PLOT` procedure can be used to select the graphic device to which IDL directs its output. IDL Object Graphics does not rely on the concept of a current graphics device; see *Using IDL* for details about IDL Object Graphics.

The `DEVICE` procedure controls the graphic device-specific functions. An attempt has been made to isolate all device-specific functions in this procedure. `DEVICE` controls the graphics device currently selected by `SET_PLOT`. When using `DEVICE`, it is important to make sure that the current graphics device is the one you intend to use. This is because most of the devices have different keywords—you will most likely get a “Keyword ... not allowed in call to: Device” error if you call `DEVICE` when the wrong device is selected.

Obsolete Graphics Devices and Device Keywords

The following graphics devices are obsolete:

- LJ
- MAC

For information on obsolete graphics devices, See [Appendix I, “Obsolete Features”](#).

The following graphics device keywords are obsolete:

- DEPTH
- FONT

For information on obsolete keywords, See [Appendix I, “Obsolete Features”](#).

Keywords Accepted by the IDL Devices

The following table indicates which keywords are accepted by the DEVICE procedure. The NULL device is not listed as it accepts no keywords. Details of the various keywords can be found on the page indicated in the table.

Note

Most keywords to the DEVICE procedure are sticky — that is, once you set them, they remain in effect until you explicitly change them again, or end your IDL session. The exceptions are keywords used to return a value from the system (GET_FONTNAMES, for example) and those that perform a one-time-only operation (CLOSE_FILE, for example).

Keywords	Devices										
	CGM	HP	METAFILE	PCL	PRINTER	PS	REGIS	TEK	WIN	X	Z
AVANTGARDE						•					
AVERAGE_LINES							•				
BINARY	•										
BITS_PER_PIXEL						•					
BKMAN						•					
BOLD						•					
BOOK						•					
BYPASS_TRANSLATION									•	•	
CLOSE											•
CLOSE_DOCUMENT					•						
CLOSE_FILE	•	•	•	•		•	•	•			

Table A-3: Keywords accepted by the IDL devices

Keywords	Devices										
	CGM	HP	METAFILE	PCL	PRINTER	PS	REGIS	TEK	WIN	X	Z
CMYK						•					
COLOR				•		•					
COLORS	•							•			
COPY									•	•	
COURIER						•					
CURSOR_CROSSHAIR									•	•	
CURSOR_IMAGE									•	•	
CURSOR_MASK									•	•	
CURSOR_ORIGINAL									•	•	
CURSOR_STANDARD									•	•	
CURSOR_XY									•	•	
DECOMPOSED									•	•	
DEMI						•					
DIRECT_COLOR										•	
EJECT		•									
ENCAPSULATED						•					
ENCODING	•										
FILENAME	•	•	•	•		•	•	•			
FLOYD				•						•	
FONT_INDEX						•					
FONT_SIZE						•					

Table A-3: Keywords accepted by the IDL devices (Continued)

Keywords	Devices										
	CGM	HP	METAFILE	PCL	PRINTER	PS	REGIS	TEK	WIN	X	Z
GET_CURRENT_FONT			•		•				•	•	
GET_DECOMPOSED									•	•	
GET_FONTNAMES			•		•				•	•	
GET_FONTNUM			•		•				•	•	
GET_GRAPHICS_FUNCTION									•	•	•
GET_PAGE_SIZE					•						
GET_SCREEN_SIZE									•	•	
GET_VISUAL_DEPTH									•	•	
GET_VISUAL_NAME									•	•	
GET_WINDOW_POSITION									•	•	
GET_WRITE_MASK										•	•
GIN_CHARS								•			
GLYPH_CACHE			•		•	•			•		•
HELVETICA						•					
INCHES		•	•	•	•	•					
INDEX_COLOR			•		•						
ISOLATIN1						•					
ITALIC						•					
LANDSCAPE		•		•	•	•					
LANGUAGE_LEVEL						•					
LIGHT						•					

Table A-3: Keywords accepted by the IDL devices (Continued)

Keywords	Devices										
	CGM	HP	METAFILE	PCL	PRINTER	PS	REGIS	TEK	WIN	X	Z
MEDIUM						•					
NARROW						•					
NCAR	•										
OBLIQUE						•					
OPTIMIZE				•							
ORDERED				•						•	
OUTPUT		•				•					
PALATINO						•					
PIXELS				•							
PLOT_TO							•	•			
PLOTTER_ON_OFF		•									
POLYFILL		•									
PORTRAIT		•		•	•	•					
PRE_DEPTH						•					
PRE_XSIZE						•					
PRE_YSIZE						•					
PREVIEW						•					
PRINT_FILE									•		
PSEUDO_COLOR										•	
RESET_STRING								•			
RESOLUTION				•							

Table A-3: Keywords accepted by the IDL devices (Continued)

Keywords	Devices										
	CGM	HP	METAFILE	PCL	PRINTER	PS	REGIS	TEK	WIN	X	Z
RETAIN									•	•	
SCALE_FACTOR					•	•					
SCHOOLBOOK						•					
SET_CHARACTER_SIZE	•	•	•	•	•	•	•	•	•	•	•
SET_COLORMAP				•							
SET_COLORS											•
SET_FONT			•		•	•			•	•	•
SET_GRAPHICS_FUNCTION									•	•	•
SET_RESOLUTION											•
SET_STRING								•			
SET_TRANSLATION										•	
SET_WRITE_MASK										•	•
STATIC_COLOR										•	
STATIC_GRAY										•	
SYMBOL						•					
TEK4014								•			
TEK4100								•			
TEXT	•										
THRESHOLD				•						•	
TIMES						•					
TRANSLATION									•	•	

Table A-3: Keywords accepted by the IDL devices (Continued)

Keywords	Devices										
	CGM	HP	METAFILE	PCL	PRINTER	PS	REGIS	TEK	WIN	X	Z
TRUE_COLOR			•		•					•	
TT_FONT			•		•	•			•	•	•
TTY							•	•			
USER_FONT						•					
VT240, VT241							•				
VT340, VT341							•				
WINDOW_STATE									•	•	
XOFFSET		•		•	•	•					
XON_XOFF		•									
XSIZE		•	•	•	•	•					
YOFFSET		•		•	•	•					
YSIZE		•	•	•	•	•					
ZAPFCHANCERY						•					
ZAPFDINGBATS						•					
Z_BUFFERING											•

Table A-3: Keywords accepted by the IDL devices (Continued)

Keywords accepted by the DEVICE command are described below. A list of devices that accept the keyword is included in parentheses below the keyword name.

AVANTGARDE

(PS)

Set this keyword to select the ITC Avant Garde PostScript font.

AVERAGE_LINES

(REGIS)

Controls the method of writing images to the VT240. If this keyword is set, (default setting), even and odd pairs of image lines are averaged and written to a single line. If clear, each image line is written to the screen. See the discussion below. This keyword has no effect when using a VT300 series terminal.

BINARY

(CGM)

Set this keyword to set the encoding type for the CGM output file to binary.

BITS_PER_PIXEL

(PS)

IDL is capable of producing PostScript images with 1, 2, 4, or 8 bits per pixel. Using more bits per pixel gives higher resolution at the cost of generating larger files.

`BITS_PER_PIXEL` is used to specify the number of bits to use. If you do not specify a value for `BITS_PER_PIXEL`, a default value of 4 is used.

It should be noted that many laser printers, including the original Apple Laserwriter are capable of only 32 different shades of gray (which can be represented by 5 bits). Thus, specifying 8 bits per pixel does not give 256 apparent shades of grey as might be expected, only 32, at a cost of sending twice the number of bits to the printer. Often, 4 bits (16 levels of gray) will give acceptable results with a large savings in file size.

BKMAN

(PS)

Set this keyword to select the ITC Bookman PostScript font.

BOLD

(PS)

Set this keyword to specify that the bold version of the current PostScript font should be used.

BOOK

(PS)

Set this keyword to specify that the book version of the current PostScript font should be used.

BYPASS_TRANSLATION

(WIN, X)

Set this keyword to bypass the translation tables, allowing direct specification of color indices. See [“Color Translation”](#) on page 4323. Pixel values read via the TVRD function are not translated if this keyword is set, and the result contains the byte value of the actual pixel values present in the display.

By default, the color translation tables are bypassed if the visual type is TrueColor or the visual type is DirectColor with a private colormap.

This keyword is accepted by the WIN device (for compatibility with the X device), but has no effect when set.

CLOSE

(Z)

Set this keyword to deallocate the memory used by the Z-buffer. The Z-buffer device is reinitialized if subsequent graphics operations are directed to the device.

CLOSE_DOCUMENT

(PRINTER)

Set this keyword to have IDL send any buffered output to the currently selected printer. This keyword is applicable only when the printer device is selected. See [“The Printer Device”](#) on page 4301 for details.

CLOSE_FILE

(CGM, HP, METAFILE, PCL, PS, REGIS, TEK)

Set this keyword to have IDL output any buffered commands and close the current graphics file.

Caution: If you close the output file and then cause IDL to produce more output (e.g., by executing a new PLOT command), IDL will open the file again, causing the contents of the recently closed file to be lost. To avoid this, use the FILENAME keyword to specify a different file name or use SET_PLOT to disable the graphics driver, or be sure to print the closed output file before creating more output.

See the discussion of printing output files in [“Printing Graphics Output Files”](#) on page 4289.

CMYK

(PS)

Set this keyword to generate PostScript output using the CMYK (cyan, magenta, yellow, and black) color model. The default PostScript color model is RGB (red, green, blue). PostScript CMYK files require a PostScript Language Level 2 interpreter. Therefore, when this keyword is set, the `LANGUAGE_LEVEL` keyword value is set to 2. See “[CMYK Output](#)” on page 4305 for additional information on CMYK output with a color PostScript device.

COLOR

(PCL, PS)

Set this keyword to enable color PCL or PostScript output. See “[The PCL Device](#)” on page 4299 or “[The PostScript Device](#)” on page 4302.

COLORS

(CGM, TEK)

This keyword specifies the maximum number of colors and the size of the color table used for output. The value of the system variable fields `!D.N_COLORS` and `!D.TABLE_SIZE` are set to this value and `!P.COLOR` is set to one less than this value.

For Tektronix Terminals Only

This keyword sets the number of colors supported by a 4100 series terminal. For example, if your terminal has 4-bit planes, the number of colors is $2^4 = 16$:

```
DEVICE, COLORS = 16
```

Valid values of this parameter are: 2, 4, 8, 16, or 64; other values will cause problems. Some Tektronix terminals will not operate properly if this parameter does not exactly match the number of colors available in the terminal hardware.

This parameter sets the field `!D.N_COLORS`, which affects the loading of color tables, the scaling used by the `TVSCL` procedure, and the number of bits output by the `TV` procedure to the terminal. It also changes the default color, `!P.COLOR` to the number of colors minus one.

COPY

(WIN, X)

Use this keyword to copy a rectangular area of pixels from one region of a window to another. `COPY` should be set a six or seven element array: $[X_s, Y_s, N_x, N_y, X_d, Y_d, W]$,

where: (X_s, Y_s) is the lower left corner of the source rectangle, (N_x, N_y) are the number of columns and rows in the rectangle, and (X_d, Y_d) is the coordinate of the destination rectangle. Optionally, W is the index of the window *from which the pixels should be copied to the current* window. If it is not supplied, the current window is used as both the source and destination.

COURIER

(PS)

Set this keyword to select the Courier PostScript font.

CURSOR_CROSSHAIR

(WIN, X)

Set this keyword to selects the crosshair cursor type. This is the IDL default.

CURSOR_IMAGE

(WIN, X)

Specifies the cursor pattern. The value of this keyword must be a 16-line by 16-column bitmap, contained in a 16-element short integer vector. The offset from the upper left pixel to the point that is considered the hot spot can be provided via the CURSOR_XY keyword.

CURSOR_MASK

(WIN, X)

When the CURSOR_IMAGE keyword is used to specify a cursor bitmap, the CURSOR_MASK keyword can be used to simultaneously specify the mask that should be used. In the mask, bits that are set indicate bits in the CURSOR_IMAGE that should be seen and bits that are not set are masked out.

By default, the CURSOR_IMAGE bitmap is used for both the image and the mask. This can cause the cursor to be invisible on a black background (because only black pixels are allowed to be displayed).

CURSOR_ORIGINAL

(WIN, X)

Set this keyword to select the window system's default cursor. Under X Windows, it is the cursor in use by the root window when IDL starts. For the Microsoft Windows device, it is the arrow pointer.

CURSOR_STANDARD

(WIN, X)

This keyword can be used to change the cursor appearance in IDL graphics windows.

For X Windows

This keyword selects one of the predefined cursors provided by the X Window system. The available cursors shapes are defined in the file `cursorfont.h` in the directory `/usr/include/X11`. In order to use one of these cursors, you select the number of the cursor and provide it as the value of the `CURSOR_STANDARD` keyword. For example, the file gives the value of `XC_CROSS` as being 30. In order to make that the current cursor, use the statement:

```
DEVICE, CURSOR_STANDARD=30
```

For Microsoft Windows

The table below shows the values for `CURSOR_STANDARD` that result in different cursor shapes. For example, to change the cursor to an “I-beam” when the cursor is in an IDL graphics window, use the command:

```
DEVICE, CURSOR_STANDARD = 32513
```

Value	Cursor Shape
32512	Arrow
32513	I-Beam
32514	Hourglass
32515	Black Crosshair
32516	Up Arrow
32640	Size (Windows only)
32641	Icon (Windows only)
32642	Size NW-SE
32643	Size NE-SW
32644	Size E-W
32645	Size N-S

Table A-4: Values for the WIN device CURSOR_STANDARD Keyword

CURSOR_XY

(WIN, X)

A two element integer vector giving the (X, Y) pixel offset of the cursor hot spot, the point which is considered to be the mouse position, from the lower left corner of the cursor image. This parameter is only applicable if `CURSOR_IMAGE` is provided. The cursor image is displayed top-down—the first row is displayed at the top.

DECOMPOSED

(WIN, X)

This keyword is used to control the way in which graphics color index values are interpreted when using displays with decomposed color (TrueColor or DirectColor visuals). This keyword has no effect with other types of visuals.

Set this keyword to 1 to cause color indices to be interpreted as 3, 8-bit color indices where the least-significant 8 bits contain the red value, the next 8 bits contain the green value, and the most-significant 8 bits contain the blue value. This is the way IDL has always interpreted pixels when using visual classes with decomposed color.

Set this keyword to 0 to cause the least-significant 8 bits of the color index value to be interpreted as a PseudoColor index. This setting allows users with DirectColor and TrueColor displays to use IDL programs written for standard, PseudoColor displays without modification.

In older versions of IDL, color index values higher than `!D.N_COLORS-1` were clipped to `!D.N_COLORS-1` in the higher level graphics routines. In some cases, this clipping caused the exclusive-OR graphics mode to malfunction with raster displays. This clipping has been removed. Programs that incorrectly specified color indices higher than `!D.N_COLORS-1` will now probably exhibit different behavior.

DEMI

(PS)

Set this keyword to specify that the demi version of the current PostScript font should be used.

DIRECT_COLOR

(X)

Set this keyword to select the DirectColor visual. The value of the keyword represents the number of bits per pixel. This keyword has effect only if no windows have been

created. Visual classes are discussed in more detail in “[X Windows Visuals](#)” on page 4319.

EJECT

(HP)

In order to perform an erase operation on a plotter, it is necessary to remove the current sheet of paper and load a fresh sheet. The ability of various plotters to do this varies, so the EJECT keyword allows you to specify what should be done. The following table describes the possible values.

Value	Meaning
0	Do nothing. Note that this is likely to cause one page to plot over the previous one, so you should limit yourself to one page of output per file. This is the default.
1	Use the sheet feeder to load the next page.
2	Put the plotter off-line at the beginning of each page after the first.

Table A-5: Values for the HP-GL Eject Keyword

Many HP-GL plotters lack a sheet feeder, and require the user to load the next page manually. Therefore, the default action is for IDL to not issue any page eject instructions. In this case, you must restrict yourself to generating only a single plot at a time. If your plotter has a sheet feeder, you will want to issue the command:

```
DEVICE, /EJECT
```

to tell IDL that it should use the sheet feeder instead of placing the plotter off-line.

If your plotter does not have a sheet feeder, but it does understand the HP-GL NR command, use the command:

```
DEVICE, EJECT=2
```

to place the plotter off-line at the start of every plot except the first one. This causes the plotter to wait between plots for the user to replace the paper. When the user puts the plotter back on-line, the graphics commands for the new page are executed by the plotter. Consult the programming manual for your plotter to determine if this instruction is provided.

ENCAPSULATED

(PS)

Set this keyword to create an encapsulated PostScript file, suitable for importing into another document (e.g., a LaTeX or FrameMaker document).

Note

You must explicitly set this keyword to zero to create “regular” PostScript output after creating encapsulated output. (That is, like most keyword settings to the DEVICE procedure, the setting “sticks” until you change it, or until you quit IDL.)

Normally, IDL assumes that its PostScript-generated output will be sent directly to a printer. It therefore includes PostScript commands to position the plot on the page and to eject the page from the printer. These commands are undesirable if the output is going to be inserted into the middle of another PostScript document. If ENCAPSULATED is present and non-zero, IDL does not generate these commands.

IDL follows the standard PostScript convention for encapsulated files. It assumes the standard PostScript scaling is in effect (72 points per inch). In addition, it declares the size, or *bounding box* of the plotting region at the top of the output file. This size is determined when the output file is opened (when the first graphics command is given), by multiplying the size of the plotting region (as specified with the XSIZE and YSIZE keywords) by the current scale factor (as specified by the SCALE_FACTOR keyword).

Changing the size of the plotting region or scale factor once graphics have been output will not be reflected in the declared bounding box, and will confuse programs that attempt to import the resulting graphics. Therefore, when generating encapsulated PostScript, do not change the plot region size or scaling factor once any graphics commands have been issued. If you need to change these parameters, use the FILENAME keyword to start a new file.

ENCODING

(CGM)

Set this keyword to set the CGM encoding type for the output file. Valid values are:

Value	Description
1	Binary encoding - the default.
2	Text encoding.
3	NCAR binary encoding.

Table A-6: CGM Encoding Values

The encoding type can only be changed when no CGM file is open.

FILENAME

(CGM, HP, METAFILE, PCL, PS, REGIS, TEK)

Normally, all generated output is sent to a file named `idl.xxx`, where `xxx` is the lowercase name of the device shown in the table under “Supported Devices” on page 4244. The `FILENAME` keyword can be used to change these defaults. If `FILENAME` is specified:

1. If the file is already open (as happens if plotting commands have been directed to the file since the call to `SET_PLOT`), then the file is completed and closed as if `CLOSE_FILE` had been specified.
2. The specified file is opened for subsequent graphics output.

HP-GL Only

Under UNIX, if you wish to send HP-GL output directly to a plotter without generating an intermediate file, you should specify the device special file for the plotter as the argument to `FILENAME`. For example, if your plotter is connected to a serial input/output port known on your system as `/dev/ttya`, you would issue the command:

```
DEVICE, FILENAME='/dev/ttya'
```

All subsequent HP-GL output is sent directly to the plotter connected to serial port `/dev/ttya`.

FLOYD

(PCL, X)

Set this keyword to select the Floyd-Steinberg method of dithering. This algorithm distributes the error, due to displaying intermediate shades in either black or white, to surrounding pixels. This method generally gives the most pleasing results but requires the most computer time.

FONT_INDEX

(PS)

An integer representing the font index to be mapped to the current PostScript font.

Normally the font specification keywords (`AVANTGARDE`, etc.) take effect immediately to change the current font. The `FONT_INDEX` keyword alters this behavior. The current font is not changed. Instead, the specified font is mapped to the

specified font index. This mapping can then be used within text strings to change the font in the middle of the string. See “[Using PostScript Fonts](#)” on page 4303

FONT_SIZE

(PS)

The default height used for displayed text. `FONT_SIZE` is given in points (a common typesetting unit of measure). The default size is 12 point text.

GET_CURRENT_FONT

(METAFILE, PRINTER, WIN, X)

Set this keyword to a named variable in which the name of the current font is returned as a scalar string. A null string is returned if the Windows font is the default font. If the current device is `PRINTER` or `METAFILE`, the current font is returned.

GET_DECOMPOSED

(WIN, X)

Set this keyword to a named variable in which is returned the current state of the decomposed flag in the current direct graphics device.

GET_FONTNAMES

(METAFILE, PRINTER, WIN, X)

Set this keyword to a named variable in which a string array containing the names of available fonts is returned. If no fonts are found, a null scalar string is returned. This keyword must be used in conjunction with the `SET_FONT` keyword. Set the `SET_FONT` keyword to a scalar string containing the name of the desired font or to a wildcard. For example, the following command will return in the variable `fnames` the names of all available fonts:

```
DEVICE, GET_FONTNAMES=fnames, SET_FONT='*'
```

GET_FONTNUM

(METAFILE, PRINTER, WIN, X)

Set this keyword to a named variable in which the number of fonts available to your installation is returned. This keyword must be used in conjunction with the `SET_FONT` keyword. Set the `SET_FONT` keyword to a scalar string containing the name of the desired font or a wildcard. For example, the following command will return in the variable `numfonts` the number of available fonts:

```
DEVICE, GET_FONTNUM=numfonts, SET_FONT='*'
```

GET_GRAPHICS_FUNCTION

(WIN, X, Z)

Set this keyword to a named variable that returns the value of the current graphics function (which is set with the `SET_GRAPHICS_FUNCTION` keyword). This can be used to remember the current graphics function, change it temporarily, and then restore it. See “[SET_GRAPHICS_FUNCTION](#)” on page 4277 keyword for an example.

GET_PAGE_SIZE

(PRINTER)

Set this keyword to a named variable in which to return a two-element vector that contains the width and height of the page size in pixels.

GET_SCREEN_SIZE

(WIN, X)

Set this keyword to a named variable in which to return a two-word array that contains the width and height of the server’s screen, in pixels.

GET_VISUAL_DEPTH

(WIN, X)

Set this keyword to a named variable into which a long integer is returned containing the depth of the visual associated with this device. Under X, if the X server is not connected when you call the `DEVICE` procedure with this keyword set, a new connection is made.

GET_VISUAL_NAME

(WIN, X)

Set this keyword equal to a named variable in which a string containing the name of the current visual class IDL is using is returned. Possible return values are:

- `StaticGray` (X only)
- `GrayScale` (X only)
- `StaticColor` (X only)
- `PseudoColor`

- TrueColor
- DirectColor (X only)

Under X, if no connection to the X server has been established when the DEVICE procedure is called with this keyword set, a new connection is made.

GET_WINDOW_POSITION

(WIN, X)

Set this keyword to a named variable that returns a two-element array containing the (X,Y) position of the lower left corner of the current window on the screen. The origin is also in the lower left corner of the screen.

GET_WRITE_MASK

(WIN, X)

Specifies the name of a variable to receive the current value of the write mask.

GIN_CHARS

(TEK)

The number of characters IDL is to read when accepting a GIN (Graphics INput) report. The default is 5. If your terminal is configured to send a carriage return at the end of each GIN report, set this parameter to 6. If the number of GIN characters is too large, the IDL CURSOR procedure will not respond until two or more keys are struck. If it is too small, the extra characters sent by the terminal will appear as input to the next IDL prompt.

GLYPH_CACHE

(METAFILE, PRINTER, PS, WIN, Z)

Set this keyword to a scalar specifying the maximum number of glyphs to cache at any given time. The first time a glyph from a TrueType font is used, it is tessellated into triangles. These triangles are cached so that the tessellation step is not repeated for each use of that glyph. If the glyph cache fills, the least used glyph will be released before a new glyph is generated and cached. The default is 256.

HELVETICA

(PS)

Set this keyword to select the Helvetica PostScript font.

INCHES

(HP, METAFILE, PCL, PRINTER, PS)

Normally, the XOFFSET, XSIZE, YOFFSET, and YSIZE keywords are specified in centimeters. However, if INCHES is present and non-zero, they are taken to be in inches instead.

INDEX_COLOR

(METAFILE, PRINTER)

Set this keyword to place the printer or MetaFile device in index color mode. This is the default. This keyword is applicable only when the printer or MetaFile device is selected.

ISOLATIN1

(PS)

Set this keyword to use Adobe ISO Latin 1 font encoding with any font that supports such coding. Use of this keyword allows access to many commonly-used foreign characters.

ITALIC

(PS)

Set this keyword to specify that the italic version of the current PostScript font should be used.

LANDSCAPE

(HP, PCL, PRINTER, PS)

IDL normally generates plots with portrait orientation (the abscissa is along the short dimension of the page). If the LANDSCAPE keyword is set, landscape orientation (abscissa along the long dimension of the page) is used instead. Note that explicitly setting LANDSCAPE=0 is the same as setting the [PORTRAIT](#) keyword.

If the current device is PRINTER, and a page is open in the printer, it is closed and a new page set to landscape layout is started.

Note

The ability to set a printer to landscape mode is printer-driver dependent. Your printer may not support this functionality; use the system native print setup dialog to set the orientation of the print job.

LANGUAGE_LEVEL

(PS)

Set this keyword to indicate the language level of the PostScript output that is to be generated by the device. Valid values include 1 (the default) and 2 (required for some features, such as filled patterns for polygons).

LIGHT

(PS)

Set this keyword to specify that the light version of the current PostScript font should be used.

MEDIUM

(PS)

Set this keyword to specify that the medium version of the current PostScript font should be used.

NARROW

(PS)

Set this keyword to specify that the narrow version of the current PostScript font should be used.

NCAR

(CGM)

Set this keyword to set the encoding type for the CGM output file to NCAR binary.

The NCAR Binary Encoding

The NCAR binary encoding is used exclusively by the NCAR graphics package. Version 3.01 of NCAR View (`ctrans`, `ictrans`, and `cgm2ncgm`) does not correctly handle the following graphic elements:

- Cell arrays (raster images) with an odd number of pixels in the X dimension. Solution: specify an even number of pixels for the X dimension or make the image one column wider and fill with zeros.
- Raster images drawn in top down order. Solution: invert the image prior to using TV or TVSCL and do not use the /ORDER keyword. For example:

```
TV, image
; Draw image top to bottom:
TV, ROTATE(image, 7)
```

OBLIQUE

(PS)

Set this keyword to specify that the oblique version of the current PostScript font should be used.

OPTIMIZE

(PCL)

It is desirable, though not always possible, to compress the size of the PCL output file. Such optimization reduces the size of the output file, and improves I/O speed to the printer. There are three levels of optimization:

Value	Description
0	No optimization is performed. This is the default because it will work with any PCL device. However, users of devices which can support optimization should use one of the other optimization levels.
1	Optimization is performed using PCL optimization primitives. This gives the best output compression and printing speed. Unfortunately, not all PCL devices support it. On those that can't, the result will be garbage printed on the page. Consult the programmers manual for your printer to determine if it supports the required escape sequences. The required sequences are: <ESC>*b0M (select full graphics mode), <ESC>*b1M (select compacted graphics mode 1), and <ESC>*b2M (select compacted graphics mode 2). The HP LaserJet II does not support this optimization level. The DeskJet PLUS does.

Table A-7: PCL Optimization Values

Value	Description
2	IDL attempts to optimize the output by explicitly moving the left margin and then outputting non-blank sections of the page. This is primarily intended for use with the LaserJet II, which does not support optimization level 1. Note: This optimization can be very slow on some devices (such as the DeskJet PLUS). On such devices, it is best to avoid this optimization level.

Table A-7: PCL Optimization Values (Continued)

ORDERED

(PCL, X)

Set this keyword to select the ordered dither method. This introduces a pseudo-random error into the display by using a 4 by 4 “dither” matrix, yielding 16 apparent intensities. This is the default method.

OUTPUT

(HP, PS)

Specifies a scalar string that is sent directly to the graphics output file without any processing, allowing the user to send arbitrary commands to the file. Since IDL does not examine the string, it is the user’s responsibility to ensure that the string is correct for the target device.

PALATINO

(PS)

Set this keyword to select the Palatino PostScript font.

PIXELS

(PCL)

Normally, the XOFFSET, XSIZE, YOFFSET, and YSIZE keywords are specified in centimeters. However, if the PIXELS keyword is set, they are taken to be in pixels instead. Note that the selected resolution will determine how large a region is actually written on the page.

PLOT_TO

(REGIS, TEK)

Directs the Tektronix graphic output that would normally go to the user's terminal to the specified I/O unit. The logical unit specified should be open with write access to a device or file. Graphic output may be saved in files for later playback, redirected to other terminals, or to devices that can accept Tektronix graphic commands.

Do not use the interactive graphics cursor when graphic output is not directed to your terminal.

To direct the graphic data to both the terminal and the file, set the unit to the negative of the actual unit number. Alternatively, you can use the TTY keyword, described below.

If the specified unit number is zero then Tektronix output to the file is stopped.

PLOTTER_ON_OFF

(HP)

There are some configurations in which a HP-GL plotter is connected between the computer and a terminal. In this mode (known as eavesdrop mode), the plotter ignores everything it is sent and passes it through to the terminal—the plotter is logically off. This state continues until an escape sequence is sent that turns the plotter logically on. At this point the plotter interprets and executes all input as HP-GL commands. Another escape sequence is sent at the end of the HP-GL commands to return the plotter to the logically off state.

Most configurations do not use eavesdrop mode, and the plotter is always logically on. However, if you are using this style of connection, you must use `PLOTTER_ON_OFF` to instruct IDL to generate the necessary on/off commands. If present and non-zero, `PLOTTER_ON_OFF` causes each output page to be bracketed by device control commands that turn the plotter logically on and off. Specifying a value of zero stops the issuing of such commands. You should only use this keyword before any output has been generated.

POLYFILL

(HP)

Some plotters (e.g., HP7550A) can perform polygon filling in hardware, while others (e.g., HP7475) cannot. IDL therefore assumes that the plotter cannot, and generates all polygon operations in software using line drawing. Specifying a non-zero value for the `POLYFILL` keyword causes IDL to use the hardware polygon filling. Setting it to zero reverts to software filling.

Different implementations of HP-GL plotters may have different limits for the number of vertices that can be specified for a polygon region before the plotter runs

out of internal memory. Since this limit can vary, the HP-GL driver cannot check for calls to POLYFILL that specify too many points. Therefore, it is possible for the user to produce HP-GL output that causes an error when sent to the plotter. To avoid this situation, minimize the number of points used. On the HP7550A, the limit is about 127 points. If you do generate output that exceeds the limit imposed by your plotter, you will have to break that polygon filling operation into multiple smaller operations.

PORTRAIT

(HP, PCL, PRINTER, PS)

Set the PORTRAIT keyword to generate plots using portrait orientation. Portrait orientation is the default. Note that explicitly setting PORTRAIT=0 is the same as setting the [LANDSCAPE](#) keyword.

If the current device is PRINTER, and a page is open in the printer, it is closed and a new page set to portrait layout is started.

Note

The ability to set a printer to portrait mode is printer-driver dependent. Your printer may not support this functionality; use the system native print setup dialog to set the orientation of the print job.

PRE_DEPTH

(PS)

Set this keyword to a value indicating the bit depth to be used for the preview in the PostScript file. Valid values are 1 (for black and white preview) and 8 (for 8-bit grayscale preview). This keyword applies only if the PREVIEW keyword is nonzero. The default depth is 8.

PRE_XSIZE

(PS)

Set this keyword to the width to be used for the preview in the PostScript file. PRE_XSIZE is specified in centimeters, unless the INCHES keyword is set. This keyword applies only if the PREVIEW keyword value is nonzero. The default is 1.77778 inches (128 pixels at 72dpi).

Also see the note below, [“A Note About Preview Dimensions”](#).

PRE_YSIZE

(PS)

Set this keyword to the height to be used for the preview in the PostScript file. `PRE_YSIZE` is specified in centimeters, unless the `INCHES` keyword is set. This keyword applies only if the `PREVIEW` keyword value is nonzero. The default is 1.77778 inches (128 pixels at 72dpi).

Also see the note below, [“A Note About Preview Dimensions”](#).

PREVIEW

(PS)

Set this keyword to 1 to add a platform-independent preview to the PostScript output file in encapsulated PostScript interchange format (EPSI). EPSI is an ASCII format. Set this keyword to 2 to write the EPS file in EPSF format, including an on-screen preview that is supported by many Windows applications, e.g. MSWord. The default (0) is to not include a preview.

Note

EPSF is not an ASCII format and cannot be sent directly to a Postscript printer, unlike the EPSI format. It must be imported into an application for printing.

A Note About Preview Dimensions

Different applications may utilize the information within a PostScript file in different ways when displaying a screen preview. Some applications will ignore the preview contents entirely, and simply use the primary PostScript contents to generate a screen preview. Other applications will use the preview data and its corresponding dimensions for screen display. Still others will use the preview data and stretch it to the dimensions of the primary PostScript contents. It is therefore recommended that the target application (into which the encapsulated PostScript file is to be loaded) be considered when selecting an appropriate `XSIZE`, `YSIZE`, `PRE_XSIZE`, and `PRE_YSIZE`.

PRINT_FILE

(WIN)

Set this keyword to the name of a file (e.g., PostScript or PCL) to be sent to the currently-selected Windows printer. IDL performs no type checking on this file before sending it to the printer. Therefore, if you have a PostScript printer selected and you send a file that contains no valid PostScript information, you'll simply get text output.

To send the file `myfile.ps` to the currently-selected Windows printer, enter:

```
DEVICE, PRINT_FILE='myfile.ps'
```

PSEUDO_COLOR

(X)

If this keyword is present, the PseudoColor visual is used. The value of the keyword represents the number of bits per pixel to be used. This keyword has effect only if no windows have been created. Visual classes are discussed in more detail in [“X Windows Visuals”](#) on page 4319.

RESET_STRING

(TEK)

The string used to place the terminal back into the normal interactive mode after drawing graphics. Use this parameter, in conjunction with the SET_STRING keyword, to control the mode switching of your terminal.

For example, the GraphON 200 series terminals require the string <ESC>2 to activate the alphanumeric window after drawing graphics. The call to set this is:

```
DEVICE, RESET = string(27b) + '2'
```

If the 4100 series mode switch is set, using the keyword TEK4100, the default mode resetting string is <ESC>%!1, which selects the ANSI code mode.

RESOLUTION

(PCL)

PCL Only

The resolution at which the PCL printer will work. PCL supports resolutions of 75, 100, 150, and 300 dots per inch. The default is 300 dpi. Lower resolution gives smaller output files, while higher resolution gives superior quality.

RETAIN

(WIN, X)

Use this keyword to specify the default method used for backing store when creating new windows. This is the method used when the RETAIN keyword is not specified with the WINDOW procedure. Backing store is discussed in more detail under [“Backing Store”](#) on page 4286, along with the possible values for this keyword. If RETAIN is not used to specify the default method, method 1 (server-supplied backing store) is used.

Microsoft Windows Only

The initial value of this parameter can be set by selecting File-Preferences from the menu bar. See [“Backing Store”](#) on page 4286.

A Note on Reading Data from Windows

On some systems, when backing store is provided by the window system (RETAIN=1), reading data from a window using TVRD may cause unexpected results. For example, data may be improperly read from the window even when the image displayed on screen is correct. Having IDL provide the backing store (RETAIN=2) ensures that the window contents will be read properly. These types of problems are described in more detail in the documentation for TVRD. See [“Unexpected Results Using TVRD with X Windows”](#) on page 2176.

SCALE_FACTOR

(PRINTER, PS)

Specifies a scale factor applied to the entire plot. The default value is 1.0, allowing output to appear at its normal size. SCALE_FACTOR is used to magnify or shrink the resulting output.

The SCALE_FACTOR keyword behaves slightly differently in the context of the PRINTER device than it does in the context of the PS device.

When the current device is PRINTER, the SCALE_FACTOR keyword is designed to emulate a scalable resolution setting on the printer. For example, if you have a 300 x 300 pixel image—stored in the variable *image*—the following IDL commands will print *image* in a 0.5 inch square on a 600 dpi printer:

```
SET_PLOT, 'printer'
TV, image
```

Setting SCALE_FACTOR to 2 will scale the image to a 1 inch square on the same 600 dpi printer:

```
SET_PLOT, 'printer'
DEVICE, SCALE_FACTOR=2
TV, image
```

The output of IDL’s Direct Graphics routines (CONTOUR, PLOT, SURFACE, etc.) is automatically scaled to fill the available drawing area. As a result, the following IDL commands will produce two identical copies of the same output on any printer:

```
SET_PLOT, 'printer'
PLOT, data
DEVICE, SCALE_FACTOR=2
PLOT, data
```

SCHOOLBOOK

(PS)

Set this keyword to select the New Century Schoolbook PostScript font.

SET_CHARACTER_SIZE

(CGM, HP, METAFILE, PCL, PRINTER, PS, REGIS, TEK, WIN, X, Z)

Set this keyword equal to a two-element vector to specify the font size and line spacing (leading) of vector and TrueType fonts, and the line spacing of device fonts. The way that the value of this vector determines character size is not completely intuitive.

The vector specified to the SET_CHARACTER_SIZE keyword sets the values of the X_CH_SIZE and Y_CH_SIZE fields in the [!D System Variable](#) structure. These values describe the size of the rectangle that contains the “average” character in the current font. (It is not important what the “average” character is; it is used only to calculate a scaling factor that will be applied to all of the characters in the font.) The first element specifies the width of the rectangle in device units (usually pixels), and the second element specifies the height.

For vector and TrueType fonts, the height of the “average” character is determined by the *width* of the rectangle. The aspect ratio of the “average” character remains fixed; the character is scaled so that its width fits in the specified rectangle. The resulting scale factor is then applied to all of the characters in the font. The amount of spacing between lines (baseline to baseline) is determined explicitly by the height of the rectangle.

For device fonts, the character size is fixed. When the device font system is in use, the first element of the vector specified to SET_CHARACTER_SIZE is silently ignored, and only the line-spacing value is used.

Note

Changing between font systems (and sometimes changing from one font to another within the same font system) can also change the !D structure, so do not assume that the character size you have set is preserved when you change fonts.

SET_COLORMAP

(PCL)

Set this keyword to a 14,739 ($= 3 \cdot 17^3$) element byte vector containing the RGB-to-printer color translation table for a color PCL printer. The default table is for an HP Deskjet 500C printer.

The translation table is divided into red, green, and blue planes of 4913 ($=17^3$) elements each. For a given RGB triple, the offset into each plane is calculated as follows:

$$\text{Offset} = (\text{Red}/16) * 289 + (\text{Green}/16) * 17 + (\text{Blue}/16)$$

Thus, if the RGB triple is [16,32,160], the offset into each plane is 333. The printer will use the value at element 332 of the translation table as the red value, the value at element 5245 ($=4913+332$) as the green value, and the value at element 10158 ($=9826+332$) as the blue value.

The following example shows how to scale an existing colortable for use by a PCL printer.

```

; Set the plot window to the X device:
SET_PLOT, 'X'
; Create a window:
WINDOW, 0, XS=300, YS=300
; Load a color table:
LOADCT, 13
; Read color table values into variables:
TVLCT, r, g, b, /GET
; Re-size color table variables:
r2=CONGRID(r, 4913)
g2=CONGRID(g, 4913)
b2=CONGRID(b, 4913)
; Create 14,739-element color map:
colormap=[r2, g2, b2]
; Change to the PCL device:
SET_PLOT, 'PCL'
; Set file name, resolution, color, and color map:
DEVICE, FILE = 'pcl.pcl', RESOLUTION = 300, $
    /COLOR, SET_COLORMAP = colormap
; Display an image:
TVSCL, DIST(900)
; Close the device:
DEVICE, /CLOSE

```

Note

The color table used need not be one of IDL's predefined tables.

SET_COLORS

(Z)

Sets the number of pixel values, !D.N_COLORS and !D.TABLE_SIZE. This value is used by a number of IDL routines to determine the scaling of pixel data and the default drawing index. Allowable values range from 2 to 256, and the default value is 256. Use this parameter to make the Z-buffer device compatible with devices with fewer than 256 colors indices.

SET_FONT

(METAFILE, PRINTER, PS, WIN, X, Z)

Set this keyword to a scalar string specifying the name of the font used when a hardware or TrueType font is selected. Note that hardware fonts cannot be rotated, scaled, or projected, and that the “!” commands for formatting may not work. When generating three-dimensional plots, it is best to use the vector-drawn or TrueType characters. Note that for the PS device, only one hardware font (other than the predefined fonts set via the fontname keywords, such as /AVANTEGARDE) may be loaded at a time.

Note

On some platforms, GET_FONT returns an empty string. When SET_FONT is passed an empty string, the first font that can be successfully loaded will be used. This is the first available font found in the user's resource file (\$HOME/.xdefaults) or the app-defaults file (\$IDL_DIR/resource/X11/lib/app-defaults/Idl).

Note on the FONT Keyword

The SET_FONT keyword was introduced with IDL version 5.1 and replaces the FONT and USER_FONT keywords used in previous versions.

Using TrueType Fonts

For TrueType fonts, the specified font name must exactly match one of the names in the first column of the `ttfont.map` file in the `resource/fonts/tt` directory or (on Windows platforms) the name of an installed font. See [“About TrueType Fonts”](#) on page 4419 for details on the `ttfont.map` file and for a listing of TrueType fonts distributed with IDL. Note that you must include the TT_FONT keyword to indicate that the font specified is a TrueType font. For example, the following sets the font to the font to the TrueType font Helvetica Bold Italic:

```
DEVICE, SET_FONT='Helvetica-BoldItalic'
```

Note

You can append additional TrueType fonts to the `ttfont.map` file if desired; on Windows platforms, additional fonts can also be added via the normal font installation procedures for your system. RSI cannot guarantee that TrueType fonts you add will be satisfactorily tessellated or displayed. See [“About TrueType Fonts”](#) on page 4419 for details.

Using Hardware Fonts

Because device fonts are specified differently on different platforms, the syntax of the *fontname* string depends on which platform you are using.

UNIX

Usually, the window system provides a directory of font files that can be used by all applications. List the contents of that directory to find the fonts available on your system. The size of the font selected also affects the size of vector drawn text. X Windows users can use the `xlsfonts` command to list available X Windows fonts.

On some machines, fonts are kept in subdirectories of `/usr/lib/X11/fonts`.

For example, to select the font 8X13:

```
!P.FONT = 0
DEVICE, SET_FONT = '8X13'
```

Microsoft Windows

The `SET_FONT` keyword should be set to a string with the following form:

```
DEVICE, SET_FONT='font*modifier1*modifier2*...modifiern'
```

where the asterisk (*) acts as a delimiter between the font's name (*font*) and any modifiers. The string is *not* case sensitive. Modifiers are simply “keywords” that change aspects of the selected font. Valid modifiers are:

- For font weight: THIN, LIGHT, BOLD, HEAVY
- For font quality: DRAFT, PROOF
- For font pitch: FIXED, VARIABLE
- For font angle: ITALIC
- For strikeout text: STRIKEOUT
- For underlined text: UNDERLINE
- For font size: Any number is interpreted as the font height in pixels.

For example, if you have Garamond installed as one of your Windows fonts, you could select 24-pixel cell height Garamond italic as the font to use in plotting. The following commands tell IDL to use hardware fonts, change the font, and then make a simple plot:

```
!P.FONT = 0
DEVICE, SET_FONT = 'GARAMOND*ITALIC*24'
PLOT, FINDGEN(10), TITLE = 'IDL Plot'
```

This feature is compatible with TrueType and Adobe Type Manager (and, possibly, other type scaling programs for Windows). If you have TrueType or ATM installed, the TrueType or PostScript outline fonts are used so that text looks good at any size.

SET_GRAPHICS_FUNCTION

(WIN, X, Z)

Most window systems allow applications to specify the graphics function. This is a logical function which specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. The complete list of possible values is given in the following table:

Logical Function	Code	Definition
GXclear	0	0
GXand	1	source AND destination
GXandReverse	2	source AND (NOT destination)
GXcopy	3	source
GXandInverted	4	(NOT source) AND destination
GXnoop	5	destination
GXxor	6	source XOR destination
GXor	7	source OR destination
GXnor	8	(NOT source) AND (NOT destination)
GXequiv	9	(NOT source) XOR destination
GXinvert	10	(NOT destination)

Table A-8: Graphic Function Codes

Logical Function	Code	Definition
GXorReverse	11	source OR (NOT destination)
GXcopyInverted	12	(NOT source)
GXorInverted	13	(NOT source) OR destination
GXnand	14	(NOT source) OR (NOT destination)
GXset	15	1

Table A-8: Graphic Function Codes (Continued)

The default graphics function is GXcopy, which causes new pixels to completely overwrite any previous pixels. Not all functions are available on all window systems.

For example, the following code segment inverts the bottom bit in the rectangle defined by its diagonal corners (x_0, y_0) and (x_1, y_1):

```

; Set graphics function to exclusive or (GXor), and save the
; old function:
DEVICE, GET_GRAPHICS_FUNCTION = oldg, SET_GRAPHICS_FUNCTION = 6
; Use POLYFILL to select the area to be inverted. The source
; pixel value is 1:
POLYFILL, [[x0,y0], [x0,y1], [x1,y1], [x1,y0]], $
    /DEVICE, COLOR=1
; Restore the previous graphics function:
DEVICE, SET_GRAPHICS_FUNCTION=oldg

```

SET_RESOLUTION

(Z)

Set this keyword to a two-element vector that specifies the width and height of the Z-buffers. The default size is 640 by 480. If this size is not the same as the existing buffers, the current buffers are destroyed and the device is reinitialized.

SET_STRING

(TEK)

The string used to place the terminal into the graphics mode from the normal interactive terminal mode. If the 4100 series mode switch is set, using the keyword TEK4100, the default graphic mode setting string is `<ESC>%!0`, which selects the Tektronix code mode.

SET_TRANSLATION

(X)

This keyword can be used to allow multiple, simultaneous IDL sessions to use the same colors from a shared colormap. Use this keyword before the X connection is established (i.e., before a window is created), IDL will use the shared color map without allocating any additional colors, and will not load a grayscale ramp as is usually done when the X server starts up. The following example shows two cooperating IDL processes sharing the same colormap:

Execute the following commands in the first IDL session:

```
WINDOW, GET_X_ID = a
DEVICE, TRANSLATION = t
OPENW, 1, 'junk.dat'
WRITEU, 1, a, !D.N_COLORS, t[0:!D.N_COLORS-1]
CLOSE, 1
LOADCT, 3
```

Execute the following commands in the second IDL session:

```
OPENR, 1, 'junk.dat'
a=0L
n=0L
READU, 1, a, n
t = BYTARR(n)
READU, 1, t
CLOSE, 1
DEVICE, SET_TRANSLATION = t
WINDOW, COLORS=n, SET_X_ID=a
TV, DIST(256)
```

SET_WRITE_MASK

(X, Z)

Sets the write mask to the specified value. For an n -bit system, the write mask can range from 0 to 2^n-1 .

STATIC_COLOR

(X)

Use this keyword to select the X Windows StaticColor visual. The value of the keyword represents the number of bits per pixel to be used. This keyword has effect only if no windows have been created. Visual classes are discussed in more detail in [“X Windows Visuals”](#) on page 4319.

STATIC_GRAY

(X)

Use this keyword to select the X Windows StaticGray visual. The value of the keyword represents the number of bits per pixel to be used. This keyword has effect only if no windows have been created. Visual classes are discussed in more detail in “X Windows Visuals” on page 4319.

SYMBOL

(PS)

Set this keyword to select the Symbol PostScript font.

TEK4014

(TEK)

Set this keyword to specify that coordinates are to be output with full 12-bit resolution. If this keyword is not present or is zero, 10-bit coordinates are output. Normally, IDL sends 10-bit coordinates. 12-bit coordinates are compatible with most terminals, even those without the full resolution, but require more characters to send.

Note

The 4014 and the 4100 modes can be used together. The coordinate system IDL uses for the Tektronix is 0 to 4095 in the X direction and 0 to 3120 in the Y direction, even when not in the 4014 mode. In the 10-bit case the internal coordinates are divided by 4 prior to output.

TEK4100

(TEK)

Set this keyword to indicate that the terminal is a 4100 or 4200 series terminal. The use of color, ANSI and Tektronix mode switching, hardware line styles, and pixel output with the TV procedure is supported with these terminals. Also, text is output differently.

TEXT

(CGM)

Set this keyword to set the encoding type for the CGM output file to text.

THRESHOLD

(PCL, X)

Set this keyword to select the threshold algorithm—the simplest dithering method. The value of this keyword is the threshold to be used. This algorithm simply compares each pixel against the given threshold, usually 128. If the pixel equals or exceeds the threshold the display pixel is set to white, otherwise it is black.

TIMES

(PS)

Set this keyword to select the Times-Roman PostScript font.

TRANSLATION

(WIN, X)

As discussed in [“Shared Colormaps”](#) on page 4322, using the shared colormap (normally recommended) causes IDL to translate between IDL color indices (which always start with zero and are contiguous) and the pixel values actually present in the display. The TRANSLATION keyword specifies the name of a variable to receive the translation vector. To read the translation table, use the command:

```
DEVICE, TRANSLATION=TRANSARR
```

where TRANSARR is a named variable into which the translation array is stored. The result is a 256-element byte vector. Element zero of the vector contains the pixel value allocated for the first color in the IDL colormap, and so forth.

Microsoft Windows Only

This keyword is accepted by the WIN device, for compatibility with the X Windows driver, but simply returns a 256-element vector where each element has the value of its subscript (0 to 255).

TRUE_COLOR

(METAFILE, PRINTER, X)

Use this keyword to select TrueColor visuals. The value of the keyword represents the number of bits per pixel to be used. This keyword has effect only if no windows have been created. Visual classes are discussed in more detail in [“X Windows Visuals”](#) on page 4319. If the current device is PRINTER or METAFILE, the printer is placed in RGB or TrueColor mode if the value of the TRUE_COLOR keyword is greater than zero (the number of bits per pixel specified is ignored.)

TT_FONT

(METAFILE, PRINTER, PS, WIN, X, Z)

Set this keyword to indicate that the font set via the [SET_FONT](#) keyword (either to set the fontname or to retrieve fontnames in conjunction with the [GET_FONTNAMES](#) or [GET_FONTNUM](#) keywords) should be treated as a TrueType font.

TTY

(REGIS, TEK)

Set this keyword to specify that output should be sent to the terminal at the same time that it is being sent to a file due to the [FILENAME](#) or [PLOT_TO](#) keywords. A zero value causes output to go only to the file. If no output file is in use, this keyword has no effect.

USER_FONT

(PS)

This keyword is now obsolete and has been replaced by the [SET_FONT](#) keyword. Code that uses the [USER_FONT](#) keyword will continue to function as before, but we suggest that all new code use [SET_FONT](#).

VT240, VT241

(REGIS)

Set this keyword to configure the REGIS device for VT240 series terminals.

VT340, VT341

(REGIS)

Set this keyword to configure the REGIS device for VT340 series terminals.

WINDOW_STATE

(WIN, X)

Set this keyword to a named variable that returns an array containing one element for each possible window. Array element *i* contains a 1 if window *i* is open, otherwise it contains a 0.

XOFFSET

(HP, PCL, PRINTER, PS)

Specifies the X position, on the page, of the lower left corner of output generated by IDL. XOFFSET is specified in centimeters, unless INCHES is specified. See [“Positioning Graphics Output”](#) on page 4291.

PostScript Only

SCALE does not affect the value of XOFFSET.

XON_XOFF

(HP)

If present and non-zero, XON_XOFF causes each output page to start with device control commands that instruct the plotter to obey xon/xoff (^S/^Q) style flow control. Specifying a value of zero stops the issuing of such commands. You should only use this keyword before any output has been generated.

Such handshaking is the default. To turn it off, use the command

```
DEVICE, XON_XOFF=0
```

Often, it is not necessary to tell the plotter to obey flow control because the printing facilities on the system handle such details for you, but it is usually harmless.

XSIZE

(HP, METAFILE, PCL, PRINTER, PS)

Specifies the width of output generated by IDL. XSIZE is specified in centimeters, unless INCHES is specified.

PostScript Only

SCALE modifies the value of XSIZE. Hence, the following statement:

```
DEVICE, /INCHES, XSIZE=7.0, SCALE_FACTOR=0.5
```

results in a real width of 3.5 inches.

Also see [“A Note About Preview Dimensions”](#) on page 4270.

YOFFSET

(HP, PCL, PRINTER, PS)

Specifies the Y position, on the page, of the lower left corner of output generated by IDL. YOFFSET is specified in centimeters, unless INCHES is specified. See [“Positioning Graphics Output”](#) on page 4291.

Note

The corner of the page from which the Y offset is measured (lower or upper left) differs on various devices. Read the device specific information in the following sections to determine how this is handled for your device.

PostScript Only

SCALE does not affect the value of YOFFSET.

YSIZE

(HP, METAFILE, PCL, PRINTER, PS)

Specifies the height of output generated by IDL. YSIZE is specified in centimeters, unless INCHES is specified.

PostScript Only

SCALE modifies the value of YSIZE. Hence, the following statement:

```
DEVICE, /INCHES, YSIZE=5.0, SCALE_FACTOR=0.5
```

results in a real width of 2.5 inches.

Also see [“A Note About Preview Dimensions”](#) on page 4270.

ZAPFCHANCERY

(PS)

Set this keyword to select the ITC Zapf Chancery PostScript font.

ZAPFDINGBATS

(PS)

Set this keyword to select the ITC Zapf Dingbats PostScript font.

Z_BUFFERING

(Z)

This keyword enables and disables the Z-buffering. If this keyword is specified with a zero value, the driver operates as a standard 2-D device, the Z-buffering is disabled,

and the Z-buffer (if any) is deallocated. Setting this keyword to one (the default value), enables the Z-buffering.

To disable Z-buffering enter:

```
DEVICE, Z_BUFFERING = 0
```

Window Systems

The different window systems supported by IDL have many features in common. This section describes those features. See the individual descriptions of each system later in this chapter for additional information about each one.

IDL utilizes the window system by creating and using one or more largely independent windows, each of which can be used for the display of graphics and/or images. One color map table is shared among all these windows. Multiple windows can be active simultaneously. Windows are referenced using their index which is a non-negative integer.

“Dithering” or halftoning techniques are used to display images with multiple shades of gray on monochrome displays—displays that can only display white or black. This topic is discussed in [“Image Display On Monochrome Devices”](#) on page 4288.

Graphic and image output is always directed to the current window. When a window system is selected as the current IDL graphics device, the index number of the current window is found in the `!D.WINDOW` system variable. This variable contains -1 if no window is open or selected. The `WSET` procedure is used to change the current window. `WSHOW` hides, displays, and iconifies windows. `WDELETE` deletes a window.

The `WINDOW` procedure creates a new window with a given index. If a window already exists with the same index, it is first deleted. The size, position, title, and number of colors, may also be specified. If you access the display before creating the first window, IDL automatically creates a window with an index number of 0 and with the default attributes.

Backing Store

One of the features that distinguishes various window systems is how they handle the issue of backing store. When part of a window that was previously not visible is exposed, there are two basic approaches that a window system can take. Some keep track of the current contents of all windows and automatically repair any damage to their visible regions (retained windows). This saved information is known as the backing store. Others simply report the damage to the program that created the

window and leave repairing the visible region to the program (non-retained windows).

Value	Description
0	No backing store.
1	Request the server or window system to perform backing store.
2	Make IDL perform backing store.

Table A-9: Allowed Values for the RETAIN Keyword

There are convincing arguments for and against both approaches. It is generally more convenient for IDL if the window system handles this problem automatically, but this often comes at a performance penalty. The actual cost of retained windows varies between systems and depends partially on the application.

The X Window system does not by default keep track of window contents. Therefore, when a window on the display is obscured by another window, the contents of its obscured portion is lost. Re-exposing the window causes the X server to fill the missing data with the default background color for that window, and request the application to redraw the missing data. Applications can request a backing store for their windows, but servers are not required to provide it. Many X servers do not provide backing store, and even those that do cannot necessarily provide it for all requesting windows. Therefore, requesting backing store from the server might help, but there is no certainty.

The IDL window system drivers allow you to control the issue of backing store using the RETAIN keyword to the DEVICE and WINDOW procedures. Using it with DEVICE allows you to set the default action for all windows, while using it with WINDOW lets you override the default for the new window. The possible values for this keyword are summarized under “[Backing Store](#)” on page 4286, and are described below:

- Setting the RETAIN keyword to 0 specifies that no backing store is kept. In this case, exposing a previously obscured window leaves the missing portion of the window blank. Although this behavior can be inconvenient, it usually has the highest performance because there is no need to keep a copy of the window contents.
- Setting the RETAIN keyword to 1 causes IDL to request that a backing store be maintained. If the window system decides to accept the request, it will automatically repair the missing portions when the window is exposed. X

Windows may or may not provide backing store when requested, depending on the capabilities of the server and the resources available to it.

- Setting the RETAIN keyword to 2 specifies that IDL should keep a backing store for the window itself, and repair any window damage when the window system requests it. This option exists for X Windows. In this case, a pixmap (off-screen display memory) the same size as the window is created at the same time the window is created, and all graphics operations sent to the window are also sent to the pixmap. When the server requests IDL to repair freshly exposed windows, this pixmap is used to fill in the missing contents. Pixmap is a precious resource in the X server, so backing pixmaps should only be requested for windows with contents that must absolutely be preserved.

If the type of backing store to use is not explicitly specified using the RETAIN keyword, IDL assumes option 1 and requests the window system to keep a backing store.

A Note on Reading Data from Windows

On some systems, when backing store is provided by the window system (RETAIN=1), reading data from a window using TVRD may cause unexpected results. For example, data may be improperly read from the window even when the image displayed on screen is correct. Having IDL provide the backing store (RETAIN=2) ensures that the window contents will be read properly. These types of problems are described in more detail in the documentation for TVRD. See [“Unexpected Results Using TVRD with X Windows”](#) on page 2176.

Image Display On Monochrome Devices

Images are automatically dithered when sent to some monochrome devices. Dithering is a technique which increases the number of apparent brightness levels at the expense of spatial resolution. Images with 256 gray levels are displayed on a display with only two colors, black and white, using halftoning techniques. PostScript handles dithering directly. IDL supports dithering for other devices if their DEVICE procedures accept the FLOYD, ORDERED, or THRESHOLD keywords.

Printing Graphics Output Files

For Direct Graphics printer and plotter devices (e.g., PCL, PostScript, and HP-GL), IDL creates a file containing output commands. This file can be sent to the printer via the normal methods provided by the local operating system. When attempting to output the file before exiting IDL, the user must be sure that the graphics output file is complete. For example, the following IDL commands (executed under UNIX) will not produce the desired result:

```
SET_PLOT, 'PS'
PLOT, x, y
SPAWN, 'lpr idl.ps'
```

These commands fail because the attempt to print the file is premature—the file is still open within IDL and is not yet complete.

Note

Use of SET_PLOT and the direct graphics commands is not required when creating graphical output files or printing to a printer using the iTools or other object-graphics routines.

The following lines of code are an IDL procedure called OUTPUT_PLOT which closes the current graphics file and sends it to the printer. This routine assumes that the graphics output file is named `idl.xxx`, where `xxx` represents the name of the graphics driver. For example, PostScript output file is assumed to be `idl.ps`. It also assumes that the graphics output to be printed is from the current graphics device, as selected with SET_PLOT.

```
; Close the current graphics file, and print it. If the
; New_file parameter is present, rename the file to the given
; name so it won't be overwritten:
Pro OUTPUT_PLOT, New_file
; Close current graphics file:
DEVICE, /CLOSE
; Build the default output file name by using the idl name for
; the current device (!D.NAME):
file = 'idl.' + STRLOWCASE(!D.NAME)
; Build shell commands to send file to the printer.
; You will probably have to change this command in accordance
; with local usage:
cmd = 'lpr ' + file
; Concatenate rename command if new file specified:
IF N_ELEMENTS(New_file) GT 0 THEN $
    cmd = cmd + '; mv' + file + ' ' + New_file
; Issue shell commands to print/rename file:
```

```
SPAWN, cmd
END
```

The call to `DEVICE` causes IDL to finish the file and close it, which makes it available for printing.

Setting Up The Printer

In order for IDL generated output files to work properly with printers and plotters, it is necessary for the device to be configured properly. This usually involves configuring both the device hardware and the operating system printing software. When setting up your system, keep the following points in mind:

- The device and computer must use some form of flow control to prevent the computer from sending data faster than the printing device can handle it. The most common form of flow control is known as XON/XOFF, and involves the sending of Control-S (off) and Control-Q (on) characters from the device to the printer to manage the flow of data.

Many printers have a large buffer into which they store incoming data they haven't yet processed. This reduces the need to invoke flow control. When testing your configuration to ensure flow control is actually enabled, you must be sure to print a document long enough to fill any such buffer, or flow control may never occur, giving a false impression that the setup is correct. A common source of problems stem from attempting to print long IDL generated output files without proper flow control.

- Some devices (such as PCL) require an eight-bit data path, while others (such as PostScript) do not. For devices that do, it is important to ensure that the printer port and system printing software provide such a connection.

If you are having problems printing on a PostScript printer, the `ehandler.ps` file in the `resource/fonts/ps` subdirectory of the IDL distribution can help you to debug your problem. Sending this file to your PostScript Printer causes it to print any subsequent errors it encounters on a sheet of paper and eject it. The effect of this file lasts until the printer is reset.

Setting Up Printers Under UNIX

Printers are configured in the `/etc/printcap` file. This file describes to the system which printers are connected to it, the characteristics of each printer, and how the printer port should be configured. Managing the `printcap` file is usually discussed in the system management documentation supplied with the system by the manufacturer.

Positioning Graphics Output

The difference between the XOFFSET and YOFFSET keywords to the DEVICE procedure, and the higher level plot positioning keywords and system variables (discussed in [Appendix B, “Graphics Keywords”](#) and [Chapter 17, “Direct Graphics Plotting”](#) in the *Using IDL* manual) can lead to confusion. A common misunderstanding is to attempt to use the DEVICE procedure “offset” and “size” keywords multiple times in an attempt to produce multiple plots on a single output page.

The DEVICE keywords are intended to specify the size and position of the entire output area on the page, not to move the plotting region for multiple plots. The driver does not monitor their values continuously, but only when initializing a new page or ejecting the current one.

The proper way to produce multiple plots is to use the high level positioning abilities. The !P.MULTI, !P.POSITION, and !P.REGION system variables can be used to position individual plots on the page. The plotting routines also accept the POSITION, MARGIN and REGION keywords.

Image Background Color

Graphical output that is displayed with a black background on a monitor frequently look better if the background is changed to white when printed on white paper. This is easily done using the **BACKGROUND** graphics keyword or (if the only values displayed in black are in the background) with a statement like:

```
a(WHERE(a EQ 0B)) = 255B
```

The CGM Device

Device Keywords Accepted by the CGM Device:

[BINARY](#), [CLOSE_FILE](#), [COLORS](#), [ENCODING](#), [FILENAME](#), [NCAR](#), [SET_CHARACTER_SIZE](#), [TEXT](#)

The CGM, Computer Graphics Metafile, standard describes a device independent file format used for the exchange of graphic information. The IDL CGM driver produces CGM files encoded in one of three methods: *Text*, *Binary* or *NCAR Binary*. To direct graphics output to a CGM file, issue the command:

```
SET_PLOT, 'CGM'
```

This causes IDL to use the CGM driver for producing graphical output. Once the CGM driver is selected, the `DEVICE` procedure controls its actions, as described below. Typing `HELP, /DEVICE` displays the current state of the CGM driver. The CGM driver defaults to the binary encoding using 256 colors.

Abilities and Limitations

This section describes details specific to IDL's CGM implementation:

- IDL uses the CGM default integer encoding for graphic primitives. Coordinate values range from 0 to 32767. It is advisable to use the values stored in `!D.X_SIZE` and `!D.Y_SIZE` instead of assuming a fixed coordinate range.
- Color information is output with a resolution of 8 bits (color indices and intensity values range from 0 to 255).
- The definition of background color in the CGM standard is somewhat ambiguous. According to the standard, color index 0 and the background color are the same. Because background color is specified in the metafile as a color value (RGB triple), not an index, it is possible to have the background color not correspond with the color value of index 0.
- The CGM `BACKGROUND_COLOUR` attribute is explicitly set by IDL only during an erase operation: changing the value of the color map at index 0 does not cause IDL to generate a `BACKGROUND_COLOUR` attribute until the next `ERASE` occurs. An `ERASE` command sets the background color to the value in the color map at index 0. The command `ERASE, INDEX` (where `INDEX` is not 0) generates the message "value of background color is out of allowed range." For consistent results, modify the color table before any graphics are output.

- The CGM standard uses scalable (variable size) pixels for raster images. By default, the TV and TVSCL procedures output images, regardless of size, using the entire graphics output area. To output an image smaller than the graphics output area, specify the XSIZE and YSIZE keywords with the TV and TVSCL procedures. For example:

```
; Select the CGM driver:
SET_PLOT, 'CGM'
; Create a 64 x 64 element array:
X = DIST(64)
; Display the image (fills entire screen):
TVSCL, X
; Now display 4 images on the screen:
ERASE
XS = !D.X_SIZE / 2 ; Size of each image, X dimension
YS = !D.Y_SIZE / 2 ; Size of each image, Y dimension
TVSCL, X, 0, XSIZE=XS, YSIZE=YS ; Upper left
TVSCL, X, 1, XSIZE=XS, YSIZE=YS ; Upper right
TVSCL, X, 2, XSIZE=XS, YSIZE=YS ; Lower left
TVSCL, X, 3, XSIZE=XS, YSIZE=YS ; Lower right
```

The HP-GL Device

Device Keywords Accepted by the HP-GL Device:

`CLOSE_FILE`, `EJECT`, `FILENAME`, `INCHES`, `LANDSCAPE`, `OUTPUT`, `PLOTTER_ON_OFF`, `POLYFILL`, `PORTRAIT`, `SET_CHARACTER_SIZE`, `XOFFSET`, `XON_XOFF`, `XSIZE`, `YOFFSET`, `YSIZE`

HP-GL (Hewlett-Packard Graphics Language) is a plotter control language used to produce graphics on a wide family of pen plotters. To use HP-GL as the current graphics device, issue the IDL command:

```
SET_PLOT, 'HP'
```

This causes IDL to use HP-GL for producing graphical output. Once the HP-GL driver is enabled via `SET_PLOT`, the `DEVICE` procedure is used to control its actions, as described below. The default settings for the HP-GL driver are shown in the following table. Use the statement:

```
HELP, /DEVICE
```

to view the current state of the HP-GL driver.

Feature	Value
File	idl.hp
Orientation	Portrait
Erase	No action
Polygon filling	Software
Turn plotter logically on/off	No
Specify xon/xoff flow control	Yes
Horizontal offset	3/4 in.
Vertical offset	5 in.
Width	7 in.
Height	5 in.

Table A-10: Default HP-GL Driver Settings

Abilities And Limitations

IDL is able to produce a wide variety of graphical output using HP-GL. The following is a list of what is and is not supported:

- All types of vector graphics can be generated, including line plots, contours, surfaces, etc.
- HP-GL plotters can draw lines in different colors selected from the pen carousel. It should be noted that color tables are not used with HP-GL. Instead, each color index refers directly to one of the pens in the carousel.
- Some HP-GL plotters can do polygon filling in hardware. Others can rely on the software polygon filling provided by IDL.
- It is possible to generate graphics using the hardware generated text characters, although such characters do not give much improvement over the standard vector fonts. To use hardware characters, set the !P.FONT system variable to zero, or set the FONT keyword to the plotting routines to zero.
- Since HP-GL is designed to drive pen plotters, it does not support the output of raster images. Therefore, TV and TVSCL do not work with HP-GL.
- Since pen plotters are not interactive devices, they cannot support such operations as cursors and windows.

HP-GL Linestyles

The LINSTYLE graphics keyword allows specifying any of 6 linestyles. HP-GL does not support all of these linestyles, and styles 3 and 4 differ from the definition in [Appendix B, “Graphics Keywords”](#). The following table summarizes the differences:

Index	Normal Line Style	HP-GL Line Style
0	Solid	same
1	Dotted	same
2	Dashed	same
3	Dash Dot	Relative size of dash and dot are different.
4	Dash Dot Dot Dot	Dash Dot Dot
5	Long Dashes	same

Table A-11: Linestyles for the HP-GL Device

The Metafile Display Device

Device Keywords Accepted by the Metafile Device:

`CLOSE_FILE`, `FILENAME`, `GET_CURRENT_FONT`, `GET_FONTNAMES`, `GET_FONTNUM`, `GLYPH_CACHE`, `INCHES`, `INDEX_COLOR`, `SET_CHARACTER_SIZE`, `SET_FONT`, `TRUE_COLOR`, `TT_FONT`, `XSIZE`, `YSIZE`

The Windows Metafile Format (WMF) is used by Windows to store vector graphics in order to exchange graphics information between applications. This format is only available on the Windows platforms. To direct graphics to a file in the WMF format, use the `SET_PLOT` procedure:

```
SET_PLOT, 'METAFILE'
```

This causes IDL to use the Metafile driver for producing graphical output. Once the Metafile driver is enabled via `SET_PLOT`, the `DEVICE` procedure is used to control its actions. The default settings are given in the following table:

Feature	Value
File	idl.emf
Mode	N/A
Horizontal offset	N/A
Vertical offset	N/A
Width	7 in.
Height	5 in.
Resolution	Screen

Table A-12: Default Metafile Driver Settings

For example, the following will create a WMF file for a simple plot:

```
;Create X and Y Axis data
x=findgen(10)
y=findgen(10)

;Save current device name
mydevice=!D.NAME
```

```
;Set the device to Metafile
SET_PLOT, 'METAFILE'

;Name the file to be created
DEVICE, FILE='test.emf'

;Create the plot
PLOT, x, y

;Close the device which creates the Metafile
DEVICE, /CLOSE

;Set the device back to the original
SET_PLOT, mydevice
```

The Null Display Device

Device Keywords Accepted by the Null Device:

No keywords are accepted by the `DEVICE` procedure when the `NULL` device is selected.

To suppress graphics output entirely, use the null device:

```
SET_PLOT, 'NULL'
```

The PCL Device

Device Keywords Accepted by the PCL Device:

`CLOSE_FILE`, `COLOR`, `FILENAME`, `FLOYD`, `INCHES`, `LANDSCAPE`, `OPTIMIZE`, `ORDERED`, `PIXELS`, `PORTRAIT`, `RESOLUTION`, `SET_CHARACTER_SIZE`, `SET_COLORMAP`, `THRESHOLD`, `XOFFSET`, `XSIZE`, `YOFFSET`, `YSIZE`

PCL (Printer Control Language) is used by Hewlett-Packard laser and ink jet printers to produce graphics output. To direct graphics output to a PCL file, issue the command:

```
SET_PLOT, 'PCL'
```

This causes IDL to use the PCL driver for producing graphical output. Once the PCL driver is enabled via `SET_PLOT`, the `DEVICE` procedure is used to control its actions, as described below. The default settings for the PCL driver are listed in the following table:

Feature	Value
File	idl.pcl
Mode	Portrait
Optimization level	0 (None)
Dither method	Floyd-Steinberg
Resolution	300 dpi
Horizontal offset	1/2 in.
Vertical offset	1 in.
Width	7 in.
Height	5 in.

Table A-13: Default PCL Driver Settings

The PCL device draws into a memory buffer of the specified size (or the default size, if the `XSIZE` and `YSIZE` keywords to `DEVICE` are not specified). Anything drawn outside this buffer will be silently discarded.

Note

Unlike monitors where white is the most visible color, PCL writes black on white paper. Setting the output color index to 0, the default when PCL output is selected, writes in black. A color index of 255 writes white which is invisible on white paper.

Color tables are not used with PCL unless the color mode has been enabled using the **COLOR** keyword to the **DEVICE** procedure. For images, color dithering produces realistic color image output even though PCL printers only produce eight output colors. In most cases, simply choosing an appropriate color table (using **LOADCT** or **XLOADCT**), or creating a color table from an image (via **TVLCT**) will work fine. If you need finer control over the colors used, see the **SET_COLORMAP** keyword for additional information. For vector graphics, only eight colors are supported—no line dithering is implemented. Any RGB component that is not zero is treated as 255. The correct RGB definitions for each color are shown in the following table. Use the **HELP , /DEVICE** command to view the current options for PCL output.

Color	Red Value	Green Value	Blue Value
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Cyan	0	255	255
Magenta	255	0	255
Yellow	255	255	0
Black	0	0	0
White	255	255	255

Table A-14: PCL RGB Color Definitions

The Printer Device

Device Keywords Accepted by the PRINTER Device:

`CLOSE_DOCUMENT`, `GET_CURRENT_FONT`, `GET_FONTNAMES`,
`GET_FONTNUM`, `GET_PAGE_SIZE`, `INDEX_COLOR`, `PORTRAIT`,
`SCALE_FACTOR`, `SET_CHARACTER_SIZE`, `TRUE_COLOR`, `XOFFSET`,
`XSIZE`, `YOFFSET`, `YSIZE`

The PRINTER device allows IDL Direct Graphics to be output to a system printer. To direct graphics output to a printer, issue the command:

```
SET_PLOT, 'printer'
```

This causes IDL to use a printer driver to produce graphical output. By default, the default system printer is used for output. Use the `DIALOG_PRINTERSETUP` function to define the printing parameters for the printer device. Use the `DIALOG_PRINTJOB` function to control the print job itself.

Note that the printer device is an IDL Direct Graphics device. Like other Direct Graphics devices, you must change to the new device and then issue the IDL commands that send output to that device. With the printer device, you must use the `CLOSE_DOCUMENT` keyword to the `DEVICE` routine to actually initiate the print job and make something come out of your printer.

The PostScript Device

Device Keywords Accepted by the PS Device:

AVANTGARDE, BITS_PER_PIXEL, BKMAN, BOLD, BOOK, CLOSE_FILE, CMYK, COLOR, COURIER, DEMI, ENCAPSULATED, FILENAME, FONT_INDEX, FONT_SIZE, HELVETICA, INCHES, ISOLATIN1, ITALIC, LANDSCAPE, LIGHT, MEDIUM, NARROW, OBLIQUE, OUTPUT, PALATINO, PORTRAIT, PRE_DEPTH, PRE_XSIZE, PRE_YSIZE, PREVIEW, SCALE_FACTOR, SCHOOLBOOK, SET_CHARACTER_SIZE, SET_FONT, SYMBOL, TIMES, TT_FONT, XOFFSET, XSIZE, YOFFSET, YSIZE, ZAPFCHANCERY, ZAPFDINGBATS

PostScript is a programming language designed to convey a description of a page containing text and graphics. Many laser printers and high-resolution, high-quality photo typesetters support PostScript. Color output or direct color separations can be produced with color PostScript. To direct graphics output to a PostScript file, issue the command:

```
SET_PLOT, 'PS'
```

This causes IDL to use the PostScript driver for producing graphical output. Once the PostScript driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described below. The default settings are given in the following table:

Feature	Value
File	idl.ps
Mode	Portrait, non-encapsulated, no color
Horizontal offset	3/4 in.
Vertical offset	5 in.
Width	7 in.
Height	5 in.
Scale factor	1.0
Font size	12 points

Table A-15: Default PostScript Driver Settings

Feature	Value
Font	Helvetica
# Bits / Image Pixel	4

Table A-15: Default PostScript Driver Settings (Continued)

Note

Unlike monitors where white is the most visible color, PostScript writes black on white paper. Setting the output color index to 0, the default when PostScript output is selected, writes black. A color index of 255 writes white which is invisible on white paper. Color tables are not used with PostScript unless the color mode has been enabled using the DEVICE procedure. See “[Color Images](#)” on page 4304

To obtain adequate resolution, the device coordinate system used for PostScript output is expressed in units of 0.001 centimeter (i.e., 1000 pixels/cm).

Use the `HELP, /DEVICE` call to view the current font, file, and other options set for PostScript output.

Using PostScript Fonts

Information necessary for rendering a set of 35 standard PostScript fonts are included with IDL. (The standard 35 fonts are the fonts found on the Apple Laserwriter II PostScript printer; the same fonts are found on almost any PostScript printer made in the time since the LaserWriter II appeared.) Use of PostScript fonts is discussed in detail in “[About Device Fonts](#)” on page 4424.

Color PostScript

If you have a color PostScript device you can enable the use of color with the statement:

```
DEVICE, /COLOR
```

Enabling color also enables the color tables. Text and graphic color indices are translated to RGB by dividing the red, green and blue color table values by 255. As with most display devices, color indices range from 0 to 255. Zero is normally black and white is normally represented by an index of 255. For example, to create and load a color table with four elements, black, red, green and blue:

```
TVLCT, [0,255,0,0], [0,0,255,0], [0,0,0,255]
```

Drawing text or graphics with a color index of 0 results in black, 1 in red, 2 in green, and 3 in blue.

Color Images

As with black and white PostScript, images may be output with 1, 2, 4, or 8 bits, yielding 1, 2, 16, or 256 possible colors. In addition, images are either pseudo-color or TrueColor. A pseudo-color image is a two dimensional image, each pixel of which is used to index the color table, thereby obtaining an RGB value for each possible pixel value. Pseudo-color images are similar to those displayed using the workstation monitor.

Note

In the case of pseudo-color images of fewer than 8 bits, the number of columns in the image should be an exact multiple of the number of pixels per byte (i.e., when displaying 4 bit images the number of columns should be even, and 2 bit images should have a column size that is a multiple of 4). If the image column size is not an exact multiple, extra pixels with a value of 255 are output at the end of each row. This causes no problems if the color white is loaded into the last color table entry, otherwise a stripe of the last (index 255) color is drawn to the right of the image.

TrueColor Images

A TrueColor image consists of an array with three dimensions, one of which has a size of three, containing the three color components. It may be considered as three two dimensional images, one each for the red, green and blue components. For example a TrueColor n by m element image can be ordered in three ways: pixel interleaved $(3, n, m)$, row interleaved $(n, 3, m)$, or image interleaved $(n, m, 3)$. By convention the first color is always red, the second green, and the last is blue.

TrueColor images are also routed through the color tables. The red color table array contains the intensity translation table for the red image, and so forth. Assuming that the color tables have been loaded with the vectors R , G , and B , a pixel with a color value of (r, g, b) is displayed with a color of (R_r, G_g, B_b) . As with other devices, a color table value of 255 represents maximum intensity, while 0 indicates an absence of the color. To pass the RGB pixel values without change, load the red, green and blue color tables with a ramp with a slope of 1.0:

```
TVLCT, INDGEN(256), INDGEN(256), INDGEN(256)
```

or with the LOADCT procedure:

```
; Load standard black/white table:
LOADCT, 0
```

Use the TRUE keyword to the TV and TVSCL procedures to indicate that the image is a TrueColor image and to specify the dimension over which color is interleaved. A value of 1 specifies pixel interleaving, 2 is row interleaving, and 3 is image interleaving. The following example writes a 24-bit image, interleaved over the third dimension, to a PostScript file:

```
SET_PLOT, 'PS'
;Set the PostScript device to *8* bits per color, not 24:
DEVICE, FILE='24bit.ps', /COLOR, BITS=8
TV, [[r]], [[g]], [[b]], TRUE=3
DEVICE, /CLOSE
; Return plotting to X windows:
SET_PLOT, 'X'
```

Note

Currently, the PostScript device does not support TrueColor plots. Only TrueColor images are supported.

Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when displayed with PostScript. This is easily done with the statement:

```
a(WHERE(a EQ 0B)) = 255B
```

CMYK Output

When the CMYK keyword is set, IDL converts the PostScript output to the CMYK color model as a final step. The indexed or RGB data is mapped to the cyan, magenta, yellow, and black color model just prior to output.

The CMYK format is a subtractive color model that is better suited for color printing. This color model allows some document processing applications and printer control software to easily adjust the overall brightness of an image by manipulating a single channel (K). PostScript files created with the CMYK color model must be processed by an interpreter capable of interpreting PostScript Language Level 2 or higher.

PostScript Positioning

Using the XOFFSET and YOFFSET Keywords

Often, IDL users are confused by the use of the XOFFSET and YOFFSET keywords to the PostScript DEVICE routine. These keywords control the position of IDL plots

on the page. XOFFSET specifies the “X” position of the lower left corner of the output generated by IDL. This offset is always taken relative to the lower left-hand corner of the page when viewed in portrait orientation. YOFFSET specifies the “Y” position of the lower left corner of the output generated by IDL. This offset is also taken relative to the lower left-hand corner of the page when viewed in portrait orientation.

The following figure shows how the XOFFSET and YOFFSET keywords are interpreted by the PostScript device in the Portrait (left) and Landscape (right) modes. Note that the landscape plot uses the same origin for determining the effect of the XOFFSET and YOFFSET keywords, but that the output is rotated 270 degrees clockwise.

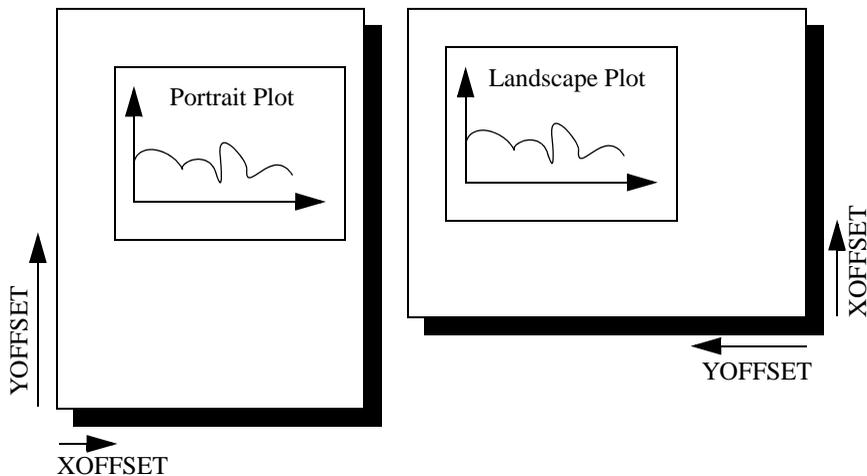


Figure A-1: Interpretation of the XOFFSET and YOFFSET Keywords

The page on the left shows an IDL plot printed in “portrait” orientation. Note that the X and Y offsets work just as we expect them to—increasing the XOFFSET moves the plot to the right and increasing the YOFFSET moves the plot up the page. The page on the right shows an IDL plot printed in “landscape” orientation. Here, the X and Y offsets are still taken relative to the same points even though the orientation of the plot has changed. This happens because IDL moves the origin of the plot *before* rotating the PostScript coordinate system 270 degrees clockwise for the landscape plot.

Note

The XOFFSET and YOFFSET keywords have no effect when you generate ENCAPSULATED PostScript output.

Encapsulated PostScript Output

Another form of PostScript output is Encapsulated PostScript. This is the format used to import PostScript files into page layout and desktop publishing programs. An Encapsulated PostScript (EPS) file is similar to a regular PostScript file except that it contains only one page of PostScript output contained in a “bounding box” that is used to tell other programs about the size and aspect ratio of the encapsulated image.

Most of the time, output from IDL to an EPS file is properly scaled into the EPS bounding box because commands such as PLOT take full advantage of the plotting area made available to them. Sometimes, however, the default bounding box is inappropriate for the image being displayed.

As an example, suppose you have an image that is narrow and tall that, when TV'ed to an IDL window, fills only a small portion of the plotting window. Similarly, when output to an EPS file, this image will only fill a small portion of the bounding box. When the resulting EPS file is brought into a desktop publishing program, it becomes very hard to properly scale the image since the aspect ratio of the bounding box bears no relation to the aspect ratio of the image itself.

To solve this problem, use the XSIZE and YSIZE keywords to the DEVICE procedure to make the bounding box just large enough to contain the image. Since IDL uses a resolution of 1000 dots per centimeter with the PostScript device, the correct XSIZE and YSIZE (in centimeters) can be computed as:

- $XSIZE = \text{Width of image in pixels} / 1000.0 \text{ pixels per cm}$
- $YSIZE = \text{Height of image in pixels} / 1000.0 \text{ pixels per cm}$

The following IDL procedure demonstrates this technique. This procedure reads an X Windows Dump file and writes it back out as a properly-sized, 8-bit-color Encapsulated PostScript file:

```
PRO XWDTOEPS, filename
; Read the XWD file. Pixel intensity information is stored
; in the variable 'array'. Values to reconstruct the color
; table are stored in 'r', 'g', and 'b':
array = READ_XWD(filename, r, g, b)
; Reconstruct the color table:
TVLCT, r,g,b
; Display the image in an IDL window:
TV, array
```

```

; Find the size of the picture. The width of the picture
; (in pixels) is stored in s[1]. The height of the picture
; is stored in s[2]:
s = SIZE(array)
; Take the 'xwd' (for X Windows Dump) extension off of
; the old filename and replace it with 'eps':
fl = STRLEN(filename)
filename = STRMID(filename, 0, fl-4)
filename = filename + '.eps'
PRINT, 'Making file: ', filename
PRINT, s
; Set the plotting device to PostScript:
SET_PLOT, 'ps'
; Use the DEVICE procedure to make the output encapsulated,
; 8 bits, color, and only as wide and high as it needs to
; be to contain the XWD image:
DEVICE, /ENCAPSUL, BITS_PER_PIXEL=8, /COLOR, $
      FILENAME=filename, XSIZE=S[1]/1000., $
      YSIZE=S[2]/1000.
; Write the image to the file:
TV, array
; Close the file:
DEVICE, /CLOSE
; Return plotting to X Windows:
SET_PLOT, 'x'
END

```

Multiple Plots on the Same Page

To put multiple plots on the same PostScript page, use the `!P.MULTI` system variable (described in more detail in [“!P System Variable”](#) on page 4379). `!P.MULTI` is a 5-element integer array that controls the number of rows and columns of plots to make on a page or in a graphics window.

The first element of `!P.MULTI` is a counter that reports how many plots remain on the page. The second element of `!P.MULTI` is the number of columns per page. The third element is the number of rows per page.

For example, the following lines of code create a PostScript file, `multi.ps`, with 6 different plots arranged as 2 columns and 3 rows:

```

; Set plotting to PostScript:
SET_PLOT, 'PS'
; Set the filename:
DEVICE, FILENAME='multi.ps'
; Make IDL's plotting area hold 2 columns and 3 rows of plots:
!P.MULTI = [0, 2, 3]
; Create a simple dataset:
A = FINDGEN(10)

```

```

; Make 6 different plots:
PLOT, A
PLOT, SIN(A)
PLOT, COS(A)
PLOT, TAN(A)
PLOT, TANH(A)
PLOT, SINH(A)
; Close the file:
DEVICE, /CLOSE
; Return plotting to Windows:
SET_PLOT, 'win'
; Reset plotting to 1 plot per page:
!P.MULTI = 0

```

The resulting file produces a set of plots as shown in the following figure:

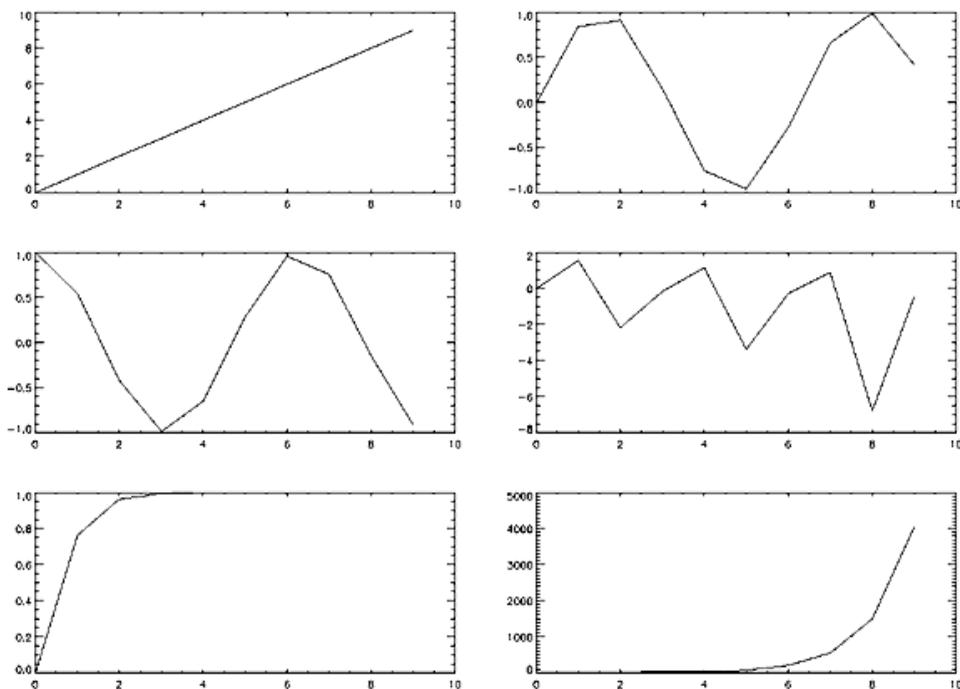


Figure A-2: Multiple plots on a single page produced by setting the `!P.MULTI` system variable.

Importing IDL Plots into Other Documents

This section shows how to generate IDL PostScript graphics so that they can be inserted into other documents. It also provides several examples of how the PostScript graphics device is used. Simply omit the `ENCAPSULATED` keyword from the calls to `DEVICE` if you wish to produce plots that can be printed directly. The following figure is an encapsulated PostScript file suitable for inclusion in other documents. The figure was produced with the following IDL statements. Note the use of the `ENCAPSULATED` keyword in the call to `DEVICE`:

```
; Select the PostScript driver:
SET_PLOT, 'PS'
; Note use of ENCAPSULATED keyword:
DEVICE, /ENCAPSULATED, FILENAME = 'pic1.ps'
x = FINDGEN(200)
; Plot the sine wave:
PLOT, 10000 * SIN(x/5) / EXP(x/100), $
      LINESTYLE = 2, TITLE = 'IDL PostScript Plot', $
      XTITLE = 'Point Number', YTITLE='Y Axis Title', $
      FONT = 0
; Add the cosine:
OPLOT, 10000 * COS(x/5) / EXP(x/100), LINESTYLE = 4
; Annotate the plot:
XYOUTS, 100, -6000, 'Sine', FONT = 0
OPLOT, [120, 180], [-6000, -6000], LINESTYLE = 2
XYOUTS, 100, -8000, 'Cosine', FONT = 0
OPLOT, [120, 180], [-8000, -8000], LINESTYLE = 4
```

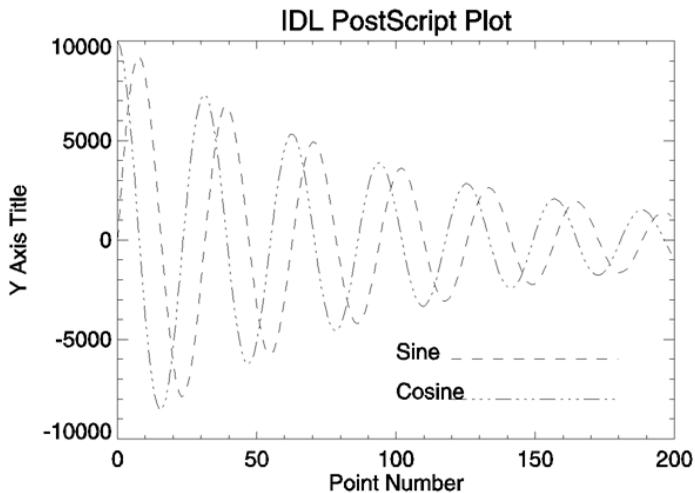


Figure A-3: Sample PostScript plot using Helvetica font.

The following figure is a more complicated plot. It demonstrates some of the three-dimensional plotting capabilities of IDL. It was produced with the following IDL statements:

```

; Select the PostScript driver:
SET_PLOT, 'PS'
; Note use of ENCAPSULATED keyword:
DEVICE, /ENCAPSULATED, FILENAME = 'pic2.ps'
; Access the data:
OPENR, 1, FILEPATH('abnorm.dat', SUBDIR=['examples', 'data'])
aa = ASSOC(1, BYTARR(64, 64))

```

```

; Get a smoothed version:
a = SMOOTH(aa[0], 3)
; Generate the surface:
SURFACE, a, /SAVE, ZAXIS = 1, XSTYLE = 1, YSTYLE = 1
; Add the contour:
CONTOUR, a, /T3D, /NOERASE, ZVALUE = 1, $
    XSTYLE = 1, YSTYLE = 1, C_LINestyle = [0,1,2], $
    TITLE = 'IDL PostScript Plot'

CLOSE, 1

```

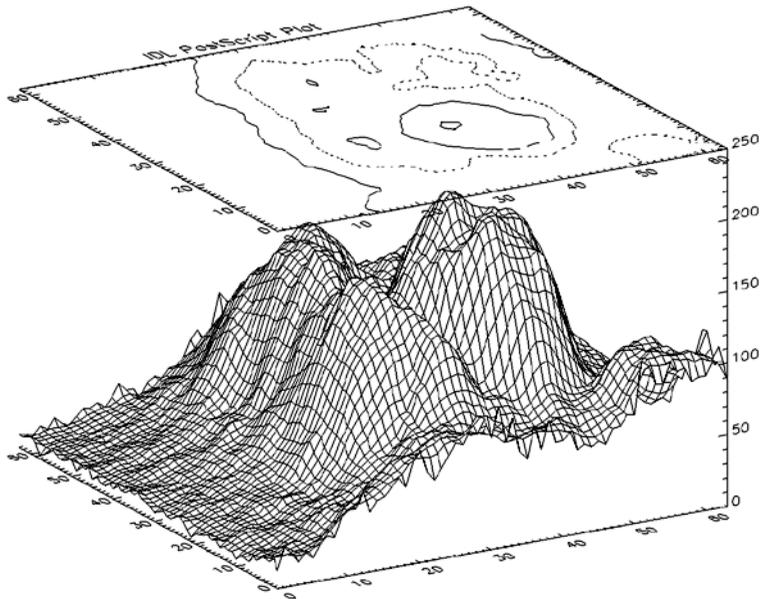


Figure A-4: Three-Dimensional Plot with Vector-Drawn Characters

The following figure illustrates polygon filling. It was produced with the following IDL statements:

```

SET_PLOT, 'PS'
DEVICE, /ENCAPSULATED, FILENAME = 'pic3.ps'
x = FINDGEN(200)
; Upper sine wave:
a = 10000 * sin(x / 5) / exp(x / 100)
PLOT, a, /NODATA, TITLE = 'IDL PostScript Plot', $
    XTITLE='Point Number', YTITLE='Y Axis Title', $
    FONT = 0

```

```

; Vector of X vertices for polygon filling. Note that the
; ROTATE(V,2) function call returns the vector V in reverse order:
C = [X, ROTATE(X, 2)]
; Vector of Y vertices for polygon filling:
D = [A, ROTATE(A-2000, 2)]
; Fill the region using an intensity of about 75% white:

POLYFILL, C, D, COLOR=192

```

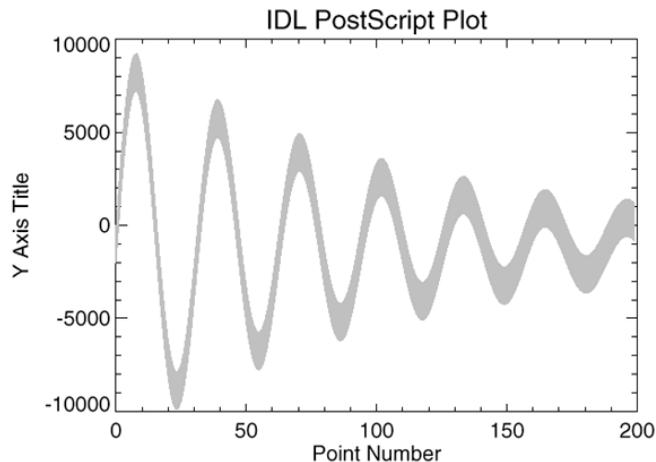


Figure A-5: Polygon Filling Example

The following figure illustrates IDL PostScript images. In this case, the same image is reproduced four times. In each case, a different number of bits are used per image pixel. It was produced with the following IDL statements:

```

SET_PLOT, 'PS'
DEVICE, /ENCAPSULATED, FILENAME = 'pic4.ps'
; Open image file:
OPENR, 1, FILEPATH('people.dat', SUBDIR = ['examples','data'])
; Variable to hold image:
a = BYTARR(192, 192, /NOZERO)
; Input the image:
READU, 1, a
; Done with the file:
CLOSE, 1
; Add a color table ramp to the bottom of the image:
A[0,0] = BYTSCL(INDGEN(192))#REPLICATE(1,16)
; Output the image four times:
FOR i = 0,3 DO BEGIN
    ; Use 1, 2, 4, and 8 bits per pixel:

```

```
DEVICE, BITS_PER_PIXEL=2^i
; Output using TV with position numbers 0, 1, 2, and 3:
TV, a, i, XSIZE=2.5, YSIZE=2.5, /INCHES

ENDFOR
```

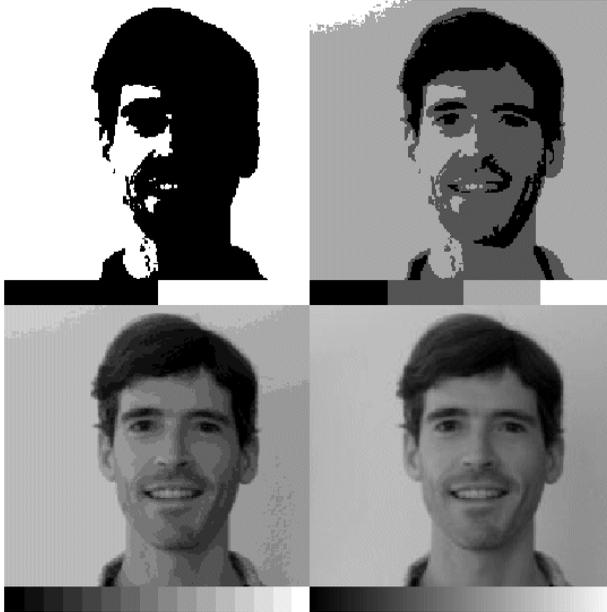


Figure A-6: 1, 2, 4, and 8-bit PostScript Images

The Regis Terminal Device

Device Keywords Accepted by the REGIS Device:

[AVERAGE_LINES](#), [CLOSE_FILE](#), [FILENAME](#), [PLOTTER_ON_OFF](#),
[SET_CHARACTER_SIZE](#), [TTY](#), [VT240](#), [VT241](#), [VT340](#), [VT341](#)

IDL provides Regis graphics output for the DEC VT240, VT330, and VT340 series of terminals. To output graphics to such terminals, issue the IDL command:

```
SET_PLOT, 'REGIS'
```

This causes IDL to use the Regis driver for producing graphical output.

Defaults for Regis Devices

The default setting for Regis output is: VT340, 16 colors, 4 bits per pixel.

Regis Limitations

- Four colors are available with VT240 and VT241 terminals, sixteen colors are available with the VT330 and VT340.
- Thick lines are emulated by filling polygons. There may be a difference in linestyle appearance between thick and normal lines.
- Image output is slow and is of poor quality, especially on the VT240 series. The VT240 is only able to write pixels on even numbered screen lines. IDL offers two methods of writing images to the 240:
 - Even and odd pairs of rows are averaged and written to the screen. An n, m image will occupy n columns and m screen rows. If this method is selected, graphics and image coordinates coincide. This method is the default (`AVERAGE_LINES = 1`). Routines that rely on a uniform graphics and image coordinate system, such as `SHADE_SURF`, work only in this mode.
 - Each line of the image is written to the screen, displaying every image pixel. An n, m image occupies $2m$ lines on the screen. (`AVERAGE_LINES = 0`). Graphics and image coordinates coincide only at the lower left corner of the image.
- Pixel values cannot be read back from the terminal, rendering the `TVRD` function inoperable.

The Tektronix Device

Device Keywords Accepted by the REGIS Device:

`CLOSE_FILE`, `COLORS`, `FILENAME`, `GIN_CHARS`, `PLOT_TO`,
`RESET_STRING`, `SET_CHARACTER_SIZE`, `SET_STRING`, `TEK4014`, `TEK4100`,
`TTY`

The Tektronix 4000 (4010, 4014, etc.), 4100 and 4200 series of graphics terminals (and the multitude of terminals and microcomputers that emulate them) are among the most common graphics devices available. To use IDL graphics with such terminals, issue the command:

```
SET_PLOT, 'TEK'
```

This causes IDL to use the Tektronix driver for producing graphical output. Once the Tektronix driver is enabled via `SET_PLOT`, the `DEVICE` procedure is used to control its actions, and to configure IDL for the specific features of your terminal. If you never call the `DEVICE` procedure, IDL assumes a plain vanilla Tektronix 4000 series compatible terminal. The 4200 series is upwardly compatible with the 4100 series; all references to the 4100 series also include the 4200 series. To set up IDL for use with a 4100 series compatible terminal with n colors, enter the following commands:

```
SET_PLOT, 'TEK'  

DEVICE, /TEK4100, COLORS = n
```

The number of colors should be set to 2^B where B is the number of bit planes in your terminal. If you use a Tektronix compatible terminal that requires calling the `DEVICE` procedure for configuration, you should probably create and use a start-up procedure that calls the `DEVICE` procedure, as described in Chapter 2. Because of the tremendous variation among the requirements and abilities of these terminals, it is crucial that you configure IDL properly for your terminal. In particular, the mode switching character sequences, set by the keyword parameters `SET_STRING` and `RESET_STRING` must be set correctly.

The DEVICE Procedure For Tektronix Terminals

The default setting for Tektronix output is: 10-bit coordinates, 4000 series terminals, and no use of color. The `DEVICE` keywords can be used to modify these defaults.

Tektronix Limitations

- The line drawing procedures work with all models. Line style and color capabilities vary greatly among terminal models and/or emulation programs.

- Color and the display of images (albeit very slowly and frequently of a poor quality because of the small number of colors) is usable only with 4100 series terminals. Hardware polygon fill and thick lines do not work with the 4000 series.
- The image coordinate system does not match the graphics coordinate system. The graphics coordinates range from 0 to 3071 in Y, and from 0 to 4095 in X. Image coordinates vary according to terminal model. A typical range is from 0 to 479 in Y, and 0 to 639 in X. Because of this, the SHADE_SURF procedure does not work with Tektronix terminals.

Warning

Not all 4100 series terminals are capable of displaying images—the Tektronix pixel operations option is required. Many terminal emulators do not emulate this option. The Tektronix commands used to output images are: RU, begin pixel operations, RS, set pixel viewport, and RP, raster write. If your terminal or emulator does not accept these commands, you will not be able to display images.

- The Tektronix graphics protocol does not allow the specification of line thickness. Lines with a thickness more than 1.0 are drawn using polygon filling in the case of 4100 series terminals. With 4000 series terminals, thick lines are emulated by drawing multiple thin lines. This scheme will produce artifacts on some Tektronix emulating devices because of differing resolutions, normal line thicknesses and inexact coordinate conversions.

Tektronix Device Limitations

Usage of Tektronix and Tektronix-compatible terminals with IDL has the following limitations:

- Image coordinates do not match the coordinates used by the rest of the graphic procedures. This is because no two models of Tektronix terminals are compatible. The graphics procedures utilize the default coordinate system of 1024 by 780, or 4096 by 3120 in the 12-bit mode. The size of the pixel memory and coordinate system vary widely between models. The *Position* parameter of the TV and TVSCL procedures does not work.
- The cursor can not be positioned from the computer meaning that the TVCRS procedure cannot be used with the Tektronix driver.
- Pixel values may not be read back from the terminal, rendering the TVRD function inoperable.

The Microsoft Windows Device

Device Keywords Accepted by the WIN Device:

`BYPASS_TRANSLATION`, `COPY`, `CURSOR_CROSSHAIR`,
`CURSOR_ORIGINAL`, `CURSOR_STANDARD`, `DECOMPOSED`,
`GET_CURRENT_FONT`, `GET_FONTNAMES`, `GET_FONTNUM`,
`GET_GRAPHICS_FUNCTION`, `GET_SCREEN_SIZE`,
`GET_WINDOW_POSITION`, `PRINT_FILE`, `RETAIN`, `SET_CHARACTER_SIZE`,
`SET_FONT`, `SET_GRAPHICS_FUNCTION`, `TRANSLATION`, `WINDOW_STATE`

The Microsoft Windows version of IDL uses the “WIN” device by default. This device is similar to the X Windows device described below. The “WIN” device is only available in IDL for Windows.

To set plotting to the Microsoft Windows device, use the command:

```
SET_PLOT, 'WIN'
```

The X Windows Device

Device Keywords Accepted by the X Device:

BYPASS_TRANSLATION, COPY, CURSOR_CROSSHAIR, CURSOR_IMAGE, CURSOR_MASK, CURSOR_ORIGINAL, CURSOR_STANDARD, CURSOR_XY, DECOMPOSED, DIRECT_COLOR, FLOYD, GET_CURRENT_FONT, GET_FONTNAMES, GET_FONTNUM, GET_GRAPHICS_FUNCTION, GET_SCREEN_SIZE, GET_VISUAL_NAME, GET_WINDOW_POSITION, GET_WRITE_MASK, ORDERED, PSEUDO_COLOR, RETAIN, SET_CHARACTER_SIZE, SET_FONT, SET_GRAPHICS_FUNCTION, SET_TRANSLATION, SET_WRITE_MASK, STATIC_COLOR, STATIC_GRAY, THRESHOLD, TRUE_COLOR, TTY, WINDOW_STATE

X Windows is a network-based windowing system developed by MIT's project Athena. IDL uses the X System (often referred to simply as "X"), to provide an environment in which the user can create one or more independent windows, each of which can be used for the display of graphics and/or images.

In the X system, there are two basic cooperating processes: *clients* and *servers*. A server consists of a display, keyboard, and pointer (such as a mouse) as well as the software that controls them. Client processes (such as IDL) display graphics and text on the screen of a server by sending X protocol requests across the network to the server. Although in the most common case, the server and client reside on the same machine, this network based design allows much more elaborate configurations.

To use X Windows as the current graphics device, issue the IDL command:

```
SET_PLOT, 'X'
```

This causes IDL to use the X Window System for producing graphical output. Once the X driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described below.

Use the statement:

```
HELP, /DEVICE
```

to view the current state of the X Windows driver.

X Windows Visuals

Visuals specify how the hardware deals with color. The X Window server (your display) may provide colors or only gray scale (black and white), or both. The color

tables may be changeable from within IDL (read-write), or may be fixed (read-only). The value of each pixel value may be mapped to any color (Un-decomposed Colormap), or certain bits of each pixel are dedicated to the red, green, and blue primary colors (Decomposed Colormap).

There are six X Windows visual classes—read-write and read-only visuals for three types of displays: Gray Scale, Pseudo Color, and Decomposed Color. The names of the visuals are shown in the following table:

Visual Name	Writable	Description
StaticGray	no	Gray scale
GrayScale	yes	Gray scale
StaticColor	no	Undecomposed color
PseudoColor	yes	Undecomposed color
TrueColor	no	Decomposed color
DirectColor	yes	Decomposed color

Table A-16: X Windows Visual Classes

IDL supports all six types of visuals, although not at all possible depths. UNIX X Window System users can use the command `xdpypinfo` to determine which visuals are supported by their systems.

Each X Window server has a default visual class. Many servers may provide multiple visual classes. For example, a server with display hardware that supports an 8-bit-deep, un-decomposed, writable color map (PseudoColor), may also easily provide StaticColor, StaticGray, and GrayScale visuals.

You can select the visual used by IDL using the DEVICE procedure before a window is created, or by including the resource `idl.gr_visual` in your X defaults file, as explained in [“Setting the X Window Defaults”](#) on page 4327.

How IDL Selects a Visual Class

When opening the display, IDL asks the display for the following visuals, in order, until a supported visual class is found:

1. DirectColor, 24-bit
2. TrueColor, 24-bit, then 16-bit, then 15-bit
3. PseudoColor, 8-bit, then 4-bit

4. StaticColor, 8-bit, then 4-bit
5. GrayScale, any depth
6. StaticGray, any depth

You can override this behavior by using the `DEVICE` routine to specify the desired visual class and depth before you create a window. For example, if you are using a display that supports both the DirectColor, 24-bit-deep visual, and an 8-bit-deep PseudoColor visual, IDL will select the 24-bit-deep DirectColor visual. To instead use PseudoColor, issue the following command before creating a window:

```
DEVICE, PSEUDO_COLOR = 8
```

The colormap/visual class combination is chosen when IDL first connects with the X Window server. Note that if you connect with the X server by creating a window or using the `DEVICE` keyword to the `HELP` procedure, the visual class will be set; it then cannot be changed until IDL is restarted. If you wish to use a visual class other than the default, be sure to set it with a call to the `DEVICE` procedure *before* creating windows or otherwise connecting with the X Window server.

Windows are created in two ways:

1. Using the `WINDOW` procedure. `WINDOW` allows you to explicitly control many aspects of how the window is created.
2. If no windows exist and a graphics operation requiring a window is executed, IDL implicitly creates window 0 with the default characteristics.

Once the visual class is selected, all subsequently-created windows share the same class and colormap. The number of simultaneous colors available is stored in the system variable `!D.N_COLORS`. The visual class and number of colors, once initialized, cannot be changed without first exiting IDL.

How IDL Obtains a Colormap

IDL chooses the type of colormap in the following manner:

- By default, the shared colormap is used whenever possible (i.e., whenever IDL is using the default visual for the system). All available colors from the shared colormap are allocated for use by IDL. This is what happens when no window currently exists and a graphics operation causes IDL to implicitly create one.
- If the number of colors to use is explicitly specified using the `COLORS` keyword with the `WINDOW` procedure, IDL attempts to allocate the number of colors specified from the shared colormap using the default visual of the screen. If there aren't enough colors available, a private colormap with that number of colors is used instead.

- Specifying a negative value for the `COLORS` keyword to the `WINDOW` procedure causes IDL to attempt to use the shared colormap, allocating all but the specified number of colors. For example:

```
WINDOW, COLORS = -8
```

allocates all but 8 of the currently available colors. This allows other applications that might need their own colors to run in tandem with IDL.

- If a visual type and depth is specified, via the `DEVICE` procedure, which does not match the default visual of the screen, a new, private, colormap is created.

Using Color Under X

Colormaps define the mapping from color index to screen color. Two attributes of colormaps are important to the IDL user: they may be *private* or *shared*; and they may be *static* or *writable*. These different types of colormaps are described below.

Shared Colormaps

The window manager creates a colormap when it is started. This is known as the default colormap, and can be shared by most applications using the display. When each application requires a colormap entry (i.e., a mapping from a color index to a color), it allocates one from this shared table. Advantages and disadvantages of shared colormaps include:

- Using the shared colormap ensures that all applications share the available colors without conflict. A given application will not change a color that is allocated to a different application. In the case of IDL it means that IDL can change the colors it has allocated without changing the colors in use by the window manager or other applications.
- The window system interface routines must translate between the actual and allocated pixel values, significantly slowing the transfer of images.
- The shared colormap might not have enough colors available to perform the desired operations with IDL.
- The number of available colors in the shared colormap depends on the window manager in use and the demands of other applications. Thus, the number of available colors can vary.
- The allocated colors in a shared colormap do not generally start at zero and they are not necessarily contiguous. This makes it difficult to use the write mask for certain operations.

Private Colormaps

An application can create its own private color map. Most hardware can only display a single colormap at a time, so these private colormaps are called virtual color maps, and only one at a time is actually in use and visible. When the window manager gives the color focus to a window with a private colormap, the X window system loads its virtual colormap into the hardware colormap.

- Every color index supported by the hardware is available to IDL, improving the quality of images.
- Allocated colors always start at zero and are contiguous. This simplifies using the write mask.
- No translation between internal pixel values and the values required by the server is required, making the transfer of images more efficient.
- When the IDL colormap is loaded, other applications are displayed using the wrong colors. Furthermore, colors from the shared colormap are usually allocated from the lower end of the map first. These are the colors allocated by the window manager for such things as window borders, the color of text, and so forth. Since most IDL colormaps have very dark colors in the lower entries, the end effect with the IDL colormap loaded is that the non-IDL portions of the screen go blank.

Static Colormaps

As mentioned above, the contents of static colormaps are determined outside of IDL and cannot be changed. When using a static colormap, the TVLCT procedure simulates writable colormaps by finding the closest RGB color entry in the colormap to the requested color. The colormap translation table is then set to map IDL color indices to those of the closest colors in the colormap.

The colors present in the colormap may, and probably will, *not* match the requested colors exactly. For example, with a typical static color map, loading the IDL standard color table number 0, which consists of 256 intensities of gray, results in only 8 or 16 distinct intensities.

With static colormaps, loading a new color table does not affect the appearance of previously written objects. The internal translation tables are modified, which only affects objects that are subsequently written.

Color Translation

As mentioned above, colors from the shared colormap do not necessarily start from index zero, and are not necessarily contiguous. IDL preserves the illusion of a zero

based contiguous colormap by maintaining a translation table between user color indices, which range from 0 to !D.TABLE_SIZE, and the actual pixel values allocated from the X server. Normally, the user need not be concerned with this translation table, but it is available using the statement:

```
DEVICE, TRANSLATION=T
```

This statement stores the current translation table, a 256 element byte vector, in the variable T. Element zero of the vector contains the value pixel allocated for the zeroth color in the IDL colormap, and so forth. In the case of a private colormap, each element of the translation vector contains it's own index value, because private colormaps start at zero and are contiguous.

The translation table may be bypassed, allowing direct access to the display's color indices, by setting the BYPASS_TRANSLATION keyword in the DEVICE procedure.

```
DEVICE, /BYPASS_TRANSLATION
```

Translation can be reestablished by setting the keyword to zero:

```
DEVICE, BYPASS_TRANSLATION=0
```

By default, the color translation tables are bypassed if the visual type is TrueColor or the visual type is DirectColor with a private colormap.

Using Pixmap

X Windows can direct graphics to *windows* or *pixmap*s. Windows are the usual windows that appear on the screen and contain graphics. Pixmap are invisible graphics memory contained in the server. Drawing to a window produces a viewable result, while drawing to a pixmap simply updates the pixmap memory.

Pixmap are useful because it is possible to write graphics to a pixmap and then copy the contents of the pixmap to a window where it can be viewed. Furthermore, this copy operation is very fast because it happens entirely within the server. Provided enough pixmap memory is available, this technique works very well for animating a series of images by placing the images into pixmap memory and then sequentially copying them to a visible window.

To create a pixmap, use the PIXMAP keyword with the WINDOW procedure. For example, to create a square pixmap with 128 pixels per side as IDL window 1, use the command:

```
WINDOW, 1, /PIXMAP, XSIZE=128, YSIZE=128
```

Once they are created, pixmaps are treated just like normal windows, although some operations (WSHOW for instance) don't do anything useful when applied to a pixmap.

The following procedure shows how animation can be done using pixmap memory. It uses a series of 15 heart images taken from the file `abnorm.dat`. This file is supplied with all IDL distributions in the `examples/data` subdirectory of the main IDL directory. It creates a pixmap and writes the heart images to it. It then uses the `COPY` keyword of the `DEVICE` procedure to copy the images to a visible window. Pressing any key causes the display process to halt:

```

; Animate heart series:
PRO animate_heart
; Open the file containing the images:
OPENR, u, FILEPATH('abnorm.dat', SUBDIR = ['examples','data']), $
  /GET_LUN
; Associate a file variable with the file. Each heart image
; is 64x64 pixels:
frame = ASSOC(u, BYTARR(64,64))
; Window 0 is a pixmap which is 4 images tall and 4
; images wide. The images will be placed in this pixmap:
WINDOW, 0, /PIXMAP, XSIZE = 512, YSIZE = 512, /FREE
; Write each image to the pixmap. SMOOTH is used to improve
; the appearance of each image and REBIN is used to
; enlarge/shrink each image to the final display size:
FOR i=0, 15-1 DO TV, REBIN(SMOOTH(frame[i],3), 128, 128),i
; Close the image file and free the file unit:
FREE_LUN, u
; The visible window will be used to display the animated heart
; cycle:
WINDOW, XSIZE = 128, YSIZE=128, TITLE='Heart', /FREE
; Current frame number:
i = 0L
; Display frames until any key is pressed:
WHILE GET_KBRD(0) EQ '' DO BEGIN
; Compute x and y locations of pixmap image's lower left corner:
  x = (i mod 4) * 128 & y = 384 - (i/4) * 128
; Copy the next image from the pixmap to the visible window:
DEVICE, COPY = [x, y, 128, 128, 0, 0, 0]
; Keep track of total frame count:
i = (i + 1) MOD 15
ENDWHILE
END

```

Animation sequences with more and/or larger images can be made. See the documentation for the `XANIMATE` procedure, which is a more generalized embodiment of the above procedure.

Note

Some X Windows servers will refuse to create a pixmap that is larger than the physical screen in either dimension.

How Color is Interpreted for a TrueColor Visual

How a color (such as `!P.COLOR`) is interpreted by IDL (when a TrueColor visual is being utilized) depends in part upon the decomposed setting for the device.

To retrieve the decomposed setting:

```
DEVICE, GET_DECOMPOSED = currentDecomposed
```

To set the decomposed setting:

```
DEVICE, DECOMPOSED = newDecomposed
```

If the decomposed value is zero, colors (like `!P.COLOR`) are interpreted as indices into IDL's color table. A color should be in the range from 0 to `!D.TABLE_SIZE - 1`. The IDL color table contains a red, green, and blue component at a given index; each of these components is in the range of 0 up to 255.

Note

IDL's color table does not map directly to a hardware color table for a TrueColor visual. If IDL's color table is modified, for example using the `LOADCT` or `TVLCT` routines, then the new color table will only take effect for graphics that are drawn after it has been modified.

If the decomposed value is non-zero, colors (like `!P.COLOR`) are interpreted as a combination of red, green, and blue settings. The least significant 8 bits contain the red component, the next 8 bits contain the green component, and the most significant 8 bits contain the blue component.

In either case, the most significant bits of each of the resulting red, green, and blue components are utilized. The number of bits utilized per component depends upon the red, green, and blue masks for the visual. On UNIX systems, a new field (Bits Per RGB) has been added to the output from `HELP /DEVICE`. This Bits Per RGB field indicates the amount of bits utilized for each component.

Tip

The UNIX command `xdisplayinfo` also provides information about each of the visuals.

Setting the X Window Defaults

You can set the initial default value of the following parameters by setting resources in the file `.xdefaults` file in your home directory as follows:

Resource Name	Description
<code>idl.colors</code>	The number of colors used by IDL.
<code>idl.gr_depth</code>	The depth, in bits, of the visual used by IDL.
<code>idl.retain</code>	The default setting for the <code>retain</code> parameter: 0=none, 1= by server, 2=by IDL.
<code>idl.gr_visual</code>	The type of visual: StaticGray, GrayScale, StaticColor, PseudoColor, TrueColor, or DirectColor.

Table A-17: IDL/ X Window Defaults

For example, to set the default visual to PseudoColor, and to allocate 100 colors, insert the following lines in your defaults file:

```
idl.gr_visual: PseudoColor
idl.colors: 100
```

The Z-Buffer Device

Device Keywords Accepted by the Z Device:

[CLOSE](#), [GET_GRAPHICS_FUNCTION](#), [GET_WRITE_MASK](#),
[SET_CHARACTER_SIZE](#), [SET_COLORS](#), [SET_FONT](#),
[SET_GRAPHICS_FUNCTION](#), [SET_RESOLUTION](#), [Z_BUFFERING](#)

The IDL Z-buffer device is a pseudo device that draws 2-D or 3-D graphics in a buffer contained in memory. This driver implements the classic Z buffer algorithm for hidden surface removal. Although primarily used for 3-D graphics, the Z-buffer driver can be used to create 2-D objects in a frame buffer in memory. The resolution of this device can be set by the user.

All of the IDL plotting and graphics routines work with the Z-buffer device driver. In addition, the POLYFILL procedure has a few keyword parameters, allowing Gouraud shading and warping images over 3-D polygons, that are only effective when used with the Z-buffer.

When used for 3-D graphics, two buffers are present: an 8-bit-deep frame buffer that contains the picture; and a 16-bit-deep Z-buffer of the same resolution, containing the z-value of the visible surface of each pixel. The Z-buffer is initialized to the depth at the back of the viewing volume. When objects are drawn, the z-value of each pixel is compared with the value at the same location in the Z-buffer, and if the z-value is greater (closer to the viewer), the new pixel is written in the frame buffer and the Z-buffer is updated with the new z-value.

The Z-buffer device is a “pseudo device” in that drawing commands update buffers in memory rather than sending commands to a physical device or file. The TVRD function reads the contents of either buffer to an IDL array. This array may then be further processed, written to a file, or output to a raster-based graphics output device.

The Z-buffer driver can be used for 2-D graphics by disabling the depth computations.

To use the Z-buffer as the current graphics device, issue the IDL command:

```
SET_PLOT, 'Z'
```

Once the Z-buffer driver is enabled the DEVICE procedure is used to control its actions, as described below.

Use the statement:

```
HELP, /DEVICE
```

to view the current state of the Z-buffer driver and the amount of memory used for the buffers.

Reading and Writing Buffers

The contents of both buffers are directly accessed by the TV and TVRD routines. The frame buffer that contains the picture is 8 bits deep and is accessed as channel 0. The Z depth buffer contains 16 bit integers and is accessed as channel 1. Always use CHANNEL=1 and set the keyword WORDS when reading or writing the depth buffer.

The normal procedure is to set the graphics device to “Z”, draw the objects, read the frame buffer, and then select another graphics device and write the image. For example, to create an image with the Z-buffer driver and then display it on an X-Window display:

```

; Select Z-buffer device:
SET_PLOT, 'Z'
; Write objects to the frame buffer using normal graphics
; routines, e.g. PLOT, SURFACE, POLYFILL
... ..
; Read back the entire frame buffer:
a=TVRD()
; Select X Windows:
SET_PLOT, 'X'
; Display the contents of the frame buffer:
TV, a

```

To read the depth values in the Z-buffer, use the command:

```
a = TVRD(CHANNEL=1, /WORDS)
```

To write the depth values, use the command:

```
TV, a, /WORDS, CHANNEL=1
```

The TV, TVSCL, and TVRD routines write or read pixels directly to a rectangular area of the designated buffer without affecting the other buffer.

Z-Axis Scaling

The values in the depth buffer are short integers, scaled from -32765 to +32765, corresponding to normalized Z-coordinate values of 0.0 to 1.0.

Polyfill Procedure

The following POLYFILL keywords are active only with the Z-buffer device: IMAGE_COORDINATES, IMAGE_INTERPOLATE, and TRANSPARENT. These

parameters allow images, specified via the PATTERN keyword, to be warped over 2-D and 3-D polygons.

The IMAGE_COORDINATES keyword contains a 2 by N array containing the image space coordinates that correspond to each of the N vertices of the polygon. The IMAGE_INTERPOLATE keyword indicates that bilinear interpolation is to be used, rather than the default nearest neighbor sampling. Pixels less than the value of TRANSPARENT are not drawn, simulating transparency. For Gouraud shading of polygons, the COLOR keyword can contain an array specifying the color index for each polygon vertex.

Examples Using the Z-Buffer

This example forms a Bessel function, draws its shaded surface and overlays its contour, using the Z-buffer as shown in the following figure. The final output is directed to PostScript.

```

; Select the Z-buffer:
SET_PLOT, 'Z'
n = 50 ; Size of array for Bessel
; Make the Bessel function:
a = BESELJ(SHIFT(DIST(n), n/2, n/2)/2, 0)
; Draw the surface, label axes in black, background in white:
SHADE_SURF, a, /SAVE, COLOR=1, BACKGROUND=255
nlev = 8 ; Number of contour levels
; Make the Contour at normalized Z=.6:
CONTOUR, a, /OVERPLOT, ZVALUE=.6, /T3D, $
    LEVELS=FOUNDGEN(nlev)*1.5/nlev-.5, COLOR=1
; Read image:
b=TVRD()
; Select PostScript output:
SET_PLOT, 'PS'
; Output the image:
TV, b
; Close the new PostScript file:
DEVICE, /CLOSE

```

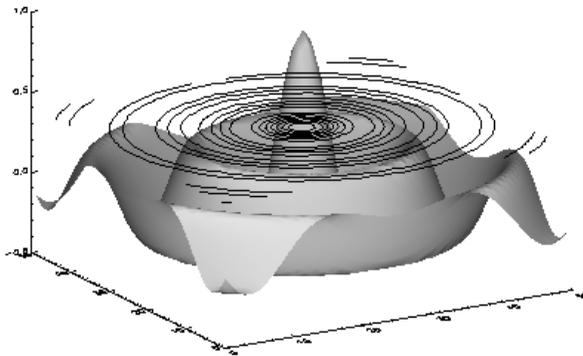


Figure A-7: Combined Shaded Surface and Contour Plot

The following example warps an image to a cube as shown in the figure below. The lower two quadrants of the image are warped to the front two faces of the cube. The upper-right quadrant is warped to the top face of the cube. The image is held in the array `a`, with dimensions `nx` by `ny`. The image is then output to PostScript as in the previous example.

```

; Select the Z-buffer:
SET_PLOT, 'Z'
; Make a white background for final output to PostScript:
ERASE, 255
; Establish 3-D scaling as (0,1) cube:
SCALE3, X RANGE=[0,1], Y RANGE=[0,1], Z RANGE=[0,1]
; Define vertices of cube. Vertices 0-3 are bottom, 4-7 are top:
verts = [[0,0,0], [1,0,0], [1,1,0], [0,1,0], $
         [0,0,1], [1,0,1], [1,1,1], [0,1,1]]
; Fill lower left face:
POLYFILL, verts[*], [3,0,4,7]], /T3D, PATTERN=a, $
         IMAGE_COORD=[[0,0], [nx/2,0], [nx/2,ny/2], [0,ny/2]]
; Fill lower right face:
POLYFILL, verts[*], [0,1,5,4]], /T3D, PATTERN=a, $
         IMAGE_COORD=[[nx/2,0], [nx-1,0], $
         [nx-1,ny/2], [nx/2,ny/2]]
; Fill top face:
POLYFILL, verts[*], [4,5,6,7]], /T3D, PATTERN=a, $
         IMAGE_COORD = [[nx/2,ny/2], [nx-1,ny/2], $
         [nx-1,ny-1], [nx/2,ny-1]]
; Draw edges of cube in black:
PLOTS, verts[*], [0,4]], /T3D, COLOR=0

```

```
    ; Edges of top face:  
PLOTS, verts[*, [4,5,6,7,4]], /T3D, COLOR=0
```

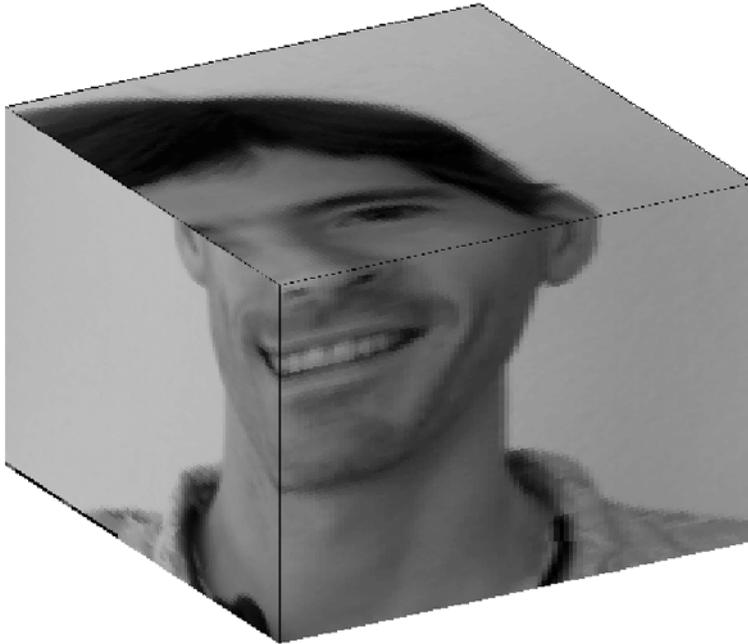
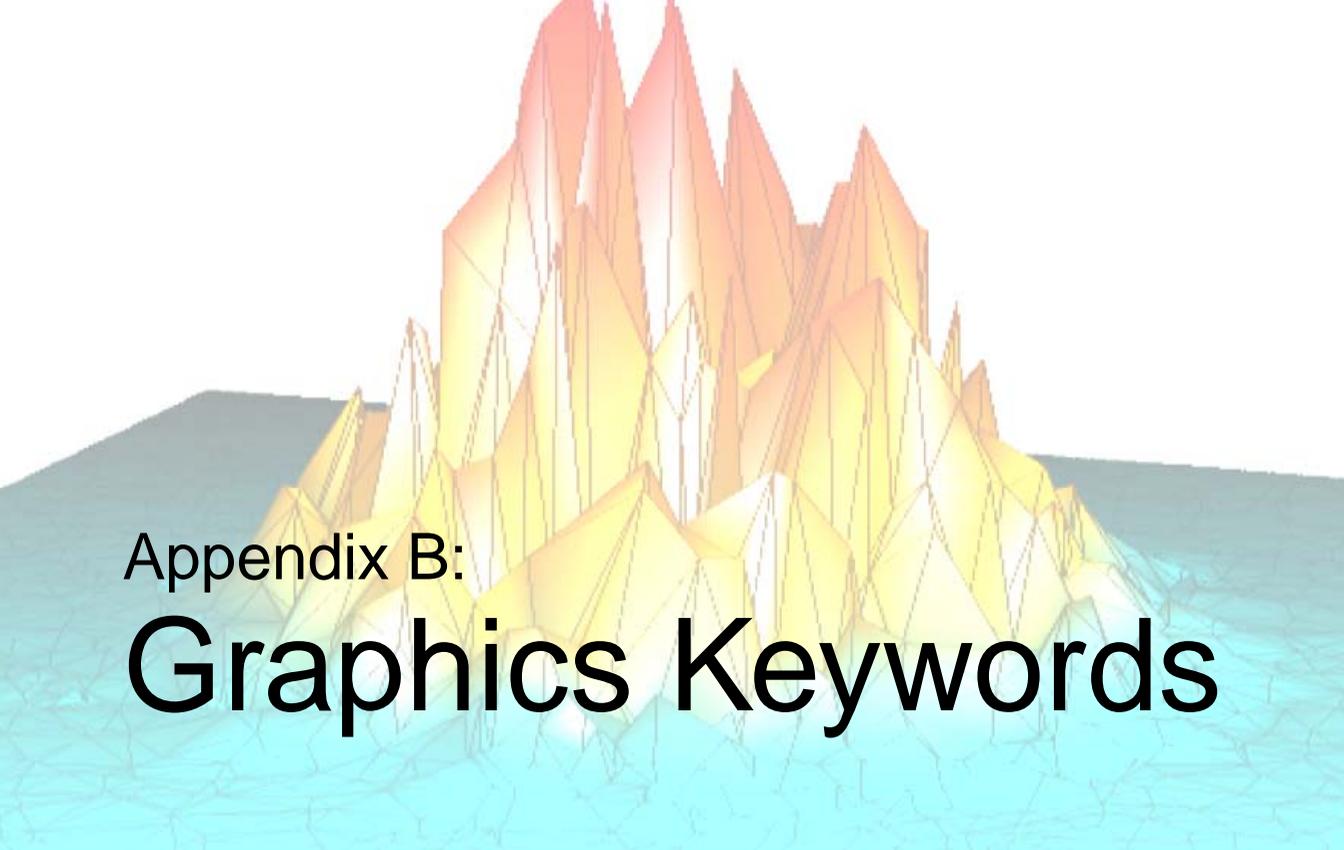


Figure A-8: Image Warped to a Cube Using the Z-Buffer



Appendix B:

Graphics Keywords

The IDL Direct Graphics routines, `CURSOR`, `ERASE`, `PLOTS`, `POLYFILL`, `TV` (and `TVSCL`), `TVCRS`, `TVRD`, and `XYOUTS`, and the plotting procedures, `AXIS`, `CONTOUR`, `PLOT`, `OPLOT`, `SHADE_SURF`, and `SURFACE`, accept a number of common keywords. Therefore, instead of describing each keyword along with the description of each routine, this section contains a brief summary of each graphics keyword. Routine-specific keywords are documented in the description of the routine. The graphics keywords are described below. The name of each keyword is followed by a list of routines that accept that keyword. Keywords that have a direct correspondence to fields in a system variable (usually `!P`) are also indicated.

The keywords that control the plot axes are prefixed with the character ‘X’, ‘Y’, or ‘Z’ depending on the axis in question. These keywords correspond to fields in the axis system variables: `!X`, `!Y`, and `!Z`, and are described in more detail in [“Graphics System Variables”](#) on page 4375. The axis keywords are shown in the form `[XYZ]NAME`. For example, `[XYZ]CHARSIZE` refers to the three keywords `XCHARSIZE`, `YCHARSIZE`, and `ZCHARSIZE`, which control the size of the characters annotating the three axes.

The system variable fields that control this are !X.CHAR.SIZE, !Y.CHAR.SIZE, and !Z.CHAR.SIZE.

The following graphics keywords are discussed in this appendix:

BACKGROUND	ORIENTATION	[XYZ]STYLE
CHANNEL	POSITION	[XYZ]THICK
CHARSIZE	PSYM	[XYZ]TICK_GET
CHARTHICK	SUBTITLE	[XYZ]TICKFORMAT
CLIP	SYMSIZE	[XYZ]TICKINTERVAL
COLOR	T3D	[XYZ]TICKLAYOUT
DATA	THICK	[XYZ]TICKLEN
DEVICE	TICKLEN	[XYZ]TICKNAME
FONT	TITLE	[XYZ]TICKS
LINestyle	[XYZ]CHARSIZE	[XYZ]TICKUNITS
NOCLIP	[XYZ]GRIDSTYLE	[XYZ]TICKV
NODATA	[XYZ]MARGIN	[XYZ]TITLE
NOERASE	[XYZ]MINOR	Z
NORMAL	[XYZ]RANGE	ZVALUE

BACKGROUND

Accepted by: **CONTOUR**, **PLOT**, **SURFACE**.

System variable equivalent: !P.**BACKGROUND**.

The background color index to which all pixels are set when erasing the screen or page. The default is 0 (black). Not all devices support erasing the background to a specified color index.

For example, to produce a black plot with a white background on a color display:

```
PLOT, Y, BACKGROUND = 255, COLOR = 0
```

Warning

If the **NOERASE** keyword (or !P.**NOERASE**) is set, the value of the **BACKGROUND** keyword (or !P.**BACKGROUND**) is ignored.

CHANNEL

Accepted by: [ERASE](#), [TV](#), [TVRD](#). System variable equivalent: `!P.CHANNEL`.

This keyword specifies the memory channel for the operation. This parameter is ignored on display systems that have only one memory channel. When using a “decomposed” display system, the red channel is 1, the green channel is 2, and the blue channel is 3. Channel 0 indicates all channels. If omitted, `!P.CHANNEL` contains the default channel value.

Note

`CONTOUR`, `PLOT`, `SHADE_SURF`, and `SURFACE` also accept the `CHANNEL` keyword, but simply pass it to `ERASE`.

CHARSIZE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#), [XYOUTS](#). System variable equivalent: `!P.CHARSIZE`.

The overall character size for the annotation when Hershey fonts are selected. This keyword does not apply when hardware (i.e. PostScript) fonts are selected. A `CHARSIZE` of 1.0 is normal. The size of the annotation on the axes may be set, relative to `CHARSIZE`, with `xCHARSIZE`, where *x* is X, Y, or Z. The main title is written with a character size of 1.25 times this parameter.

CHARTHICK

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#), [XYOUTS](#). System variable equivalent: `!P.CHARTHICK`.

An integer value specifying the line thickness of the vector drawn font characters. This keyword has no effect when used with the hardware drawn fonts. The default value is 1.

CLIP

Accepted by: [CONTOUR](#), [DRAW_ROI](#), [OPLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SURFACE](#), [XYOUTS](#). System variable equivalent: `!P.CLIP`.

The coordinates of a rectangle used to clip the graphics output. The rectangle is specified as a vector of the form $[X_0, Y_0, X_1, Y_1]$, giving coordinates of the lower left and upper right corners, respectively. The default clipping rectangle is the plot window, the area enclosed within the axes of the most recent plot. Coordinates are

specified in data units unless an overriding coordinate unit specification keyword is present (i.e., `NORMAL` or `DEVICE`). If the clipping is provided in data or normalized units, the actual clipping rectangle is computed by converting those values to device units. The clipping itself always occurs in device space.

Note

The default is not to clip the output of `PLOTS` and `XYOUTS`. To enable clipping include the keyword parameter `NOCLIP = 0`. With `PLOTS`, `POLYFILL`, and `XYOUTS`, this keyword controls the clipping of vectors and vector-drawn text.

For example, to draw a vector using normalized coordinates with its contents clipped within a rectangle covering the upper left quadrant of the display:

```
PLOTS, X, Y, CLIP=[0.,.5,.5,1.0], /NORM, NOCLIP=0
```

COLOR

Accepted by: [AXIS](#), [CONTOUR](#), [DRAW_ROI](#), [ERASE](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE_SURF](#), [SURFACE](#), [XYOUTS](#). System variable equivalent: `!P.COLOR`.

The color index of the data, text, line, or solid polygon fill to be drawn. If this keyword is omitted, `!P.COLOR` specifies the color index.

When used with the `PLOTS`, `POLYFILL`, or `XYOUTS` procedure, this keyword parameter can be set to a vector to specify multiple color indices.

Gouraud shading of polygons is performed with the Z-buffer graphics output device and `POLYFILL` procedure when `COLOR` contains an array of color indices, one for each vertex.

DATA

Accepted by: [AXIS](#), [CONTOUR](#), [CURSOR](#), [DRAW_ROI](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE_SURF](#), [SURFACE](#), [TV](#), [TVCRS](#), [XYOUTS](#).

Set this keyword to indicate that the clipping and/or positioning coordinates supplied are specified in the data coordinate system. The default coordinate system is `DATA` if no other coordinate-system specifications are present.

DEVICE

[AXIS](#), [CONTOUR](#), [CURSOR](#), [DRAW_ROI](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE_SURF](#), [SURFACE](#), [TV](#), [TVCRS](#), [XYOUTS](#).

Set this keyword to indicate that the clipping and/or positioning coordinates supplied are specified in the device coordinate system. The default coordinate system is DATA if no other coordinate-system specifications are present.

For example, the following code displays an image contained in the variable A and then draws a contour plot of pixels [100:499, 100:399] over the correct section of the image:

```
;Display the image.
TV,A

;Draw the contour plot, specify the coordinates of the plot, in
;device coordinates, do not erase, set the X and Y axis styles to
;EXACT.
CONTOUR, A[100:499, 100:399], $
      POS = [100,100, 499,399], /DEVICE, $
      /NOERASE, XSTYLE=1, YSTYLE=1
```

Note that in the above example, the keyword specification /DEVICE is equivalent to DEVICE = 1.

FONT

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#), [XYOUTS](#).
System variable equivalent: !P.FONT.

An integer that specifies the graphics text font system to use. Set FONT equal to -1 to select the Hershey character fonts, which are drawn using vectors. Set FONT equal to 0 (zero) to select the device font of the output device. Set FONT equal to 1 (one) to select the TrueType font system. See [Appendix H, “Fonts”](#) for a complete description of IDL’s font systems.

LINestyle

Accepted by: [DRAW_ROI](#), [OPLOT](#), [PLOT](#), [PLOTS](#), [SURFACE](#). System variable equivalent: !P.LINestyle.

This keyword indicates the line style used to draw lines; it indicates the line style of the lines used to connect the data points. This keyword should be set to the appropriate index for the desired linestyle as described in the following table.

Index	Linestyle
0	Solid
1	Dotted
2	Dashed
3	Dash Dot
4	Dash Dot Dot
5	Long Dashes

Table B-18: IDL Linestyles

NOCLIP

Accepted by: [CONTOUR](#), [DRAW_ROI](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SURFACE](#), [XYOUTS](#). System variable equivalent: `!P.NOCLIP`.

Set this keyword to suppress clipping of the plot. The clipping rectangle is contained in `!P.CLIP`. By default, the plot is clipped within the plotting window.

Note

The default value is clipping-disabled for `PLOTS`, `POLYFILL`, and `XYOUTS`. For all other routines, the default is to enable clipping.

With `PLOTS`, `POLYFILL`, and `XYOUTS`, this keyword controls the clipping of vectors and vector-drawn text. The default is to disable clipping, so to enable clipping include the parameter `NOCLIP = 0`. To explicitly disable clipping set this parameter to one.

NODATA

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#).

If this keyword is set, only the axes, titles, and annotation are drawn. No data points are plotted.

For example, to draw an empty set of axes between some given values:

```
PLOT, [XMIN, XMAX], [YMIN, YMAX], /NODATA
```

NOERASE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SURFACE](#). System variable equivalent: [!P.NOERASE](#).

Specifies that the screen or page is not to be erased. By default, the screen is erased, or a new page is begun, before a plot is produced.

Warning

If the NOERASE keyword (or [!P.NOERASE](#)) is set, the value of the [BACKGROUND](#) keyword (or [!P.BACKGROUND](#)) is ignored.

NORMAL

Accepted by: [AXIS](#), [CONTOUR](#), [CURSOR](#), [DRAW_ROI](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE_SURF](#), [SURFACE](#), [TV](#), [TVCRS](#), [XYOUTS](#).

Set this keyword to indicate that the clipping and/or positioning coordinates supplied are specified in the normalized coordinate system, and range from 0.0 to 1.0. The default coordinate system is DATA if no other coordinate-system specifications are present.

ORIENTATION

Accepted by: [DRAW_ROI](#), [POLYFILL](#), [XYOUTS](#).

Specifies the counterclockwise angle in degrees from horizontal of the text baseline and the lines used to fill polygons. When used with the POLYFILL procedure, this keyword forces the “linestyle” type of fill, rather than solid or patterned fill.

POSITION

Accepted by: [CONTOUR](#), [MAP_SET](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: [!P.POSITION](#).

Allows direct specification of the plot window. POSITION is a 4-element vector giving, in order, the coordinates $[(X_0, Y_0), (X_1, Y_1)]$, of the lower left and upper right corners of the data window. Coordinates are expressed in normalized units ranging from 0.0 to 1.0, unless the DEVICE keyword is present, in which case they are in

actual device units. The value of POSITION is never specified in data units, even if the DATA keyword is present.

When setting the position of the window, be sure to allow space for the annotation, which resides outside the window. IDL outputs the message “% Warning: Plot truncated.” if the plot region is larger than the screen or page size. The plot region is the rectangle enclosing the plot window and the annotation.

When plotting in three dimensions, the POSITION keyword is a 6-element vector with the first four elements describing, as above, the XY position, and with the last two elements giving the minimum and maximum Z coordinates. The Z specification is always in normalized coordinate units.

When making more than one plot per page it is more convenient to set !P.MULTI than to manipulate the position of the plot directly with the POSITION keyword.

For example, the following statement produces a contour plot with data plotted in only the upper left quarter of the screen:

```
CONTOUR, Z, POS=[0., 0.5, 0.5, 1.0]
```

Because no space on the left or top edges was allowed for the axes or their annotation, the above described warning message results.

PSYM

Accepted by: [DRAW_ROI](#), [OPLOTT](#), [PLOT](#), [PLOTS](#). System variable equivalent: !P.PSYM.

The symbol used to mark each data point. Normally, PSYM is 0, data points are connected by lines, and no symbols are drawn to mark the points. Set this keyword, or the system variable !P.PSYM, to the symbol index as shown in the table below to mark data points with symbols. The keyword SYMSIZE is used to set the size of the symbols.

PSYM Value	Plotting Symbol
1	Plus sign (+)
2	Asterisk (*)
3	Period (.)
4	Diamond

Table B-19: Values for the PSYM Keyword

PSYM Value	Plotting Symbol
5	Triangle
6	Square
7	X
8	User-defined. See USERSYM procedure.
9	Undefined
10	Histogram mode. Horizontal and vertical lines connect the plotted points, as opposed to the normal method of connecting points with straight lines.

Table B-19: Values for the PSYM Keyword

Negative values of PSYM cause the symbol designated by PSYM to be plotted at each point with solid lines connecting the symbols. For example, a value of -5 plots triangles at each data point and connects the points with lines. Histogram mode is the exception to this rule; since the points are already connected when PSYM=10, specifying the value -10 is meaningless, and will result in an error.

The following IDL code plots an array using points, and then overplots the smoothed array, connecting the points with lines:

```
;Plot using points.
PLOT, A, PSYM=3

;Overplot smoothed data.
OPLOT, SMOOTH(A,7)
```

SUBTITLE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `!P.SUBTITLE`.

A text string to be used as a subtitle for the plot. Subtitles appear below the X axis.

SYMSIZE

Accepted by: [DRAW_ROI](#), [OPLOT](#), [PLOT](#), [PLOTS](#).

Specifies the size of the symbols drawn when PSYM is set. The default size of 1.0 produces symbols approximately the same size as a character.

T3D

Accepted by: [AXIS](#), [CONTOUR](#), [DRAW_ROI](#), [MAP_SET](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE_SURF](#), [SURFACE](#), [TV](#), [TVCRS](#), [XYOUTS](#). System variable equivalent: !P.T3D.

Set this keyword to indicate that the generalized transformation matrix in !P.T is to be used. If not present, the user-supplied coordinates are simply scaled to screen coordinates. See the examples in the description of the SAVE keyword.

Note

Since T3D uses the transformation matrix in !P.T, it is important that !P.T contain a valid transformation matrix. This can be achieved in several ways:

- Use the SAVE keyword to save the transformation matrix from an earlier graphics operation.
- Establish a transformation matrix using the T3D, SURFR, or, SCALE3 procedures.
- Set the value of !P.T directly.

THICK

Accepted by: [AXIS](#), [DRAW_ROI](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: !P.THICK.

Indicates the line thickness. THICK overrides the setting of !P.THICK.

TICKLEN

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: !P.TICKLEN.

Controls the length of the axis tick marks, expressed as a fraction of the window size. The default value is 0.02. TICKLEN of 1.0 produces a grid, while a negative TICKLEN makes tick marks that extend outside the window, rather than inwards.

For example, to produce outward-going tick marks of the normal length:

```
PLOT, X, Y, TICKLEN = -0.02
```

To provide a new default tick length, set !P.TICKLEN.

TITLE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `!P.TITLE`.

Produces a main title centered above the plot window. The text size of this main title is larger than the other text by a factor of 1.25. For example:

```
PLOT, X, Y, TITLE = 'Final Results'
```

[XYZ]CHARSIZE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalents: `![XYZ].CHARSIZE`.

The size of the characters used to annotate the axis and its title when Hershey fonts are selected. This keyword does not apply when hardware (i.e. PostScript) fonts are selected. This field is a scale factor applied to the global scale factor set by `!P.CHARSIZE` or the keyword `CHARSIZE`.

[XYZ]GRIDSTYLE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#)

The index of the linestyle to be used for plot tickmarks and grids (i.e., when `[XYZ]TICKLEN` is set to 1.0). See [LINESTYLE](#) for a list of linestyles.

[XYZ]MARGIN

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `![XYZ].MARGIN`.

A 2-element array specifying the margin on the left (bottom) and right (top) sides of the plot window, in units of character size. Default margins are 10 and 3 for the X axis, and 4 and 2 for the Y axis. The `ZMARGIN` keyword is present for consistency and is currently ignored.

[XYZ]MINOR

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `![XYZ].MINOR`.

The number of minor tick mark intervals.

[XYZ]RANGE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `![XYZ].RANGE`.

The desired data range of the axis, a 2-element vector. The first element is the axis minimum, and the second is the maximum. IDL will frequently round this range. This override can be defeated using the `[XYZ]STYLE` keywords.

[XYZ]STYLE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `![XYZ].STYLE`.

This keyword allows specification of axis options such as rounding of tick values and selection of a box axis. Each option is described in the following table:

Value	Description
1	Force exact axis range.
2	Extend axis range.
4	Suppress entire axis
8	Suppress box style axis (i.e., draw axis on only one side of plot)
16	Inhibit setting the Y axis minimum value to 0 (Y axis only)

Table B-20: Values for the [XYZ]STYLE Keyword

Note that this keyword is set bitwise, so multiple effects can be set by adding values together. For example, to make an X axis that is both exact (value 1) and suppresses the box style (setting 8), set the XAXIS keyword to 1+8, or 9.

[XYZ]THICK

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `![XYZ].THICK`.

This keyword controls the thickness of the lines forming the axis and tick marks. A value of 1.0 is the default.

[XYZ]TICK_GET

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#).

A named variable in which to return the values of the tick marks for the designated axis. The result is a double precision floating-point array with the same number of elements as ticks.

For example, to retrieve in the variable V the values of the tick marks selected by IDL for the Y axis:

```
PLOT, X, Y, YTICK_GET = V
```

[XYZ]TICKFORMAT

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: `![XYZ].TICKFORMAT`.

Set this keyword to a string or a vector of strings. If a vector is provided, each string corresponds to a level of the axis. The [\[XYZ\]TICKUNITS](#) keyword determines the number of levels for an axis.

Each string is one of the following:

A format code:

If the string begins with an open parenthesis, it is treated as a standard format string. See “[Format Codes](#)” in Chapter 11 of the *Building IDL Applications* manual for more information on format codes.

Example 1: Display the X axis tick values using a format of F6.2 (six characters, with 2 places after the decimal point):

```
PLOT, X, Y, XTICKFORMAT='(F6.2)'
```

Example 2: Display the Y tick values using the “dollars and cents” format `$dddd.dd`:

```
PLOT, X, Y, YTICKFORMAT='("$", F7.2)'
```

The string 'LABEL_DATE' :

Set `[XYZ]TICKFORMAT` to the string 'LABEL_DATE' to create axes with date labels. The formatting of the labels is specified by first calling `LABEL_DATE` with the `DATE_FORMAT` keyword. See [LABEL_DATE](#) for more information.

Example: Use the `LABEL_DATE` function as the callback function to display the X tick values in a date/time format:

```
dummy = LABEL_DATE(DATE_FORMAT='%M %Z')
```

```
mytimes = TIMEGEN(12, UNITS='MONTHS', START=JULDAY(1,1,2000))
y = FINDGEN(12)
PLOT, mytimes, y, XTICKUNITS='Time', XTICKFORMAT='LABEL_DATE'
```

The name of a user-defined function:

If the string does not begin with an open parenthesis, it is interpreted as the name of a callback function to be used to generate tick mark labels. This function is defined with either three or four parameters, depending on whether [XYZ]TICKUNITS is specified:

If [XYZ]TICKUNITS is *not* specified, the callback function is called with three parameters, *Axis*, *Index*, and *Value*, where:

- *Axis* is the axis number: 0 for X axis, 1 for Y axis, 2 for Z axis.
- *Index* is the tick mark index (indices start at 0).
- *Value* is the data value at the tick mark (a double-precision floating point value).

Note

Value is a double-precision floating-point value that represents the Julian date. The Julian date follows the astronomical convention, where Julian date 0.0d corresponds to 1 Jan 4713 B.C.E. at 12 pm.

If [XYZ]TICKUNITS is specified, the callback function is called with four parameters, *Axis*, *Index*, *Value*, and *Level*, where:

- *Axis*, *Index*, and *Value* are the same as described above.
- *Level* is the index of the axis level for the current tick value to be labeled (level indices start at 0).

Example 1: Use a callback function to display the Y tick values as a percentage of a fixed value. Note that because we don't specify [XYZ]TICKUNITS, we do not include the *Level* parameter in our function definition:

```
FUNCTION YTICKS, axis, index, value
    fixvalue = 389.0d
    pvalue = (value/fixvalue) * 100.0d
    RETURN, STRING(pvalue, FORMAT='(D5.2,"%")')
END

PRO use_callback
    Y = FINDGEN(10)
    PLOT, Y, YTICKFORMAT='YTICKS'
END
```

Example 2: Create a two-level X axis. Display the X tick values in a customized date/time format that shows the number of days open for business for each month on one level, and marks leap years with an asterisk on another level:

```

FUNCTION XTICKS, axis, index, value, level

CASE level OF
  0: BEGIN ; months
      ; Number of days open for business in given month:
      CALDAT, value, month
      open = [18,19,23,20,22,22,19,10,20,21,22,14]
      nbdays = open[month]
      ; Return a string containing the month name plus
      ; the number of business days in parentheses:
      RETURN, STRING(value, nbdays, $
          FORMAT='(C(CMoA), "(, I2, ")")')
  END
  1: BEGIN ; years
      ; Generate a string for the year.
      yrStr = STRING(value, FORMAT='(C(CYI))')
      ; Determine if a leap year. If so,
      ; append an asterisk to the string.
      CALDAT, value, mo, da, yr
      IF (yr MOD 4 EQ 0) THEN BEGIN
          IF (yr MOD 100 EQ 0) THEN $
              isLeap = (yr MOD 400) EQ 0 $
          ELSE $
              isLeap = 1b
          ENDIF ELSE $
              isLeap = 0b
          IF (isLeap NE 0b) THEN $
              yrStr = yrStr + '*'
          RETURN, yrStr
      END
  ENDCASE

END

PRO plot_sales

myDates = TIMEGEN(12, UNITS='Months', START=JULDAY(1,1,2000))
sales = [180,190,230,200,220,220,190,100,200,210,220,140]
PLOT, myDates, sales, XTICKUNITS=['Months', 'Years'], $
    XTICKFORMAT='XTICKS', XTITLE = 'Date (* = Leap Year)', $
    YTITLE='Sales (units)', POSITION = [0.2, 0.2, 0.9, 0.9]

END

```

[XYZ]TICKINTERVAL

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#)

variable equivalent: ![XYZ].[TICKINTERVAL](#)

Set this keyword to a scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis range ([XYZ]RANGE) and the number of major tick intervals ([XYZ]TICKS). This keyword takes precedence over [XYZ]TICKS.

For example, if TICKUNITS=["Seconds", "Hours", "Days"], and XTICKINTERVAL=30, then the interval between major ticks for the first axis level will be 30 seconds.

[XYZ]TICKLAYOUT

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].[TICKLAYOUT](#).

Set this keyword to a scalar that indicates the tick layout style to be used to draw each level of the axis.

Valid values include:

- 0 = The axis line, major tick marks and tick labels are all included. Minor tick marks only appear on the first level of the axis. This is the default tick layout style.
- 1 = Only the labels for the major tick marks are drawn. The axis line, major tick marks, and minor tick marks are omitted.
- 2 = Each major tick interval is outlined by a box. The tick labels are positioned within that box (left-aligned). For the first axis level only, the major and minor tick marks will also be drawn.

Note

For all tick layout styles, at least one tick label will appear on each level of the axis (even if no major tick marks fall along the axis line). If there are no major tick marks, the single tick label will be centered along the axis.

[XYZ]TICKLEN

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].[TICKLEN](#).

This keyword controls the lengths of tick marks (expressed in normal coordinates) for the individual axes. This keyword, if nonzero, overrides the global tick length specified in !P.TICKLEN, and/or the TICKLEN keyword parameter, which is expressed in terms of the window size.

[XYZ]TICKNAME

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].TICKNAME.

A string array of up to 30 elements that controls the annotation of each tick mark.

[XYZ]TICKS

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].TICKS.

The number of major tick *intervals* to draw for the axis. If this keyword is omitted, IDL selects from three to six tick intervals. Setting this field to n , where $n > 1$, produces exactly n tick intervals, and $n+1$ tick marks. Setting this field equal to 1 suppresses tick marks.

[XYZ]TICKUNITS

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].TICKUNITS.

Set this keyword to a string or a vector of strings indicating the units to be used for axis tick labeling. If a vector of strings is provided, the axis will be drawn in multiple levels, where each string represents one level in the specified units.

Note

When creating multiple-level axes, you may need to adjust the plot positioning using the [POSITION](#) or [\[XYZ\]MARGIN](#) keywords in order to ensure that axis labels and titles are visible in the plot window.

The order in which the strings appear in the vector determines the order in which the corresponding unit levels will be drawn. The first string corresponds to the first level (the level nearest to the primary axis line).

Valid unit strings include:

- 'Numeric'
- 'Years'

- 'Months'
- 'Days'
- 'Hours'
- 'Minutes'
- 'Seconds'
- 'Time' - Use this value to indicate that the tick values are time values; IDL will determine the appropriate time intervals and tick label formats based upon the range of values covered by the axis.
- '' - Use the empty string to indicate that no tick units are being explicitly set. This implies that a single axis level will be drawn using the 'Numeric' unit. This is the default setting.

If any of the time units are utilized, the tick values are interpreted as Julian date/time values.

Note that the singular form of each of the time value strings is also acceptable (e.g, TICKUNITS='Day' is equivalent to TICKUNITS='Days').

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000, respectively.

[XYZ]TICKV

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].[TICKV](#).

The data values for each tick mark, an array of up to 60 elements.

Note

To specify the number of ticks and their values exactly, set [XYZ]TICKS= N (where $N > 1$) and [XYZ]TICKV=*Values*, where *Values* has $N+1$ elements.

[XYZ]TITLE

Accepted by: [AXIS](#), [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#). System variable equivalent: ![XYZ].[TITLE](#).

A string that contains a title for the specified axis.

Z

Accepted by: [PLOTS](#), [POLYFILL](#), [TV](#), [TVCRS](#), [XYOUTS](#).

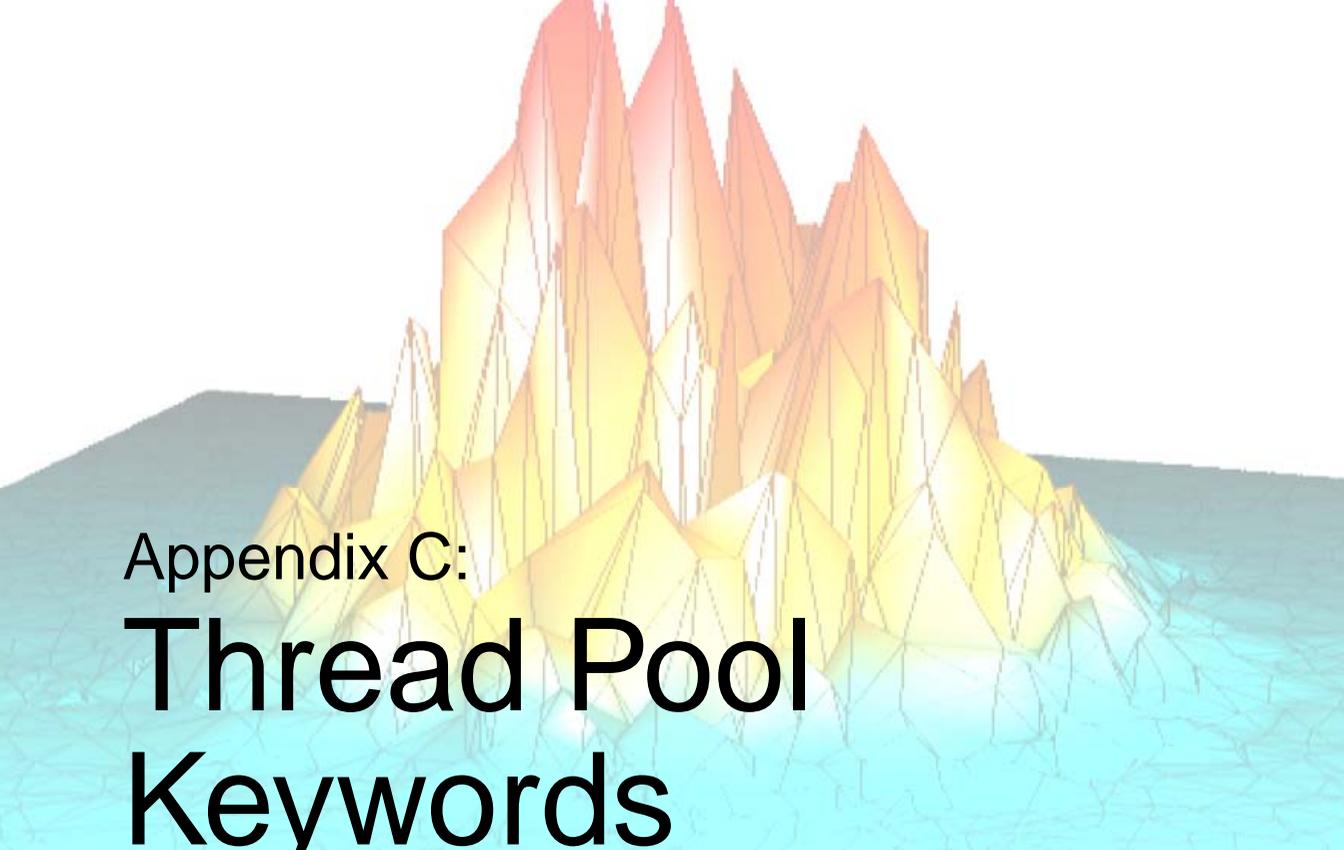
Provides the Z coordinate if a Z parameter is not present in the call. This is of use only if the three-dimensional transformation is in effect (i.e., the T3D keyword is set).

ZVALUE

Accepted by: [AXIS](#), [CONTOUR](#), [MAP_SET](#), [OPLOT](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#).

Sets the Z coordinate, in normalized coordinates in the range of 0 to 1, of the axis and data output from [PLOT](#), [OPLOT](#), and [CONTOUR](#).

This keyword has effect only if `!P.T3D` is set and the three-dimensional to two-dimensional transformation is stored in `!P.T`. If `ZVALUE` is not specified, [CONTOUR](#) will output each contour at its Z coordinate, and the axes and title at a Z coordinate of 0.0.



Appendix C:

Thread Pool Keywords

Many of the system routines documented in this manual make use of the IDL thread pool. The thread pool is discussed in detail in [Chapter 15, “Multithreading in IDL”](#) in the *Building IDL Applications* manual. System-wide use of the thread pool can be controlled with the [CPU](#) procedure, and the current system settings are visible via the [!CPU](#) system variable.

All system routines that use the thread pool accept the following keywords, which can be used to modify the default behavior for the duration of a single call. This allows you to modify the settings for a particular computation without affecting the global default settings of the IDL session.

Thread Pool Keywords

TPOOL_MAXELTS

Set this keyword to a non-zero value to set the maximum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation exceeds the number you specify, IDL will not use the thread pool for the computation. Setting this value to 0 removes any limit on the maximum number of elements, and any computation with at least TPOOL_MINELTS will use the thread pool.

This keyword overrides the default value, given by !CPU.TPOOL_MAXELTS. See [“Possible Drawbacks to the Use of the IDL Thread Pool”](#) in Chapter 15 of the *Building IDL Applications* manual for discussion of the circumstances under which it may be useful to specify a maximum number of elements.

TPOOL_MINELTS

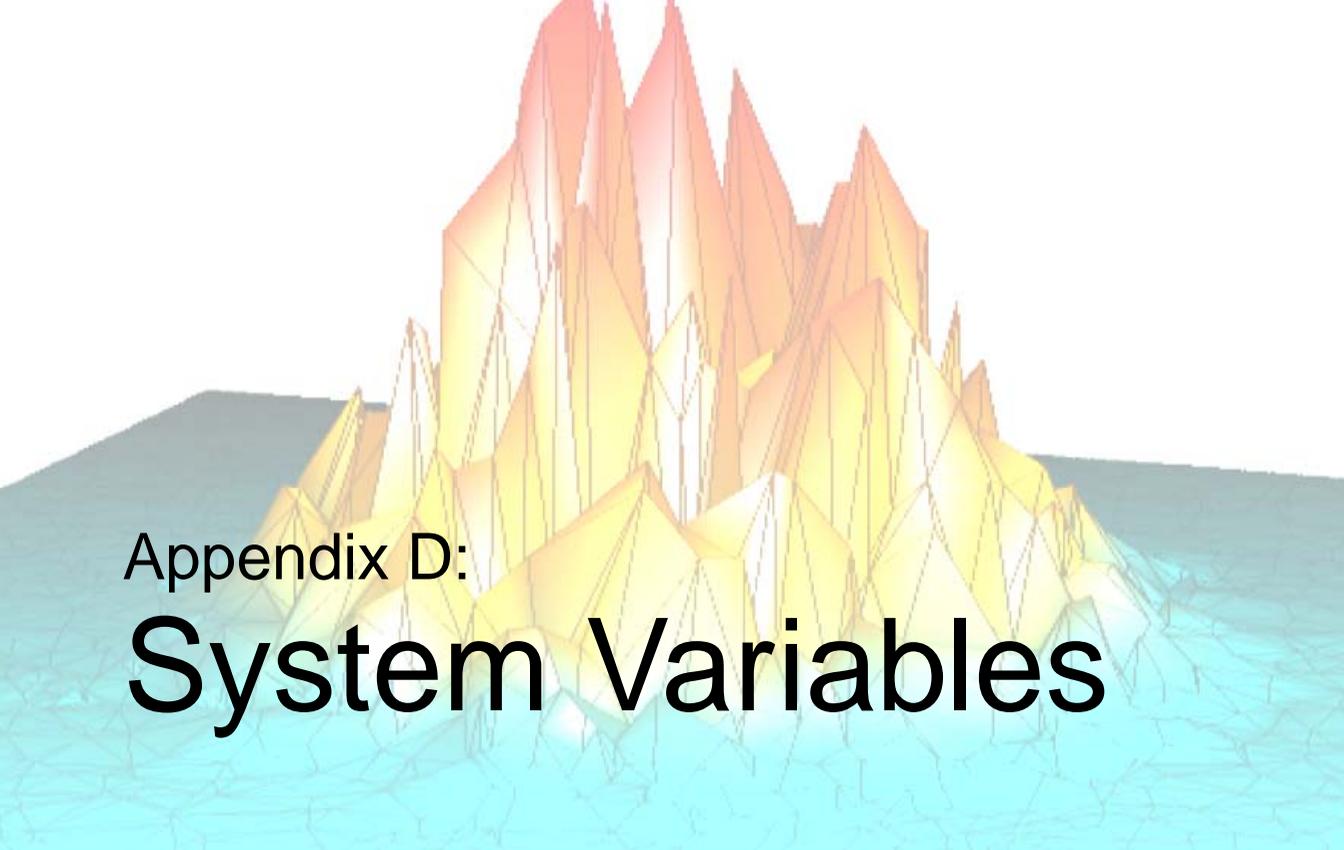
Set this keyword to a non-zero value to set the minimum number of data elements involved in a computation that uses the thread pool. If the number of elements in the computation is less than the number you specify, IDL will not use the thread pool for the computation. Use this keyword to prevent IDL from using the thread pool on tasks that are too small to benefit from it.

This keyword overrides the default value, given by !CPU.TPOOL_MINELTS. See [“Possible Drawbacks to the Use of the IDL Thread Pool”](#) in Chapter 15 of the *Building IDL Applications* manual for discussion of the circumstances under which it may be useful to specify a minimum number of elements.

TPOOL_NOTHREAD

Set this keyword to explicitly prevent IDL from using the thread pool for the current computation. If this keyword is set, IDL will use the non-threaded implementation of the routine even if the current settings of the !CPU system variable would allow use of the threaded implementation.

This keyword overrides the default value, given by !CPU.TPOOL_NTHREADS. See [“Possible Drawbacks to the Use of the IDL Thread Pool”](#) in Chapter 15 of the *Building IDL Applications* manual for discussion of the circumstances under which it may be useful to disable use of the thread pool entirely.



Appendix D: System Variables

The following topics are included in this appendix:

What Are System Variables?	4356	IDL Environment System Variables	4363
Constant System Variables	4357	Graphics System Variables	4375
Error Handling System Variables	4359		

What Are System Variables?

System variables are a special class of predefined variables available to all program units. Their names always begin with the exclamation mark character (!). System variables are used to set the options for plotting, to set various internal modes, to return error status, etc.

System variables have a predefined type and structure that cannot be changed. When an expression is stored into a system variable, it is converted to the variable type, if necessary and possible. Certain system variables are *read only*, and their values cannot be changed. The user can define new system variables with the DEFSYSV procedure.

Constant System Variables

The following system variables contain pre-defined constants or values for use by IDL routines. System variables can be used just like other variables. For example, the command:

```
PRINT, ACOS(A) * !RADEG
```

converts a result expressed in radians to one expressed in degrees.

!DPI

A read-only variable containing the double-precision value of pi (π).

!DTOR

A read-only variable containing the floating-point value used to convert degrees to radians ($\pi/180 \cong 0.01745$).

!MAP

An array variable containing the information needed to effect coordinate conversions between points of latitude and longitude and map coordinates. The values in this array are established by the MAP_SET procedure; the user should not change them directly.

!PI

A read-only variable containing the single-precision value of pi (π).

!RADEG

A read-only variable containing the floating-point value used to convert radians to degrees ($180/\pi \cong 57.2958$).

!VALUES

A read-only variable containing the IEEE single- and double-precision floating-point values *Infinity* and *NaN* (Not A Number). !VALUES is a structure variable with the following fields:

```
** Structure !VALUES, 4 tags, length=24:  
F_INFINITY      FLOAT      Infinity  
F_NAN           FLOAT      NaN  
D_INFINITY      DOUBLE     Infinity  
D_NAN           DOUBLE     NaN
```

where *Infinity* is the value Infinity and *NaN* is the value Not A Number. For more information on these special floating-point values, see [“Special Floating-Point Values”](#) in Chapter 19 of the *Building IDL Applications* manual.

Error Handling System Variables

The following system variables are either set by IDL when an error condition occurs or used by IDL when displaying information about errors.

!ERR

This system variable is now obsolete and has been replaced by the [!ERROR_STATE](#) system variable. Code that uses the `!ERR` system variable will continue to function as before, but all new code should use `!ERROR_STATE.CODE`.

!ERROR_STATE

A structure variable which contains the status of the last error message.

`!ERROR_STATE` includes the following fields:

```
** Structure !ERROR_STATE, 8 tags, length=112, data length=108:
  NAME           STRING      'IDL_M_SUCCESS'
  BLOCK          STRING      'IDL_MBLK_CORE'
  CODE           LONG         0
  SYS_CODE       LONG         Array[2]
  SYS_CODE_TYPE  STRING      ''
  MSG            STRING      ''
  SYS_MSG        STRING      ''
  MSG_PREFIX     STRING      '% '
```

- **NAME:** A read-only string variable containing the error name of the IDL-generated component of the last error message.
- **BLOCK:** A read-only string variable containing the name of the message block for the last error message's IDL-generated component.

Note

See the *External Development Guide* for more information about blocks.

- **CODE:** A long-integer variable containing the error code of the last error's IDL-generated component.
- **SYS_CODE:** A long-integer variable containing the error code of the last error's operating system-generated component, if it exists.

For historical reasons, `SYS_CODE` is a two-element longword array. The first element of the array (that is, `SYS_CODE[0]`) contains the OS-defined error code. The second element of the array is not used, and always contains zero.

Either `!ERROR_STATE.SYS_CODE` or `!ERROR_STATE.SYS_CODE[0]` will return the relevant error code.

- `SYS_CODE_TYPE`: A string describing the type of system code contained in `SYS_CODE`. A null string in this field indicates that there is no system code corresponding to the current error. The possible non-NULL values are:

Value	Meaning
errno	Unix/Posix system error
win32	Microsoft Windows Win32 system error
winsock	Microsoft Windows sockets library error

Table D-21: SYS_COD_TYPE Values

- `MSG`: A read-only string variable containing the text of the last IDL-generated error message.
- `SYS_MSG`: A read-only string variable containing the text of the last error's operating system-generated component, if it exists.
- `MSG_PREFIX`: A string variable containing the prefix string used for error messages.

This system variable replaces `!ERROR`, `!ERR_STRING`, `!MSG_PREFIX`, `!SYSERR_STRING`, and `!SYSERROR`, and includes two new fields: error name and block name. For a more detailed explanation of `!ERROR_STATE`, see “[Error Handling](#)” in Chapter 19 of the *Building IDL Applications* manual.

!ERROR

This system variable is now obsolete and has been replaced by the `!ERROR_STATE` system variable. Code that uses the `!ERROR` system variable will continue to function as before, but we suggest that all new code use `!ERROR_STATE.CODE`.

!ERR_STRING

This system variable is now obsolete and has been replaced by the `!ERROR_STATE` system variable. Code that uses the `!ERR_STRING` system variable will continue to function as before, but we suggest that all new code use `!ERROR_STATE.MSG`.

!EXCEPT

An integer variable that controls when IDL checks for invalid mathematical computations (exceptions), such as division by zero. The three allowed values are:

Value	Description
0	Never report exceptions.
1	Report exceptions when the interpreter is returning to an interactive prompt (the default).
2	Report exceptions at the end of each IDL statement. Note that this slows IDL by roughly 5% compared to setting !EXCEPT=1.

Table D-22: EXCEPT Values

For more information on invalid mathematical computations and error reporting, see “[Math Errors](#)” in Chapter 19 of the *Building IDL Applications* manual.

The value of !EXCEPT is used by the CHECK_MATH function to determine when to return errors. See “[CHECK_MATH](#)” on page 245 for details.

Note

In versions of IDL up to and including IDL 4.0.1, the default exception handling was functionally identical to setting !EXCEPT=2.

!MOUSE

A structure variable that contains the status from the last cursor read operation. !MOUSE has the following fields:

```
** Structure !MOUSE, 4 tags, length=16:
X                LONG                511
Y                LONG                252
BUTTON           LONG                4
TIME             LONG                1428829775
```

- X and Y: Contain the location (in device coordinates) of the cursor when the mouse button was pressed.
- BUTTON: Contains
 - 1 (one) if the left mouse button was pressed,

- 2 (two) if the middle mouse button was pressed
- 4 (four) if the right mouse button was pressed.
- TIME: Contains the number of milliseconds since a base time.

See “[CURSOR](#)” on page 371 for details on reading the cursor position.

!MSG_PREFIX

This keyword is now obsolete and has been replaced by the [!ERROR_STATE](#) system variable. Code that uses the [!MSG_PREFIX](#) system variable will continue to function as before, but we suggest that all new code use [!ERROR_STATE.MSG_PREFIX](#).

!SYSERROR

This keyword is now obsolete and has been replaced by the [!ERROR_STATE](#) system variable. Code that uses the [!SYSERROR](#) system variable will continue to function as before, but we suggest that all new code use [!ERROR_STATE.SYS_CODE](#).

!SYSERR_STRING

This keyword is now obsolete and has been replaced by the [!ERROR_STATE](#) system variable. Code that uses the [!SYSERR_STRING](#) system variable will continue to function as before, but we suggest that all new code use [!ERROR_STATE.SYS_MSG](#).

!WARN

A structure variable that causes IDL to print warnings to the console or command log when obsolete IDL features are found at compile time. !WARN has the following fields:

```
** Structure !WARN, 3 tags, length=3:
OBS_ROUTINES      BYTE      0
OBS_SYSVARS       BYTE      0
PARENS            BYTE      0
```

Setting each of the three fields to 1 (one) generates a warning for a different type of obsolete code. If the OBS_ROUTINES field is set equal to one, IDL generates warnings when it encounters references to obsolete internal or library routines. If the OBS_SYSVARS field is set equal to one, IDL generates warnings when it encounters references to obsolete system variables. If the PARENS field is set equal to one, IDL generates warnings when it encounters a use of parentheses to specify an index into an array. No warnings are generated when the fields of the !WARN structure are set equal to zero (the default).

IDL Environment System Variables

The following system variables contain information about IDL's configuration.

!CPU

IDL can use multiple system processors to perform some computations in parallel. See [Chapter 15, "Multithreading in IDL"](#) in the *Building IDL Applications* manual for additional information.

The !CPU system variable supplies information about the state of the system processor, and of IDL's use of it. !CPU is readonly, and cannot be modified directly. Use the CPU procedure to modify values contained in !CPU.

The !CPU structure is defined as follows:

```
{ !CPU, HW_VECTOR:0L, VECTOR_ENABLE:0L, HW_NCPU:0L,
  TPOOL_NTHREADS:0L, TPOOL_MIN_ELTS:0L, TPOOL_MAX_ELTS:0L }
```

The meaning of the fields of !CPU are given in the following table.

Field	Meaning
HW_VECTOR	True (1) if the system supports a vector unit (e.g. Macintosh AltiVec/Velocity Engine). False (0) otherwise. Note - This value is currently always 0 (False) on platforms other than Macintosh.
VECTOR_ENABLE	True (1) if IDL will use a vector unit, if such a unit is available on the current system, and False (0) otherwise. Note - This value is currently always 0 (False) on platforms other than Macintosh.
HW_NCPU	The number of CPUs contained in the system on which IDL is currently running.

Table D-23: Meaning of !CPU fields

Field	Meaning
TPOOL_NTHREADS	<p>The number of threads that IDL will use in thread pool computations. The default is to use HW_NCPU threads, so that each thread will have the potential to run in parallel with the others. For numerical computation, there is no benefit to using more threads than your system has CPUs. However, depending on the size of the problem and the number of other programs running on the system, there may be a performance advantage to using fewer CPUs.</p> <p>If the IDL_CPU_TPOOL_NTHREADS environment variable has been set, then TPOOL_NTHREADS is initialized with this value at startup. On systems shared by multiple users, you may wish to set this environment variable so that IDL uses the specified number of threads instead of defaulting to the number of CPUs present in the underlying hardware. See “IDL_CPU_TPOOL_NTHREADS” in Chapter 1 of the <i>Using IDL</i> manual for more information.</p>
TPOOL_MIN_ELTS	<p>The number of elements in a computation that are necessary before IDL will use the thread pool to perform the work. For fewer than TPOOL_MIN_ELTS, the main IDL thread will simply perform the work without using the thread pool. It is important not to use the thread pool for small tasks because the overhead of threading will not be offset by the overhead incurred by operation of the pool, and the overall computation will go slower than if threading is not used.</p>

Table D-23: Meaning of !CPU fields (Continued)

Field	Meaning
TPOOL_MAX_ELTS	The maximum number of elements in a computation for which IDL will use the thread pool. If this value is 0 (zero) (the default), then no limit is imposed and any computation with at least TPOOL_MIN_ELTS is a candidate for the thread pool. If your computation is too large for the physical memory available on the system, the virtual memory system of the operating system will begin paging. Under such conditions, the performance of the thread pool can be worse than that of a single threaded computation because the threads end up fighting each other for access to memory. TPOOL_MAX_ELTS can be used to prevent this.

Table D-23: Meaning of !CPU fields (Continued)

!DIR

A string variable containing the path to the main IDL directory.

!DLM_PATH

Significant portions of IDL's built in functionality are packaged in the form of Dynamically Loadable Modules (DLMs). DLMs correspond to UNIX sharable libraries or Windows DLLs, depending on the operating system in use. At startup, IDL searches for DLM definition files (which end in the `.d1m` suffix) and makes note of the routines supplied by each DLM. If such a routine is called, IDL loads the DLM that supplies it into memory. To see a list of the DLMs that IDL knows about, use `HELP, /DLM` (see "[HELP](#)" on page 859 for more information).

!DLM_PATH is initialized from the environment variable `IDL_DLM_PATH` at startup. This initialization is similar to that performed for `IDL_PATH`, (see "[!PATH](#)" on page 4371), including recursive path expansion denoted with a leading "+" but *not* including any use of IDLDE preferences settings. See "[Environment Variables Used by IDL](#)" in Chapter 1 of the *Using IDL* manual for details on setting the `IDL_DLM_PATH` environment variable.

If the `IDL_DLM_PATH` environment variable is not defined, IDL supplies a default that contains the directory in the IDL distribution where the DLMs supplied by RSI reside. Once !DLM_PATH is expanded, IDL uses it as the list of places to look for DLM definition files.

Since all DLM searching happens once at startup time, it would be meaningless to change the value of `!DLM_PATH` afterwards. For this reason, it is a read-only system variable and cannot be assigned to. The value of `!DLM_PATH` is useful because it shows you where IDL looked for DLMs when it started.

Using Path Definition Tokens to Load a DLM Path

Note that using the `<IDL_BIN_DIRNAME>` token in the `IDL_DLM_PATH` environment variable can be useful for distributing packages of DLMs with support for multiple operating system and hardware combinations. This token is described in “[The Path Definition String](#)” under [EXPAND_PATH](#).

For example, assume that you have your DLMs installed in `/usr/local/myd1m`, with support for each platform in a subdirectory using the same naming convention that IDL uses for the platform dependant subdirectories underneath the bin directory of the IDL distribution. The following line, which might be located in a file executed by a UNIX shell when you log in (`.cshrc`, `.login`), will add the location of the proper DLM for your current system to IDL's `!DLM_PATH` at startup:

```
% setenv IDL_DLM_PATH
  "/usr/local/myd1m/<IDL_BIN_DIRNAME>:<IDL_DEFAULT>"
```

(all on a single line). Setting the Windows environment variable `IDL_DLM_PATH` to a similar string would produce the same result on a Windows system.

Similarly, the `<IDL_VERSION_DIRNAME>` token can be useful for distributing packages of DLMs with support for multiple IDL versions, operating systems, and hardware platforms. This token is described in “[The Path Definition String](#)” under [EXPAND_PATH](#).

For example, assume that you have your DLMs installed in `/usr/local/myd1m`. Within the `myd1m` subdirectory would be a directory for each supported version of IDL. Within each of those subdirectories would be a subdirectory for each operating system and hardware combination supported by that version of IDL. The following line, which might be located in a file executed by your shell when you log in (`.cshrc`, `.login`) will add the location of the proper DLM for your current system to IDL's `!DLM_PATH` at startup:

```
% setenv IDL_DLM_PATH
  "/usr/local/myd1m/<IDL_VERSION_DIRNAME>/<IDL_BIN_DIRNAME>:<IDL_DEFAULT>"
```

(all on a single line). Setting the Windows environment variable `IDL_DLM_PATH` to a similar string would produce the same result on a Windows system.

!EDIT_INPUT

An integer variable indicating whether keyboard line editing is enabled (when set to a non-zero value) or disabled (when set to zero). By default, !EDIT_INPUT is set equal to one, and line editing is enabled.

By default, IDL saves the last 20 command lines. You can change the number of command lines saved in the recall buffer by setting !EDIT_INPUT equal to the number of lines you would like to save. In order for the change to take effect, IDL must be able to process the assignment statement before providing a command prompt. This means that you must put the assignment statement in the IDL startup file. (See “[Startup Files](#)” in Chapter 1 of the *Using IDL* manual for more information on startup files.)

!HELP_PATH

A string variable listing the directories IDL will search for online help files. On UNIX systems, help files must be Adobe Portable Document Format (.pdf) files or HTML (.html or .htm) files. On Windows systems, help files can be HTML Help (.chm), Windows Help (.hlp), Portable Document Format (.pdf), or HTML (.html or .htm) files.

!HELP_PATH is initialized from the environment variable IDL_HELP_PATH at startup. This initialization is similar to that performed for IDL_PATH (see “[!PATH](#)” on page 4371), including recursive path expansion denoted with a leading “+” but *not* including any use of IDLDE preferences settings. See “[Environment Variables Used by IDL](#)” in Chapter 1 of the *Using IDL* manual for details on setting the IDL_HELP_PATH environment variable

If the IDL_HELP_PATH environment variable is not defined, IDL supplies a default that contains the directory in the IDL distribution where the help files supplied by RSI reside.

To change the value of !HELP_PATH for the duration of an IDL session, simply set the variable equal to a new string containing the desired path. See “[Changing the Value of !PATH After IDL Starts](#)” on page 4372 for tips that also apply to setting the value of !HELP_PATH.

!JOURNAL

A read-only long-integer variable containing the logical unit number of the file used for journal output.

!MAKE_DLL

The MAKE_DLL procedure and the CALL_EXTERNAL function's AUTO_GLUE keyword use the standard system C compiler and linker to generate sharable libraries that can be used by IDL in various contexts (CALL_EXTERNAL, DLMS, LINKIMAGE). There is a great deal of variation possible in the use of these tools between different platforms, operating system versions, and compiler releases. The !MAKE_DLL system variable is used to configure how IDL uses them for the current platform.

The !MAKE_DLL structure is defined as follows:

```
{ !MAKE_DLL, COMPILE_DIRECTORY:'', COMPILER_NAME:'', CC:'', LD:'' }
```

The meaning of the fields of !MAKE_DLL are given in the following table. When expanding !MAKE_DLL.CC and !MAKE_DLL.LD, IDL substitutes text in place of the PRINTF style codes described in the following table. These codes are case-insensitive, and can be either upper or lower case.

It is possible to use C compilers other than the one assumed by RSI in !MAKE_DLL to build sharable libraries. To do so, you can alter the contents of !MAKE_DLL or use the CC and/or LD keyword to MAKE_DLL and CALL_EXTERNAL. Please understand that RSI cannot and does not maintain a list of all possible compilers and the necessary compiler options. This information is available in your compiler and

system documentation. It is the programmers responsibility to understand the rules for their chosen compiler.

Field	Meaning
COMPILE_DIRECTORY	<p>IDL requires a place to create the intermediate files necessary to build a sharable library, and possibly the final library itself. Unless told to use an explicit directory, it uses the directory given by the COMPILE_DIRECTORY field of !MAKE_DLL. If the IDL_MAKE_DLL_COMPILE_DIRECTORY environment variable is set, IDL uses its value to initialize the COMPILE_DIRECTORY field. Otherwise, IDL supplies a standard location.</p> <p>Note - If the directory given by !MAKE_DLL.COMPILE_DIRECTORY does not exist when IDL needs it, IDL automatically creates it for you if possible.</p>
COMPILER_NAME	<p>A string containing the name of the C compiler used by RSI to build the currently running IDL. This field is not used by IDL, and exists solely for informational purposes and to help the end user decide which C compiler to install on their system.</p>
CC	<p>A string used by IDL as a template to construct the command for using the C compiler. This template uses PRINTF style substitution codes, as described in the following table.</p>
LD	<p>A string used by IDL as a template to construct the command for using the linker. This template uses PRINTF style substitution codes, as described in the following table.</p>

Table D-24: Meaning of !MAKE_DLL Fields

The following table describes the substitution codes for the CC and LD fields:

Code	Meaning
%B %b	The base name of a C file to compile. For example, if the C file is <code>moose.c</code> , then %B substitutes <code>moose</code> .
%C %c	The name of the C file.
%E %e	The name of the linker options file. This file, which is automatically generated by IDL as needed, is used to control the linker. Under UNIX, the system documentation refers to this as an export file, or a linker map file. Microsoft Windows calls it a <code>.DEF</code> file.
%F %f	This substitution code is no longer meaningful, and will be ignored.
%L %l	The name of the resulting sharable library. IDL constructs this name by using the base name (%B) and adding the appropriate suffix for the current platform (<code>.dll</code> , <code>.so</code> , <code>.sl</code> , <code>.exe</code> , ...).
%O %o	An object file name. IDL constructs this name by using the base name (%B) and adding the appropriate suffix for the current platform (<code>.o</code> , <code>.obj</code>).
%X %x	When expanding <code>!MAKE_DLL.CC</code> , any text supplied via the <code>EXTRA_CFLAGS</code> keyword to <code>MAKE_DLL</code> or <code>CALL_EXTERNAL</code> is inserted in place of %X. IDL does not interpret this text. It is the users responsibility to ensure that it is meaningful in the command. When expanding <code>!MAKE_DLL.LD</code> , the text from the <code>EXTRA_LFLAGS</code> keyword is substituted. The primary use for this code is to include necessary header include directories and link libraries.
%%	Replaced with a single % character.

Table D-25: Description of CC and LD Field Codes

!MORE

An integer variable indicating whether IDL should paginate help output sent to a tty device. Setting `!MORE` to zero (0) prevents IDL from paginating the output text. A non-zero value (the default) causes IDL to display output text one screen at a time.

!PATH

A string variable listing the directories IDL will search for libraries and include files.

Note

The current directory is always searched before consulting !PATH.

!PATH is initialized from the environment variable IDL_PATH when IDL starts. See [“Environment Variables Used by IDL”](#) in Chapter 1 of the *Using IDL* manual for details on setting the IDL_PATH environment variable.

Note

If the IDL_PATH environment variable does not exist, IDL uses the value set in the **Path** tab of the **Preferences** dialog of the IDLDE. If no preference value is set, or if you are using the UNIX command-line interface rather than the IDLDE, IDL uses a default value in order to initialize !PATH.

Note on Path Expansion

When IDL starts, it reads the IDL_PATH environment variable (or the default value, if IDL_PATH does not exist) and builds a value to be stored in !PATH from the specified initialization value. While the initialization value can consist of a normal path string specifying all directories to be included in !PATH, it can also take advantage of several special values that IDL uses to dynamically create the value of !PATH. These values (the “+” symbol and the “<IDL_*>” strings) are described in detail in [“The Path Definition String”](#) under [“EXPAND_PATH”](#) in the *IDL Reference Guide* manual.

Path Caching

By default, as IDL searches directories included in the !PATH system variable for .pro or .sav files to compile, it creates an in-memory list of *all* .pro and .sav files contained in each directory. When IDL later searches for a .pro or .sav file, before attempting to open the file in a given directory, IDL checks the path cache to determine whether the directory has already been cached. If the directory is included in the cache, IDL uses the cached information to determine whether the file will be found in that directory, and will only attempt to open the file there if the cache tells it that the file exists. By eliminating unnecessary attempts to open files, the path cache speeds the path searching process. See [“PATH_CACHE”](#) in the *IDL Reference Guide* manual for details.

Changing the Value of !PATH After IDL Starts

Once IDL has started, you can alter the value of !PATH by setting it to a new string value. For example, on a UNIX system, to add a directory to !PATH for the duration of an IDL session, you would use a command like the following:

```
!PATH = '/usr2/project/idl_files:' + !PATH
```

Keep the following in mind when changing the value of !PATH by setting its value to a new string:

- Remember to use the proper path string separator character for your platform. If you are writing a cross-platform application that changes !PATH, you may want to use code that looks something like this:

```
pathsep = PATH_SEP(/SEARCH_PATH)
!PATH = 'new_path' + pathsep + !PATH
```

where *new_path* is the path to the directory you want to add.

- You can use the EXPAND_PATH function to generate a path string. This method allows you to specify one or more directories and let IDL figure out whether those directories contain .pro or .sav files. It also allows you to use the “+” character path expansion technique. For example, the following commands add all directories below the *new_path* directory that contain .pro or .sav files to !PATH:

```
pathsep = PATH_SEP(/SEARCH_PATH)
!PATH = EXPAND_PATH('+new_path') + pathsep + !PATH
```

See [“EXPAND_PATH”](#) in the *IDL Reference Guide* manual for details.

!PROMPT

A string variable containing the text string used by IDL to prompt the user for input. The default is IDL>.

!QUIET

A long-integer variable indicating whether informational messages should be printed (0) or suppressed (nonzero). By default, !QUIET is set to zero.

!VERSION

A structure variable containing information about the version of IDL in use. The structure is defined as follows:

```
{ !VERSION, ARCH:'', OS:'', OS_FAMILY:'', OS_NAME:'', $
  RELEASE:'', BUILD_DATE:'', MEMORY_BITS:0, FILE_OFFSET_BITS:0 }
```

The meaning of the fields of !VERSION are given in the following table.

Field	Meaning
ARCH	CPU hardware architecture of the system.
OS	The vendor name of the operating system (for example: AIX, HP-UX, IRIX, linux, MacOS, OSF, sunos, Win32). This is the name of the underlying operating system kernel (not necessarily of the overall operating environment: see OS_NAME). Once an OS name is assigned to a platform by RSI, it is not altered in subsequent releases. This makes it safe for use in IDL programs that need to distinguish between platforms. RSI recommends that you first consider using the OS_FAMILY field before using the OS field, as most programs are mainly concerned with high level platform differences.
OS_FAMILY	The generic name of the operating system (UNIX, Windows). RSI recommends that whenever possible this field (rather than OS or OS_NAME) be used in code that must distinguish between platforms.
OS_NAME	The vendor's name for the operating system environment, as used by the vendor for casual descriptive and promotional purposes. For example, on Sun workstations, the name of the operating system kernel (!VERSION.OS) is "sunos", whereas the name of the overall system (!VERSION.OS_NAME) is "Solaris". Vendors change their descriptive environment names from time to time, and RSI updates the OS_NAME field to reflect this. As a result, RSI recommends that IDL users restrict their use of this field to descriptive textual uses, and that the OS_FAMILY or OS fields of !VERSION be used in code that must distinguish between platforms.
RELEASE	IDL version number.

Table D-26: Meaning of the !VERSION Fields

Field	Meaning
BUILD_DATE	The date the IDL executable was compiled, in the format dictated by ANSI C for the <code>__DATE__</code> macro.
MEMORY_BITS	The number of bits used to address memory. Possible values are 32 or 64. The number of bits used to address memory places a theoretical upper limit on the amount of memory available to IDL.
FILE_OFFSET_BITS	The number of bits used to position file offsets. Possible values are 32 or 64. The number of bits used to position files places a theoretical upper limit on the largest file IDL can access.

Table D-26: Meaning of the !VERSION Fields

If you need to differentiate between different IDL versions in your code, use `!VERSION.OS_FAMILY`. At present, two operating system families are supported: UNIX and Windows. For even more detail, you can use `!VERSION.OS`.

Graphics System Variables

The following system variables control various IDL Direct Graphics functions. These system variables are structures that contain many tags. For example, the command

```
!P.TITLE = 'Cross Section'
```

sets the default plot title.

Many of the functions of the graphics keywords described in [Appendix B, “Graphics Keywords”](#), are also controlled by the system variables !P, !X, !Y, and !Z.

You can change the default style of plots, fonts, etc., by setting the corresponding field in the appropriate system variable. Also, some effects that persist longer than one call are controlled only by system variables. The field !P.MULTI is one example.

!C System Variable

The cursor system variable. Currently, the only function of this system variable is to contain the subscript of the largest or smallest element found by the MAX and MIN functions. That information is better obtained through the optional output arguments to those routines. !C is included only for compatibility with old versions of IDL.

!D System Variable

This system variable is a structure that contains information about the current graphics output device (or window, on a windowing system). Fields, in alphabetical order, are:

FILL_DIST

The line interval, in device coordinates, required to obtain a solid fill.

FLAGS

A longword of flags that provide information about the current device. Each bit is a flag encoded as shown in the following table.

Bit	Value	Function
0	1	Device has scalable pixel size (e.g., PostScript).
1	2	Device can output text at an arbitrary angle using hardware.
2	4	Device can control line thickness with hardware.
3	8	Device can display images.
4	16	Device supports color.
5	32	Device supports polygon filling with hardware.
6	64	Device hardware characters are monospace.
7	128	Device can read pixels (i.e., it supports TVRD).
8	256	Device supports windows.
9	512	Device prints black on a white background (e.g., printers are plotters).
10	1024	Device has <i>no</i> hardware characters.
11	2048	Device does line-fill style polygon filling in hardware.
12	4096	Device will apply Hershey-style embedded formatting commands to device fonts.
13	8192	Device is a pen plotter.
14	16384	Device can transfer 16-bit pixels.
15	32768	Device supports Kanji characters.
16	65536	Device supports widgets.
17	131072	Device has Z-buffer.
18	262144	Device supports TrueType fonts.

Table D-27: !D.FLAGS Bit Definitions

To test whether a particular bit is set on your system, use an IDL command like the following:

```
IF (!D.FLAGS AND value) NE 0 THEN PRINT, 'Bit is set.'
```

where `value` is the value associated with the bit you wish to examine. For example, to check whether the device supports color, use:

```
IF (!D.FLAGS AND 16) NE 0 THEN PRINT, 'Bit is set.'
```

N_COLORS

The number of allowed color values. In the case of devices with windows, this field is set after the window system is initialized. For a monochrome system, `!D.N_COLORS` is 2. For TrueColor displays, `!D.N_COLORS` is $2^{24}-1$ (roughly 16.7 million colors).

NAME

A string containing the name of the device.

ORIGIN

A two-element integer array containing the current pan/scroll offset. An offset of (0, 0) is normal. Positive offsets shift the display memory to the right and upwards. This field has relevance only with devices with hardware pan and scroll abilities.

TABLE_SIZE

The number of color table indices.

Note

For TrueColor visuals, `!D.TABLE_SIZE` will always be 256. If the visual depth is less than 24 bits, IDL emulates 256 entries.

UNIT

The logical number of the file open for output by the current graphics device. This field only has meaning for devices that write to a file if the file is accessible to the user from IDL, and is 0 if no file is open.

For example, the PostScript driver fills this field with the unit number of the file open for PostScript output. In the case of Tektronix output to a file, `!D.UNIT` may be set to either + or – the logical unit number.

WINDOW

The index of the currently open window. This field is set to -1 if no window is currently open. This field is used only with devices that support windows.

X_CH_SIZE, Y_CH_SIZE

The width and height of the rectangle that encloses the “average” character in the current font, in device units (usually pixels).

These values describe the size of the rectangle that contains the “average” character in the current font. (It is not important what the “average” character is; it is used only to calculate a scaling factor that will be applied to all of the characters in the font.) The first element specifies the width of the rectangle in device units (usually pixels), and the second element specifies the height.

For vector and TrueType fonts, the height of the “average” character is determined by the *width* of the rectangle. The aspect ratio of the “average” character remains fixed; the character is scaled so that its width is the value of X_CH_SIZE. The resulting scale factor is then applied to all of the characters in the font. The amount of spacing between lines is determined explicitly by the value of Y_CH_SIZE.

For device fonts, the character size is fixed. When the device font system is in use, the value of X_CH_SIZE is silently ignored, and only the Y_CH_SIZE value is used.

X_PX_CM, Y_PX_CM

The approximate number of pixels per centimeter in the X and Y directions.

X_SIZE, Y_SIZE

The total size of the display or window in the X and Y directions, in device units.

X_VSIZE, Y_VSIZE

The size of the visible area of the display or window. This area can be smaller than the total size fields.

ZOOM

This field contains the current X and Y zoom factors for the display or window. This field has relevance only with devices equipped with hardware zoom. A zoom factor of [1, 1] is normal.

!ORDER System Variable

Controls the direction of image transfers when using the TV, TVSCL, and TVRD procedures. If !ORDER is 0, images are transferred from bottom to top, i.e. the row with a 0 subscript is written on the bottom. Setting !ORDER to 1, transfers images from top to bottom.

!P System Variable

The main plotting system variable structure. All fields, except !P.MULTI, have a directly corresponding keyword parameter in the plot procedures: PLOT, OPLOT, CONTOUR, and SURFACE. Fields, in alphabetical order, are:

BACKGROUND

The background color index. When erasing the screen or page, all pixels are set to this color. The default value is 0. Not all devices support this feature.

Warning

If !P.NOERASE (or the NOERASE keyword) is set, the value of !P.BACKGROUND (or the BACKGROUND keyword) is ignored.

CHANNEL

The default source or destination channel. This field has meaning only on graphics devices that contain multiple display channels, and is device dependent. It may contain either a channel mask or index.

CHARSIZE

The overall character size of all annotations when Hershey fonts are selected. This field has no effect on the character size when hardware (device) fonts are selected, except for devices that support scalable pixel sizes (i.e., Postscript). Note, however, that !P.CHARSIZE always affects the layout and scaling of a plot, regardless of the font system being used. The default size is 1.0.

CHARTHICK

An integer specifying the thickness of the lines used to draw the characters when using the vector drawn fonts. This field has no effect on the appearance of characters drawn with the hardware fonts. Normal thickness is 1.

CLIP

The device coordinates of the clipping window, a 6-element vector of the form $[x_0, y_0, x_1, y_1, z_0, z_1]$, specifying two opposite corners of the volume to be displayed. In the case of two-dimensional displays, the Z coordinates can be omitted. Normally, the clipping window coordinates are implicitly set by PLOT, CONTOUR, SHADE_SURF, and SURFACE to correspond to the plot window. You may also manually set !P.CLIP if you want to specify a different rectangular clipping window or if the clipping coordinates have not yet been set in the current IDL session.

COLOR

The default color index.

FONT

An integer that specifies the graphics text font system to use. Set FONT equal to -1 to select the Hershey character fonts, which are drawn using vectors. Set FONT equal to 0 (zero) to select the device font of the output device. Set FONT equal to 1 (one) to select the TrueType font system. See [Appendix H, “Fonts”](#), for a complete description of IDL’s font systems.

LINestyle

The default style of the lines used to connect points. A line style index of 0 yields a solid line. See “[LINestyle](#)” on page 4337 for a description of the linestyles.

MULTI

!P.MULTI allows making multiple plots on a page or screen. It is a 5-element integer array defined as follows:

!P.MULTI[0] contains the number of plots remaining on the page. If !P.MULTI[0] is less than or equal to 0, the page is cleared, the next plot is placed in the upper left hand corner, and !P.MULTI[0] is reset to the number of plots per page.

Setting !P.MULTI[0] to a value greater than zero can be used to manually set the plotting area to a specific row and column. For example, to plot in the lower left corner of a window of two rows and two columns, set !P.MULTI as follows:

```
!P.MULTI=[ 2, 2, 2 ]
PLOT, X, Y
```

!P.MULTI[1] is the number of plot columns per page. If this value is less than or equal to 0, one is assumed. If more than two plots are ganged in either the X or Y direction, the character size is halved.

!P.MULTI[2] is the number of rows of plots per page. If this value is less than or equal to 0, one is assumed.

!P.MULTI[3] contains the number of plots stacked in the Z dimension.

!P.MULTI[4] is 0 to make plots from left to right (column major), and top to bottom, and is 1 to make plots from top to bottom, left to right (row major).

Note

If !P.MULTI[0] is zero, an erase will occur before the current plot is displayed (unless the /NOERASE keyword is set). This is true no matter whether !P.POSITION and/or !P.REGION are set.

For example, to gang two plots across the page:

```
!P.MULTI = [0, 2, 0, 0, 0]
PLOT, X0, Y0                ;Make left plot.
PLOT, X1, Y1                ;Right plot.
```

To gang two plots vertically:

```
!P.MULTI = [0, 0, 2, 0, 0]
PLOT, X0, Y0                ;Make top plot.
PLOT, X1, Y1                ;Bottom plot.
```

To make four plots per page, two across and two up and down:

```
!P.MULTI = [0, 2, 2, 0, 0]
```

and then call plot four times.

To reset !P.MULTI back to the normal one plot per page:

```
!P.MULTI = 0
```

NOCLIP

A field which, if set, inhibits the clipping of the graphic vectors and vector-drawn text. By default, most routines clip to the plotting window, with the exception of PLOTS and XYOUTS. !P.CLIP contains the clipping rectangle.

NOERASE

Set this field to a non-zero value to inhibit erasing the screen before plotting.

Warning

If !P.NOERASE (or the NOERASE keyword) is set, the value of !P.BACKGROUND (or the BACKGROUND keyword) is ignored.

NSUM

The number of adjacent points to average to obtain a plotted point.

POSITION

Specifies the normalized coordinates of the rectangular plot window. This is a four element floating point vector (x_0, y_0, x_1, y_1) , where (x_0, y_0) is the origin, and (x_1, y_1) is the upper right corner.

!P.POSITION determines the plotting window if x_0 does not equal x_1 , and the POSITION keyword is not present. If set, it overrides the effect of the MARGIN and !P.MULTI variables and keywords.

Note

If !P.POSITION (or the POSITION keyword) or !P.REGION is set, all but the first element of !P.MULTI are ignored.

PSYM

The default plotting symbol index. Each point drawn by PLOT, PLOTS, and OPLOT is marked with a symbol if this field is non-zero. The possible symbols are given in “PSYM” on page 4340.

REGION

A four element vector that specifies the normalized coordinates of the rectangle enclosing the plot region, which includes the plot data window and its surrounding annotation area. It is in the same form as !P.POSITION, (x_0, y_0, x_1, y_1) , where (x_0, y_0) is the origin, and (x_1, y_1) is the upper right corner. It is ignored if !P.REGION[0] is equal to !P.REGION[2].

Note

!P.POSITION (or the POSITION keyword) takes precedence over !P.REGION.

SUBTITLE

The plot subtitle, placed under the X axis label.

T

Contains the homogeneous 4 x 4 transformation matrix. This field is a two-dimensional array of double-precision floating-point values. For more information

about transformations, refer to “[Three-Dimensional Graphics](#)” in Chapter 18 of the *Using IDL* manual.

T3D

Enables the three-dimensional to two-dimensional transformation contained in the homogeneous 4 by 4 matrix !P.T. Note that if T3D is set, !P.T must contain a valid transformation matrix.

THICK

The thickness of the lines connecting points. 1.0 is normal.

TITLE

The main plot title.

TICKLEN

The length of the tick marks, expressed as a fraction of the plot size (from 0.0 to 1.0). The default is 0.02. A value of 0.5 makes a grid. Negative values make the tick marks point outward.

!X, !Y, !Z System Variables

The system variables !X, !Y, and !Z, are structures of type AXIS, that affect the appearance and scaling of the three axes. The fields for !X, !Y, and !Z have identical fields with identical meanings and usage. In addition, almost all fields have corresponding keyword parameters, with identical function, but with temporary effect. For example, to suppress the minor tick marks on the X axis using the !X system variable, you could use the command:

```
!X.MINOR = -1
```

To suppress the tick marks for just one call to plot, you could use the command:

```
PLOT, X, Y, XMINOR = -1
```

The name of the keyword parameter is simply the name of the system variable field, prefixed with the letter X, Y, or Z.

The fields for these system variables, in alphabetical order are:

CHARSIZE

The size of the characters used to annotate the axis and its title when Hershey fonts are selected. This field has no meaning when hardware (i.e. PostScript) fonts are

selected. This field is a scale factor applied to the global scale factor. For example, setting !P.CHAR.SIZE to 2.0, and !X.CHAR.SIZE to 0.5 results in a character size of 1.0 for the X axis.

CRANGE

The output axis range. Setting this variable has no effect; set ![XYZ].RANGE to change the range. ![XYZ].CRANGE[0] always contains the minimum axis value, and ![XYZ].CRANGE[1] contains the maximum axis value of the last plot before extending the axes.

Note

If the axis is logarithmic, the CRANGE field reports the log (base 10) of the minimum and maximum axis values.

Example 1:

```
;Create a 10-element array.
a = INDGEN(10)

;Plot the straight line.
PLOT, a

;Print the minimum and maximum axis values.
PRINT, !X.CRANGE
```

IDL prints:

```
0.00000      10.0000
```

Example 2:

```
;Plot a with logarithmic scaling on the X axis.
PLOT, a, /XLOG

;Print the minimum and maximum axis values.
PRINT, !X.CRANGE
```

The axis is scaled from 10^{-12} to 10^2 . IDL prints:

```
-12.0000      2.00000
```

GRIDSTYLE

The index of the linestyle to be used for tick marks and grids. See “[LINESTYLE](#)” on page 4337 for a description of the linestyles

MARGIN

A 2-element array specifying the margin on the left (bottom) and right (top) sides of the plot window, in units of character size. The plot window is the rectangular area that contains the plot data, i.e. the area enclosed by the axes.

The default values for !X.MARGIN are [10, 3] yielding a 10-character wide left margin and a 3-character wide right margin. The values for !Y.MARGIN are [4, 2], for a 4-character high bottom margin and a 2-character high top margin. While specifying !Z.MARGIN will not cause an error, Z margins are currently ignored.

When calculating the size and position of the plot window, IDL first determines the plot region, the area enclosing the window plus the axis annotation and titles. It then subtracts the appropriate margin from each side, obtaining the window.

Setting !P.POSITION, or specification of the POSITION parameter overrides the effect of this field.

MINOR

The number of minor tick mark intervals. If !X.MINOR is 0, the default, the number of minor tick intervals is automatically determined from the tick mark increment. You can force a given number of minor tick intervals by setting this field to the desired number. To suppress minor tick marks, set !X.MINOR to -1.

OMARGIN

A 2-element array specifying the “outer” margin on the left (bottom) and right (top) sides of a multi-plot window, in units of character size. A multi-plot window is created by setting the !P.MULTI system variable field. OMARGIN controls the amount of space around the entire plot area, including individual plot margins set with !X.MARGIN and !Y.MARGIN. The default values for !X.OMARGIN and !Y.OMARGIN are [0, 0].

When calculating the size and position of the individual plots, IDL first determines the plot region, the area enclosing the window plus the axis annotation and titles. It then subtracts the appropriate margin from each side, obtaining the window.

Setting !P.POSITION, or specification of the POSITION parameter overrides the effect of this field.

RANGE

The input axis range, a 2-element vector. The first element is the axis minimum, and the second is the maximum. Set this field, or use the corresponding keyword parameter, to specify the data range to plot. If axis end point rounding is selected (see

STYLE above), the final axis range may not be equal to this input range. The field !X.CRANGE contains the axis range used for the plot before extending the axes. Set both elements equal to 0 for automatic axis ranges:

```
!X.RANGE = 0
```

For example, to force the X axis to run from 5.5 to 8.3:

```
!X.RANGE = [5.5, 8.3]
PLOT, X, Y
```

Alternatively, by using keywords:

```
PLOT, X, Y, XRANGE=[5.5, 8.3]
```

Note that even though the range was set to (5.5, 8.3), the resulting plot has a range of (5.5, 8.5), because axis rounding is the default.

REGION

Contains the normalized coordinates of the region. This field is similar to WINDOW, in that it is set by the graphics procedures and is a 2-element floating point array. To change the default plotting region, set !P.REGION.

S

The scaling factors for converting between data coordinates and normalized coordinates (a 2-element array). The formula for conversion from data (X_d) to normalized (X_n) coordinates is $X_n = S_1 X_d + S_0$

If logarithmic scaling is in effect, substitute $\log_{10}(X_d)$ for X_d .

The CONVERT_COORD function can be used to convert between coordinate systems. The user should save and restore these fields when switching between windows or devices with different sizes and/or scaling.

STYLE

The style of the axis encoded as bits in a longword. The axis style can be set to exact, extended, none, or no box using this field. The following table lists the axis style bit values:

Bit	Value	Function
0	1	Exact. By default, the end points of the axis are rounded in order to obtain even tick increments. Setting this bit inhibits rounding, making the axis fit the data range exactly.
1	2	Extend. If this bit is set, the axes are extended by 5% in each direction, leaving a border around the data.
2	4	None. If this bit is set, the axis and its annotation are not drawn.
3	8	No box. Normally, PLOT and CONTOUR draw a box-style axis with the data window surrounded by axes.
4	16	Inhibits setting the Y axis minimum value to zero when the data are all greater than 0. The keyword YNOZERO sets this bit temporarily.

Table D-28: Axis Style Bit Values

Note that this system variable field is set bitwise, so multiple effects can be set by adding values together. For example, to make an X axis that is both exact (value 1) and suppresses the box style (setting 8), set the !X.STYLE system variable to 1+8, or 9.

For example, to set the Y axis style to exact using the !Y system variable:

```
!Y.STYLE = 1
```

or by using a keyword parameter:

```
PLOT, X, Y, YSTYLE = 1
```

THICK

The thickness of the axis line. 1.0 is normal.

TICKFORMAT

Set this field to a format string or a string containing the name of a function that returns a string to be used to format the axis tick mark labels.

See “[XYZ]TICKFORMAT” on page 4345 for more information.

TICKINTERVAL

A scalar indicating the interval between major tick marks for the first axis level. This setting takes precedence over `![XYZ].TICKS`.

For example, if `!X.TICKUNITS=["Seconds", "Hours", "Days"]`, and `!X.TICKINTERVAL=30`, then the interval between major ticks for the first axis level will be 30 seconds.

See [“`\[XYZ\]TICKINTERVAL`”](#) on page 4348 for more information.

TICKLAYOUT

Set this keyword to a scalar that indicates the tick layout style to be used to draw each level of the axis.

See [“`\[XYZ\]TICKLAYOUT`”](#) on page 4348 for more information.

TICKLEN

The lengths of tick marks (expressed in normal coordinates) for the individual axes.

TICKNAME

The annotation for each tick. A string array of up to 60 elements. Setting elements of this array allows direct specification of the tick label. If this element contains a null string, the default value, IDL annotates the tick with its numeric value. Setting the element to a 1-blank string suppresses the tick annotation.

For example, to produce a plot with an abscissa labeled with the days of the week:

```
;Set up X axis tick labels.
!X.TICKNAME = ['SUN', 'MON', 'TUE', 'WED', $
              'THU', 'FRI', 'SAT']

;Use six tick intervals, requiring seven tick labels.
!X.TICKS = 6

;Plot the data, this assumes that Y contains 7 elements.
PLOT, Y
```

The same plot can be produced, using keyword parameters, with:

```
;Set fields, as above, only temporarily.
PLOT, Y, XTICKN = ['SUN', 'MON', 'TUE', 'WED', $
                  'THU', 'FRI', 'SAT'], XTICKS = 6
```

TICKS

The number of major tick intervals to draw for the axis. If `!X.TICKS` is set to 0, the default, IDL will select from three to six tick intervals. Setting this field to n , where $n > 1$, produces exactly n tick intervals, and $n+1$ tick marks. Setting this field equal to 1 suppresses tick marks.

TICKUNITS

Set this keyword to a string (or a vector of strings) indicating the units to be used for axis tick labeling.

Note

The singular form of each of the time value strings is also acceptable (e.g, `!X.TICKUNITS='Day'` is equivalent to `!X.TICKUNITS='Days'`).

Note

To set the `![XYZ].TICKUNITS` field to a single string, the following approach is recommended:

```
!X.TICKUNITS = '' ; Clear all previous tick unit strings.
!X.TICKUNITS = ['Days'] ;Single unit string in array.
```

The following:

```
!X.TICKUNITS = 'Days'
```

will copy the 'Days' string to all levels, resulting in a multi-level axis.

See “[[XYZ](#)][TICKUNITS](#)” on page 4349 for more information.

TICKV

An array of up to 60 elements containing the data values for each tick mark. You can directly specify the location of each tick by setting `!X.TICKS` to the number of tick marks (the number of intervals plus 1) and storing the data values of the tick marks in `!X.TICKV`. If, as is true by default, `!X.TICKV[0]` is equal to `!X.TICKV[1]`, IDL automatically determines the value of the tick marks.

TITLE

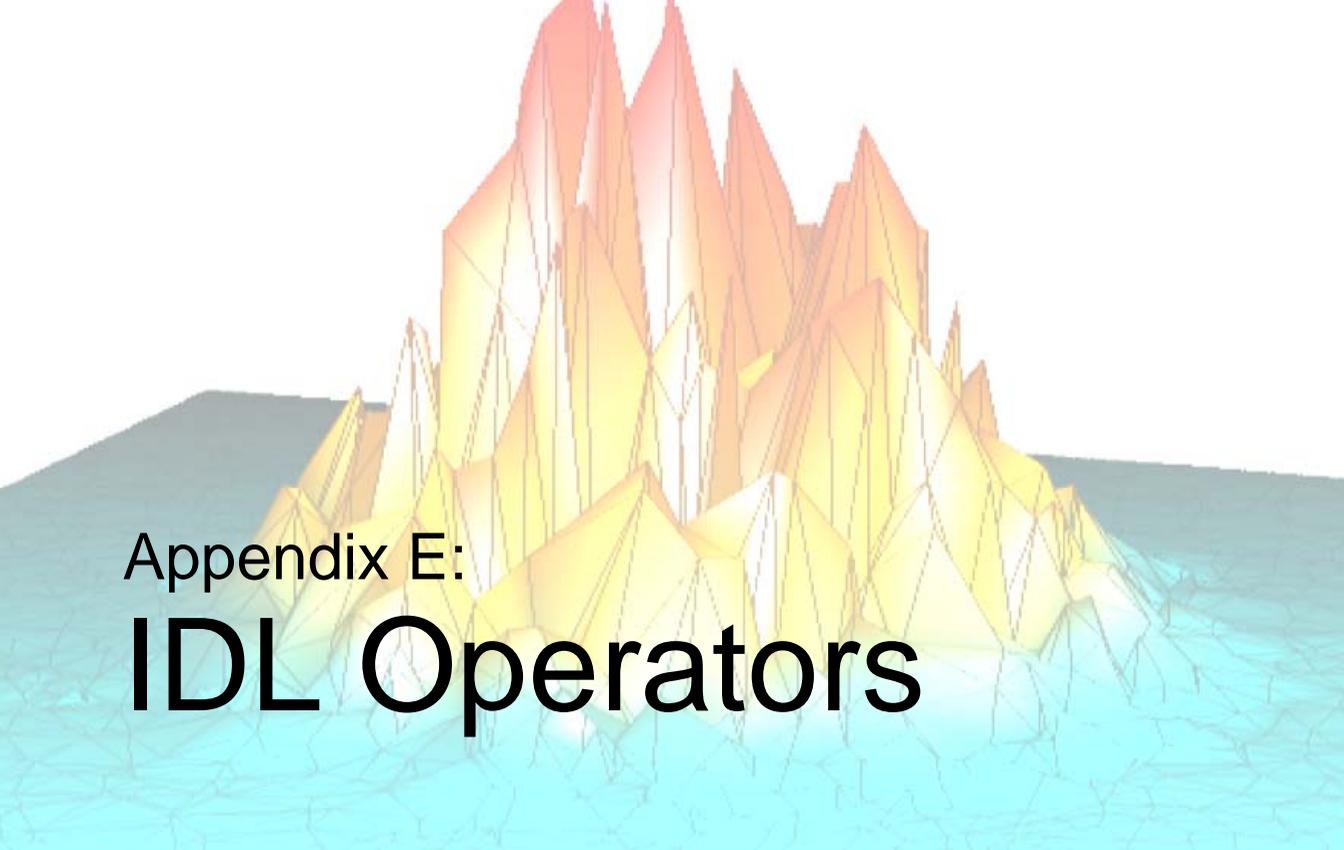
A string containing the axis title.

TYPE

The type of axis, 0 for linear, 1 for logarithmic.

WINDOW

Contains the normalized coordinates of the axis end points, the plot data window. This field is set by `PLOT`, `CONTOUR`, `SHADE_SURF`, and `SURFACE`. Changing its value has no effect. A 2-element floating point array. To change the default plotting window, set `!P.POSITION`. The keyword parameter `POSITION` sets the plot data window on a per call basis.



Appendix E: IDL Operators

This appendix lists all IDL operators and provides brief examples of their usage. For more information on IDL operators, see [Chapter 2, “Expressions and Operators”](#) in the *Building IDL Applications* manual. The following topics are covered in this appendix:

Mathematical Operators	4392	Bitwise Operators	4397
Minimum and Maximum Operators	4394	Relational Operators	4399
Matrix Operators	4395	Other Operators	4400
Logical Operators	4396	Operator Precedence	4402

Mathematical Operators

Operator	Description	Example
+	Addition	<pre>;Store the sum of 3 and 6 in B: B = 3 + 6</pre>
	String Concatenation	<pre>;Store the string value of "John Doe" in B: B = 'John' + ' ' + 'Doe'</pre>
++	Increment	<p>Adds one to the operand:</p> <pre>A = 3 A++ PRINT, A</pre> <p>IDL Prints:</p> <pre>4</pre> <p>Note - The increment operator supports both pre- and post-fix syntax. See “Increment/Decrement” in Chapter 2 of the <i>Building IDL Applications</i> manual.</p>
-	Subtraction	<pre>;Store the value of 5 subtracted from 9 in C: C = 9 - 5</pre>
	Negation	<pre>;Change the sign of C: C = -C</pre>
--	Decrement	<p>Subtracts one from the operand:</p> <pre>A = 3 A-- PRINT, A</pre> <p>IDL Prints:</p> <pre>2</pre> <p>Note - The decrement operator supports both pre- and post-fix syntax. See “Increment/Decrement” in Chapter 2 of the <i>Building IDL Applications</i> manual.</p>

Table E-29: Mathematical Operators

Operator	Description	Example
*	Multiplication	<pre>;Store the product of 2 and 5 in variable C: C = 2 * 5</pre>
	Pointer dereference	If <code>ptr</code> is a valid pointer (created via the <code>PTR_NEW</code> function), then <code>*ptr</code> is the value held by the heap variable that <code>ptr</code> points to. For more information on IDL pointers, see Chapter 8, “Pointers” in the <i>Building IDL Applications</i> manual.
/	Division	<pre>;Store result of 10.0 divided by 3.2 in ;variable D: D = 10.0/3.2</pre>
^	Exponentiation	<pre>;Store result of 2 raised to the 3rd power in ;variable B: B = 2^3</pre>
MOD	Modulo	<pre>;Print the value of 9 modulo 5: PRINT, 9 MOD 5</pre> IDL Prints: 4

Table E-29: Mathematical Operators (Continued)

Minimum and Maximum Operators

Operator	Description	Example
<	<p>Minimum operator. The value of “A < B” is equal to the smaller of A or B.</p> <p>For complex numbers the absolute value is used.</p>	<pre>;Set A equal to 3. A = 5 < 3 ;Set all points in array ARR that are larger ;than 100 to 100: ARR = ARR < 100 ;Set X to the smallest of the three operands: X = X0 < X1 < X2</pre>
>	<p>Maximum operator. “A > B” is equal to the larger of A or B.</p> <p>For complex numbers the absolute value is used.</p>	<pre>;'>' is used to avoid taking the log of zero ;or negative numbers: C = ALOG(D > 1E - 6) ;Plot positive points only. Negative points ;are plotted as zero: PLOT, ARR > 0</pre>

Table E-30: Minimum and Maximum Operators

Matrix Operators

Operator	Description	Example
#	Computes array elements by multiplying the columns of the first array by the rows of the second array.	<pre> ;A 3-column by 2-row array: array1 = [[1, 2, 1], [2, -1, 2]] ;A 2-column by 3-row array: array2 = [[1, 3], [0, 1], [1, 1]] PRINT, array1#array2 IDL prints: 7 -1 7 2 -1 2 3 1 3 </pre>
##	Computes array elements by multiplying the rows of the first array by the columns of the second array.	<pre> ;A 3-column by 2-row array: array1 = [[1, 2, 1], [2, -1, 2]] ;A 2-column by 3-row array: array2 = [[1, 3], [0, 1], [1, 1]] PRINT, array1##array2 IDL prints: 2 6 4 7 </pre>

Table E-31: Matrix Operators

Logical Operators

Operator	Description	Example
&&	<p>Logical AND. Returns 1 whenever both of its operands are true; otherwise, returns 0. Non-zero numerical values, non-null strings, and non-null heap variables (pointers and object references) are considered true, everything else is false.</p> <p>Operands must be scalars or single-element arrays. The && operator <i>short-circuits</i>; the second operand will not be evaluated if the first is false.</p>	<pre>PRINT, 5 && 7 IDL Prints: 1 PRINT, 5 && 2 IDL Prints: 1 PRINT, 4 && 0 IDL Prints: 0</pre>
	<p>Logical OR. Returns 1 whenever either of its operands are true; otherwise, returns 0. Uses the same test for “truth” as the && operator.</p> <p>Operands must be scalars or single-element arrays. The operator <i>short-circuits</i>; the second operand will not be evaluated if the first is true.</p>	<pre>IF ((5 GT 3) (4 GT 5)) THEN \$ PRINT, 'True'</pre> <p>IDL Prints: True</p>
~	<p>Logical negation. Returns 1 when its operand is false; otherwise, returns 0. Uses the same test for “truth” as the && operator.</p>	<pre>PRINT, ~ [1, 2, 0]</pre> <p>IDL Prints: 0 0 1</p>

Table E-32: Logical Operators

Bitwise Operators

Operator	Description	Example
AND	Bitwise AND. For integer, longword, and byte operands, a bitwise AND operation is performed. For operations on other types, the result is equal to the second operand if the first operand is not equal to zero or the null string; otherwise, the result is zero or the null string.	<pre>PRINT, (5 GT 2) AND (4 GT 2) IDL Prints: 1 PRINT, (5 GT 2) AND (4 GT 5) IDL Prints: 0 PRINT, 5 AND 7 IDL Prints: 5 PRINT, 5 AND 2 IDL Prints: 0 PRINT, 4 AND 2 IDL Prints: 0</pre>
NOT	Bitwise complement. For integer, longword, and byte operands, NOT returns the complement of each bit of the operand. For floating-point operands, the result is 1.0 if the operand is zero; otherwise, the result is zero. Not valid for string or complex operands.	<pre>PRINT, NOT 1 IDL Prints: -2 IF (NOT (5 GT 6)) THEN \$ PRINT, 'True' IDL Prints: True</pre>
OR	Bitwise OR. For integer or byte operands, a bitwise inclusive OR is performed. For floating- point operands, returns the first operand if it is non-zero, or the 2nd operand otherwise.	<pre>IF ((5 GT 3) OR (4 GT 5)) THEN \$ PRINT, 'True' IDL Prints: True</pre>

Table E-33: Logical Operators

Operator	Description	Example
XOR	Bitwise exclusive OR. XOR is only valid for byte, integer, and longword operands. A bit in the result is set to 1 if the corresponding bits in the operands are different; if they are equal, it is set to zero.	<pre>IF ((5 GT 3) XOR (4 GT 5)) THEN \$ PRINT, 'Different' \$ ELSE PRINT, 'Same'</pre> <p>IDL Prints: Different</p>

Table E-33: Logical Operators (Continued)

Relational Operators

Operator	Description	Example
EQ	Equal to	<pre>;Determine if A equals B: IF (A EQ B) THEN PRINT, 'True'</pre>
GE	Greater than or equal to	<pre>;Determine if A is greater than or equal ;to B: IF (A GE B) THEN PRINT, 'True'</pre>
GT	Greater than	<pre>;Determine if A is greater than B: IF (A GT B) THEN PRINT, 'True'</pre>
LE	Less than or equal to	<pre>;Determine if A is less than or equal ;to B: IF (A LE B) THEN PRINT, 'True'</pre>
LT	Less than	<pre>; Determine if A is less than B: IF (A LT B) THEN PRINT, 'True'</pre>
NE	Not equal to	<pre>; Determine if A does not equal B: IF (A NE B) THEN PRINT, 'True'</pre>

Table E-34: Relational Operators

Other Operators

Operator	Description	Examples
[]	Array concatenation. The expression [A, B] is an array formed by concatenating A and B.	<pre>;Define C as three-point vector: C = [0, 1, 3] ;Add 5 to the end of C: PRINT, [C, 5] IDL Prints: 0 1 3 5 ;Insert -1 at the beginning of C: PRINT, [-1, C] IDL Prints: -1 0 1 3</pre>
	Enclose array subscripts	<pre>A = [2, 1, 5] ;Print the 3rd element in A: PRINT, A[2] IDL Prints: 5</pre>
()	Group expressions to control order of evaluation	<pre>PRINT, 3 + 4 * 2 ^ 2 / 2 IDL Prints: 11 PRINT, (3 + (4 * 2) ^ 2 / 2) IDL Prints: 35</pre>
=	Assignment	<pre>;Assign 5 to variable A: A = 5 ;Assign "Hello World" to variable B: B='Hello World'</pre>

Table E-35: Other Operators

Operator	Description	Examples
<i>op=</i>	Compound Assignment where <i>op</i> is one of the following operators: ##, #, *, +, -, /, <, >, ^, AND, EQ, GE, GT, LE, LT, MOD, NE, OR, XOR	<p>Applies the specified operation to the target variable “in place,” without making a copy of the variable. For example,</p> <pre>A += 5</pre> <p>adds 5 to the value of the variable A.</p> <p>Note -</p> <pre>A op= expression</pre> <p>is equivalent to:</p> <pre>A = TEMPORARY(A) op (expression)</pre> <p>See “Compound Assignment Operators” in Chapter 12 of the <i>Building IDL Applications</i> manual for details.</p>
?:	<p>Conditional expression.</p> <p>For</p> <pre>value=expr1 ? expr2 : expr3</pre> <p><i>expr1</i> is evaluated first. If <i>expr1</i> is true, then <i>value=expr2</i>. If <i>expr1</i> is false, <i>value=expr3</i>.</p>	<pre>A=6 & B=4 ;Set Z to the greater of A and B: Z = (A GT B) ? A : B PRINT, Z</pre> <p>IDL Prints: 6</p>

Table E-35: Other Operators (Continued)

Operator Precedence

The following table lists IDL's operator precedence. Operators with the highest precedence are evaluated first. Operators with equal precedence are evaluated from left to right.

Priority	Operator
First (highest)	() (parentheses, to group expressions)
	[] (brackets, to concatenate arrays)
Second	. (structure field dereference)
	[] (brackets, to subscript an array)
	() (parentheses, used in a function call)
Third	* (pointer dereference)
	^ (exponentiation)
	++ (increment)
	-- (decrement)
Fourth	* (multiplication)
	# and ## (matrix multiplication)
	/(division)
	MOD (modulus)
Fifth	+ (addition)
	- (subtraction and negation)
	< (minimum)
	> (maximum)
	NOT (bitwise negation)

Table E-36: Operator Precedence

Priority	Operator
Sixth	EQ (equality)
	NE (not equal)
	LE (less than or equal)
	LT (less than)
	GE (greater than or equal)
	GT (greater than)
Seventh	AND (bitwise AND)
	OR (bitwise OR)
	XOR (bitwise exclusive OR)
Eighth	&& (logical AND)
	(logical OR)
	~ (logical negation)
Ninth	?: (conditional expression)

Table E-36: Operator Precedence (Continued)



Appendix F:

Special Characters

Within the IDL environment, a number of characters have special meanings.

The following table lists characters with special interpretations and states their functions in IDL. These characters are discussed further in the descriptions following the table.

Character	Function
!	<ul style="list-style-type: none">• First character of system variable names and font-positioning commands.
'	<ul style="list-style-type: none">• Delimit string constants• Indicate part of octal or hex constant
;	<ul style="list-style-type: none">• Begin comment field

Table F-37: Special Characters

Character	Function
\$	<ul style="list-style-type: none"> • Continue current command on the next line • Issue operating system command if entered on a line by itself.
"	<ul style="list-style-type: none"> • Delimit string constants or precede octal constants
.	<ul style="list-style-type: none"> • Indicate constant is floating point • Start executive command
&	<ul style="list-style-type: none"> • Separate multiple statements on one line
:	<ul style="list-style-type: none"> • End label identifiers • Separate start and end subscript ranges
*	<ul style="list-style-type: none"> • Multiplication operator • Array subscript range • Pointer dereference (if in front of a valid pointer)
@	<ul style="list-style-type: none"> • Include file • Execute IDL batch file
?	<ul style="list-style-type: none"> • Invokes online help when entered at the IDL command line. • Part of the ?: ternary operator used in conditional expressions.

Table F-37: Special Characters (Continued)

Exclamation Point (!)

The exclamation point is the first character of names of IDL system-defined variables. System variables are predefined scalar variables of a fixed type. Their purpose is to override defaults for system procedures, to return status information, and to control the action of IDL.

Apostrophe (')

The apostrophe delimits string literals and indicates part of an octal or hex constant.

Semicolon (;)

The semicolon is the first character of the optional comment field of an IDL statement. All text on a line following a semicolon is ignored by IDL. A line can consist of a comment only or both a valid statement and a comment.

Dollar Sign (\$)

The dollar sign at the end of a line indicates that the current statement is continued on the following line. The dollar sign character can appear anywhere a space is legal except within a string constant or between a function name and the first open parenthesis. Any number of continuation lines are allowed.

When the \$ character is entered as the first character after the IDL prompt, the rest of the line is sent to the operating system as a command. If \$ is the only character present, an interactive subprocess is started. Under UNIX, IDL execution suspends until the new shell process terminates.

Quotation Mark (")

The quotation mark precedes octal numbers, which are always integers, and delimits string constants. Example: "100B is a byte constant equal to 64 base 10 and "Don't drink the water" is a string constant.

Period (.)

The period or decimal point indicates in a numeric constant that the number is of floating-point or double-precision type. Example: 1.0 is a floating-point number. Also, in response to the IDL prompt, the period begins an executive command. For example,

```
.run myfile
```

causes IDL to compile the file *myfile.pro*. If *myfile.pro* contains a main program, the program also will be executed. In addition, the period precedes the name of a tag when referring to a field within a structure. For example, a reference to a tag called NAME in a structure stored in the variable A is A.NAME.

Ampersand (&)

The ampersand separates multiple statements on one line. Statements can be combined until the maximum line length is reached. For example, the following line contains two statements:

```
I = 1 & PRINT, 'value:', I
```

Colon (:)

The colon ends label identifiers. Labels can only be referenced by GOTO and ON_ERROR statements. The following line contains a statement with the label LOOP1.

```
LOOP1: X = 2.5
```

The colon also separates the starting and ending subscripts in subscript range specifiers. For example, A(3:6) designates elements three to six of the variable A.

Asterisk (*)

The asterisk represents one of the following, depending on context:

1. Multiplication ($3 * 3$).
2. An ending subscript range equal to the size of the dimension. For example, A[3 : *] represents all elements of the vector A from A[3] to the last element, while B[* , 3] represents all elements of row four of matrix B.
3. A pointer dereference operation. For example, if ptr is a valid pointer (created via the PTR_NEW function), then *ptr is the value held by the heap variable that ptr points to. For more information on IDL pointers, see [Chapter 8](#), “Pointers” in the *Building IDL Applications* manual.

At Sign (@)

The “at” sign is used both as an include character and to signal batch execution.

@ as an Include Character

The “at” sign at the beginning of a line causes the IDL compiler to substitute the contents of the file whose name appears after the @ for the line. If the full path name is not specified after the @ symbol, IDL searches the current directory and a list of known locations where procedures are kept.

- **UNIX:** IDL searches for the file in the list of directories (as established by the environment variable `IDL_PATH`) stored in the system variable `!PATH`.
- **Windows:** IDL searches for the file in the list of directories stored in the system variable `!PATH` (specified in the “Preferences” dialog of the File menu).

For example, the line

```
@doit
```

when included in a file, causes the file *doit.pro* to be compiled in its place. (The suffix *.pro* is the default for IDL program files.) When the end of the file is reached, compilation resumes at the line after the `@`.

@ to Signal Batch Processing

When IDL is running in interactive mode, a line beginning with the character `@` is entered in response to the IDL prompt and the file is opened for batch input. See [Chapter 10, “Executing Batch Jobs in IDL”](#) in the *Using IDL* manual for details.

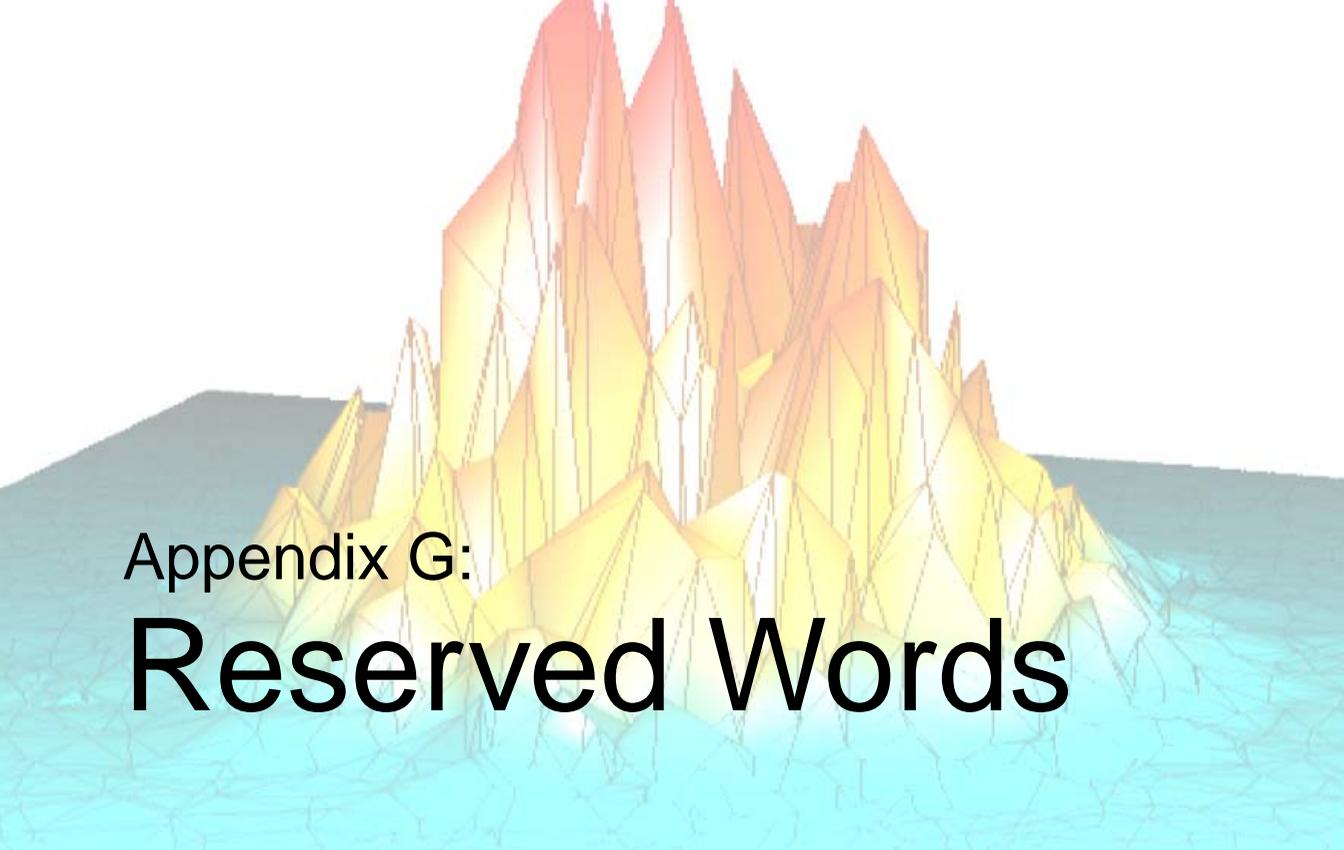
Question Mark (?)

The question mark is used as follows:

- When entered at the IDL command line, the IDL online help facility is invoked.
- Used in conditional expressions as part of the `?:` ternary operator. For example:

```
; A shorter way of saying IF (a GT b) THEN z=a ELSE z=b:  
z = (a GT b) ? a : b
```

For more on conditional expressions, see [“Conditional Expression”](#) in Chapter 2 of the *Building IDL Applications* manual.



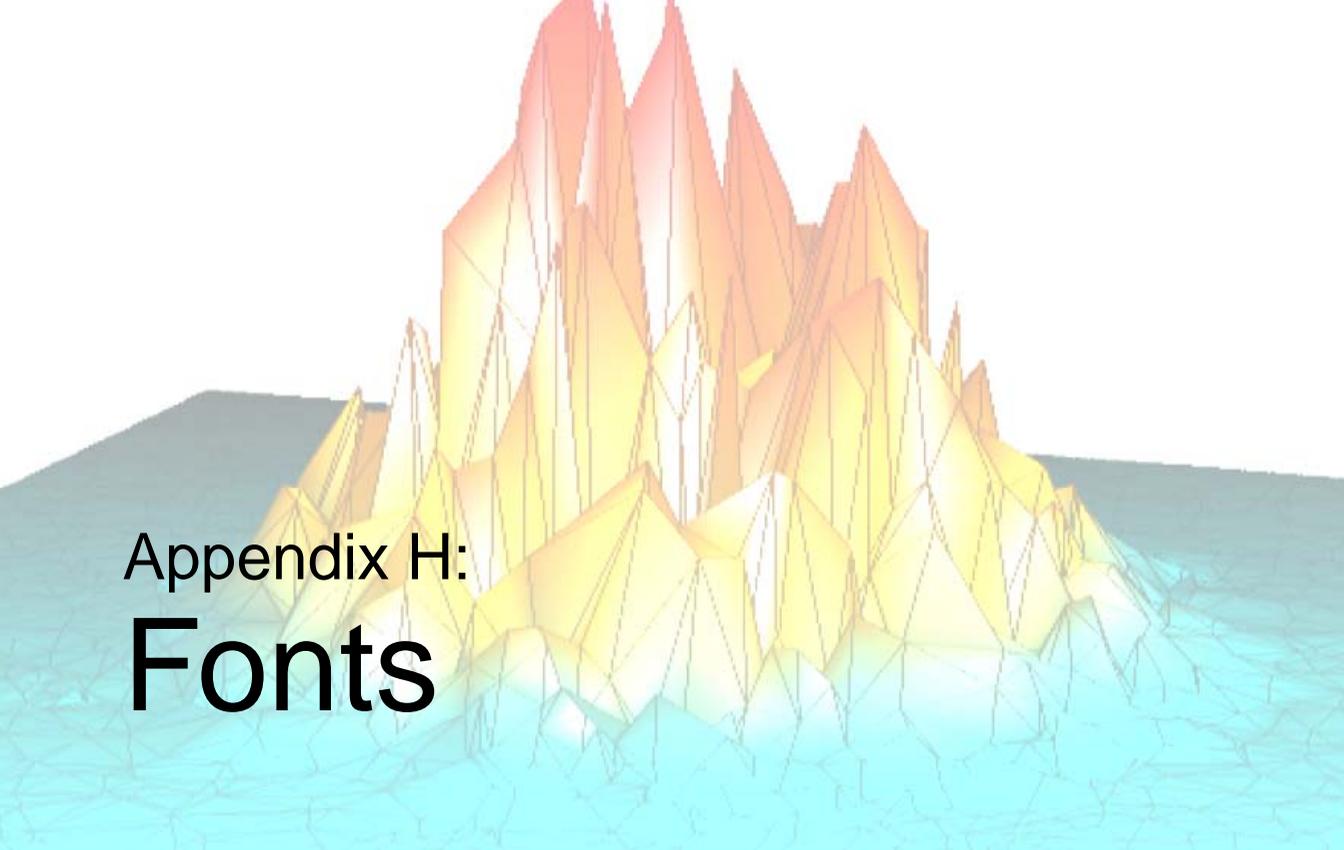
Appendix G:

Reserved Words

Variables, user-written procedures, and user-written functions should not have the same names as IDL functions or procedures. Re-using names of IDL routines can lead to syntax errors or to “hiding” variables. In addition, certain words representing IDL language constructs are strictly forbidden—using any of these *reserved words* as a variable, procedure, or function name will cause an immediate syntax error. The following table lists all of the reserved words in IDL.

AND	BEGIN	BREAK
CASE	COMMON	COMPILE_OPT
CONTINUE	DO	ELSE
END	ENDCASE	ENDELSE
ENDFOR	ENDIF	ENDREP
ENDSWITCH	ENDWHILE	EQ
FOR	FORWARD_FUNCTION	FUNCTION

GE	GOTO	GT
IF	INHERITS	LE
LT	MOD	NE
NOT	OF	ON_IOERROR
OR	PRO	REPEAT
SWITCH	THEN	UNTIL
WHILE	XOR	



Appendix H: Fonts

The following topics are covered in this appendix:

Overview	4414	Choosing a Font Type	4431
Fonts in IDL Direct vs. Object Graphics .	4415	Embedded Formatting Commands	4433
About Vector Fonts	4416	Formatting Command Examples	4437
About TrueType Fonts	4419	TrueType Font Samples	4442
About Device Fonts	4424	Vector Font Samples	4445

Overview

IDL uses three font systems for writing characters on the graphics device: Hershey (vector) fonts, TrueType (outline) fonts, and device (hardware) fonts. This chapter describes each of the three types of fonts, discusses when to use each type, and explains how to use fonts when creating graphical output in IDL.

Vector-drawn fonts, also referred to as *Hershey fonts*, are drawn as lines. They are device-independent (within the limits of device resolution). All vector fonts included with IDL are guaranteed to be available in any IDL installation. See [“About Vector Fonts”](#) on page 4416 for additional details.

TrueType fonts, also referred to here as *outline fonts*, are drawn as character outlines, which are filled when displayed. They are largely device-independent, but do have some device-dependent characteristics. Four TrueType font families are included with IDL; these fonts should display in a similar way on any IDL platform. TrueType font support for IDL Object Graphics was introduced in IDL version 5.0 and support in IDL Direct Graphics was introduced in IDL version 5.1. See [“About TrueType Fonts”](#) on page 4419 for additional details.

Device fonts, also referred to as *hardware fonts*, rely on character-display hardware or software built in to a specific display device. Device fonts, necessarily, are device-dependent and differ from platform to platform and display device to display device. See [“About Device Fonts”](#) on page 4424 for additional details.

Fonts in IDL Direct vs. Object Graphics

This volume deals almost exclusively with IDL Direct Graphics. However, the vector and TrueType font systems described here are also available in the IDL Object Graphics system, described in *Using IDL*.

IDL Direct Graphics

When generating characters for Direct Graphics plots, IDL uses the font system specified by the value of the system variable `!P.FONT`. The normal default for this variable is `-1`, which specifies that the built-in, vector-drawn (Hershey) fonts should be used. Setting `!P.FONT` equal to `1` specifies that TrueType fonts should be used. Setting `!P.FONT` equal to zero specifies that fonts supplied by the graphics device should be used.

The setting of the IDL system variable `!P.FONT` can be overridden for a single IDL Direct Graphics routine (`AXIS`, `CONTOUR`, `PLOT`, `SHADE_SURF`, `SURFACE`, or `XYOUTS`) by setting the `FONT` keyword equal to `-1`, `0`, or `1`.

Once a font system has been selected, an individual font can be chosen either via a formatting command embedded in a text string as described in [“Embedded Formatting Commands”](#) on page 4433, or by setting the value of the `SET_FONT` keyword to the `DEVICE` routine (see [“SET_FONT”](#) on page 4275).

IDL Object Graphics

IDL Object Graphics can use the vector and TrueType font systems. See *Using IDL* for more information on using fonts with Object Graphics. Any TrueType fonts you add to your IDL installation as described in [“About TrueType Fonts”](#) on page 4419 will also be available to the Object Graphics system.

About Vector Fonts

A Hershey Font

The vector fonts used by IDL were digitized by Dr. A.V. Hershey of the Naval Weapons Laboratory. Characters in the vector fonts are stored as equations, and can be scaled and rotated in three dimensions. They are drawn as lines on the current graphics device, and are displayed quickly and efficiently by IDL. The vector fonts are built into IDL itself, and are always available.

All the available fonts are illustrated in [“Vector Font Samples”](#) on page 4445. The default vector font (Font 3, Simplex Roman) is in effect if no font changes have been made.

Using Vector Fonts

To use the vector font system with IDL Direct Graphics, either set the value of the IDL system variable `!P.FONT` equal to -1 (negative one), or set the `FONT` keyword of one of the Direct Graphics routines equal to -1. The vector font system is the default font system for IDL.

Once the vector font system is selected, use an embedded formatting command to select a vector font (or fonts) for each string. (See [“Embedded Formatting Commands”](#) on page 4433 for details on embedded formatting commands.) The font selected “sticks” from string to string; that is, if you change fonts in one string, future strings will use the new font until you change it again or exit IDL.

For example, to use the Duplex Roman vector font for the title of a plot, you would use a command that looks like this:

```
PLOT, mydata, TITLE="!5Title of my plot"
```

Consult *Using IDL* for details on using the vector font system with IDL Object Graphics.

Specifying Font Size

To specify the size of a vector font, use the `SET_CHARACTER_SIZE` keyword to the `DEVICE` procedure. The `SET_CHARACTER_SIZE` keyword takes a two-

element vector as its argument. The first element specifies the width of the “average” character in the font (in pixels) and calculates a scaling factor that determines the height of the characters. (It is not important what the “average” character is; it is used only to calculate a scaling factor that will be applied to all of the characters in the font.) The second element of the vector specifies the number of pixels between baselines of lines of text.

The ratio of the “average” character’s height to its width differs from font to font, so specifying the same value $[x, y]$ to the `SET_CHARACTER_SIZE` keyword may produce characters of different sizes in different fonts.

Note

While the first element of the vector specified to `SET_CHARACTER_SIZE` is technically a width, it is important to note that the width value has no effect on the widths of individual characters in the font. The width value is used only to calculate the appropriate scaling factor for the font.

For example, the following IDL commands display the word “Hello There” on the screen, in letters based on an “average” character that is 70 pixels wide, with 90 pixels between lines:

```
DEVICE, SET_CHARACTER_SIZE=[70,90]
XYOUTS, 0.1, 0.5, 'Hello!CThere'
```

You can also use the `CHARSIZE` keyword to the graphics routines or the `CHARSIZE` field of the IP System Variable to change the size of characters to a multiple of the size of the currently-selected character size. For example, to create characters one half the size of the current character size, you could use the following command:

```
XYOUTS, 0.1, 0.5, 'Hello!CThere', CHARSIZE=0.5
```

Note

Changing `CHARSIZE` adjusts both the character size and the space between lines.

ISO Latin 1 Encoding

The default font (Font 3, Simplex Roman) follows the ISO Latin 1 Encoding scheme and contains many international characters. The illustration of this font under “[Vector Font Samples](#)” on page 4445 can be used to find the octal codes for the special characters.

For example, suppose you want to display some text with an Angstrom symbol in it. Looking at the chart of font 3, we see that the Angstrom symbol has octal code 305. Non-printable characters can be represented in IDL using octal or hexadecimal

notation and the `STRING` function (see “[Representing Non-Printable Characters](#)” in Chapter 3 of the *Building IDL Applications* manual for details). So the Angstrom can be printed by inserting a `STRING(" 305B)` character in our text string as follows:

```
XYOUTS,.1, .5, 'Here is an Angstrom symbol: ' + STRING("305B), $  
/NORM, CHARSIZE=3
```

Customizing the Vector Fonts

The `EFONT` procedure is a widget application that allows you to edit the Hershey fonts and save the results. Use this routine to add special characters or completely new, custom fonts to the Hershey fonts.

About TrueType Fonts

A TrueType Font

Beginning with version 5.2, five TrueType font families are included with IDL. The fonts included are:

Font Family	Italic	Bold	BoldItalic
Courier	Courier Italic	Courier Bold	Courier Bold Italic
Helvetica	Helvetica Italic	Helvetica Bold	Helvetica Bold Italic
Monospace Symbol			
Times	Times Italic	Times Bold	Times Bold Italic
Symbol			

Table H-38: TrueType Font Names

When TrueType fonts are rendered on an IDL graphics device or destination object, the font outlines are first scaled to the proper size. After scaling, IDL converts the character outline information to a set of polygons using a triangulation algorithm. When text in a TrueType font is displayed, IDL is actually drawing a set of polygons calculated from the font information. This process has two side effects:

1. Computation time is used to triangulate and create the polygons. This means that you may notice a slight delay the first time you use text in a particular font and size. Once the polygons have been created, the information is cached by IDL and there is no need to re-triangulate each time text is displayed. Subsequent uses of the same font and size happen quickly.
2. Because the TrueType font outlines are converted into polygons, you may notice some chunkiness in the displayed characters, especially at small point sizes. The smoothness of the characters will vary with the quality of the TrueType font you are using, the point size, and the general smoothness of the font outlines.

Using TrueType Fonts

To use the TrueType font system with IDL Direct Graphics, either set the value of the IDL system variable `!P.FONT` equal to 1 (one), or set the `FONT` keyword to on one of the Direct Graphics routines equal to 1.

Once the TrueType font system is selected, use the `SET_FONT` keyword to the `DEVICE` routine to select the font to use. The value of the `SET_FONT` keyword is a *font name string*. The font name is the name by which IDL knows the font; the names of the TrueType fonts included with IDL are listed under [“About TrueType Fonts”](#) on page 4419. Finally, specify the `TT_FONT` keyword in the call to the `DEVICE` procedure. For example, to use Helvetica Bold Italic, use the following statement:

```
DEVICE, SET_FONT='Helvetica Bold Italic', /TT_FONT
```

To use Times Roman Regular:

```
DEVICE, SET_FONT='Times', /TT_FONT
```

IDL’s default TrueType font is 12 point Helvetica regular.

Specifying Font Size

To specify the size of a TrueType font, use the [SET_CHARACTER_SIZE](#) keyword to the `DEVICE` procedure. The `SET_CHARACTER_SIZE` keyword takes a two-element vector as its argument. The first element specifies the width of the “average” character in the font (in pixels) and calculates a scaling factor that determines the height of the characters. (It is not important what the “average” character is; it is used only to calculate a scaling factor that will be applied to all of the characters in the font.) The second element of the vector specifies the number of pixels between baselines of lines of text.

The ratio of the “average” character’s height to its width differs from font to font, so specifying the same value [*x*, *y*] to the `SET_CHARACTER_SIZE` keyword may produce characters of different sizes in different fonts.

Note

While the first element of the vector specified to `SET_CHARACTER_SIZE` is technically a width, it is important to note that the width value has no effect on the widths of individual characters in the font. The width value is used only to calculate the appropriate scaling factor for the font.

For example, the following IDL commands display the word “Hello There” on the screen in Helvetica Bold, in letters based on an “average” character that is 70 pixels wide, with 90 pixels between lines:

```
DEVICE, FONT='Helvetica Bold', /TT_FONT,
SET_CHARACTER_SIZE=[70,90]
XYOUTS, 0.1, 0.5, 'Hello!CThere'
```

You can also use the [CHARSIZE](#) keyword to the graphics routines or the [CHARSIZE](#) field of the !P System Variable to change the size of characters to a multiple of the size of the currently-selected character size. For example, to create characters one half the size of the current character size, you could use the following command:

```
XYOUTS, 0.1, 0.5, 'Hello!CThere', CHARSIZE=0.5
```

Note that changing the CHARSIZE adjusts both the character size and the space between lines.

Using Embedded Formatting Commands

Embedded formatting commands allow you to position text and change fonts within a single line of text. A subset of the embedded formatting commands available for use with the vector fonts are also available when using the TrueType font system. See [“Embedded Formatting Commands”](#) on page 4433 for a list of in-line formatting commands.

IDL TrueType Font Resource Files

The TrueType font system relies on a resource file named `ttfont.map`, located in the `resource/fonts/tt` subdirectory of the IDL directory. The format of the `ttfont.map` file is:

```
FontName      FileName      DirectGraphicsScale      ObjectGraphicsScale
```

where the fields in each column must be separated by white space (spaces and/or tabs). The fields contain the following information

The *Fontname* field contains the name that would be used for the `SET_FONT` keywords to the `DEVICE` routine.

The *Filename* field contains the name of the TrueType font file. On UNIX platforms, IDL will search for the file specified in the *FileName* field in the current directory (that is, in the `resource/fonts/tt` subdirectory of the IDL directory) if a bare filename is provided, or it will look for the file in the location specified by the fully-qualified file name if a complete path is provided. Because different platforms use different path-specification syntax, we recommend that you place any TrueType font files you wish to add to the `ttfont.map` file in the `resource/fonts/tt` subdirectory of the IDL directory. On Windows platforms, this entry may be '*', in which case the font will be loaded from the operating system font list, but that the following two scale entries will be honored.

The *DirectGraphicsScale* field contains a correction factor that will be applied when choosing a scale factor for the glyphs prior to being rendered on a Direct Graphics device. If you want the tallest character in the font to fit exactly within the vertical dimension of the device's current character size (as set via the `SET_CHARACTER_SIZE` keyword to the `DEVICE` procedure), set the scale factor equal to 1.0. Change the scale factor to a smaller number to scale a smaller portion of the tallest character into the character size.

For example, suppose the tallest character in your font is “Ã”. Setting the scale factor to 1.0 will scale this character to fit the current character size, and then apply the same scaling to all characters in the font. As a result, the letter “M” will fill only approximately 85% of the full height of the character size. To scale the font such that the height of the “M” fills the vertical dimension of the character size, you would include the value 0.85 in the scale field of the `ttfont.map` file.

The *ObjectGraphicsScale* field contains a correction factor that will be applied when choosing a scale factor for the glyphs prior to being rendered on a Object Graphics device. (This field works just like the *DirectGraphicsScale* field.) This scale factor should be set to 1.0 if the maximum ascent among all glyphs within a given font is to fit exactly within the font size (as set via the `SIZE` property to the `IDLgrFont` object).

Adding Your Own Fonts

To add your own font to the list of fonts known to IDL, use a text editor to edit the `ttfont.map` file, adding the *FontName*, *FileName*, *DirectGraphicsScale*, and *ObjectGraphicsScale* fields for your font. You will need to restart IDL for the changes to the `ttfont.map` file to take effect. On Windows systems, you can use fonts that are not mentioned in the `ttfont.map` file, as long as they are installed in the Fonts control panel, as described below.

Warning

If you choose to modify the `ttfont.map` file, be sure to keep a backup copy of the original file so you can restore the defaults if necessary. Note also that applications that use text may appear different on different platforms if the scale entries in the `ttfont.map` file have been altered.

Where IDL Searches for Fonts

The TrueType font files included with IDL are located in the `resource/fonts/tt` subdirectory of the IDL directory. When attempting to resolve a font name (specified via the `FONT` keyword to the `DEVICE` procedure), IDL will look in the

`ttfont.map` file first. If it fails to find the specified font file in the `ttfont.map` file, it will search for the font file in the following locations:

UNIX

No further search will be performed. If the specified font is not included in the `ttfont.map` file, IDL will substitute Helvetica.

Microsoft Windows

If the specified font is not included in the `ttfont.map` file, IDL will search the list of fonts installed in the system (the fonts installed in the Font control panel). If the specified font is not found, IDL will substitute Helvetica.

About Device Fonts

A PostScript Font

Device, or hardware, fonts are fonts that are provided directly by your system's hardware or by software other than IDL. In past releases of IDL, we have used the term "*hardware fonts*" extensively to discuss these types of fonts. This is because in the early days of IDL, computer displays were either text-only terminals or dedicated graphics display devices such as plotters or Tektronix graphics terminals. These graphics displays generally came with a set of fonts built-in; when IDL asked the device to display characters in a built-in font, it was making a request to the hardware to display those characters.

As computer displays have become more sophisticated, the concept of fonts provided "by the hardware" has expanded to include fonts provided by the computer's operating system, or by font-management software. For example, many computers now use font management software like Adobe Type Manager to manage the fonts made available by the operating system to all applications. We use the term "device font" to refer to a font that is available to one of IDL's graphics devices from a source *other than IDL itself*. (In this case, a "graphics device" can be either a Direct Graphics device as specified by the DEVICE routine or an Object Graphics "destination" such as a window or a printer.) While device fonts may include fonts only available because a particular piece of hardware knows how to draw characters in that font (a pen plotter is an example of a device that may still have its own special fonts), in most cases device fonts are fonts supplied by the operating system to any application that may want to use them.

Which Device Fonts Are Available?

To determine which device fonts are available on your system and the exact font strings to specify for each, use the [GET_FONTNAMES](#) keyword to the DEVICE procedure. You can also use an operating system specific method to determine which fonts are available:

UNIX

On most systems, the `xlsfonts` utility displays a list of fonts available to the operating system.

Microsoft Windows

Fonts available to the system are displayed in the Fonts control panel. You may also have other fonts available if you use font-management software such as Adobe Type Manager.

Using Device Fonts

To use the Device font system with IDL Direct Graphics, either set the value of the IDL system variable `!P.FONT` equal to 0 (zero), or set the `FONT` keyword to on one of the Direct Graphics routines equal to 0.

Once the Device font system is selected, use the `SET_FONT` keyword to the `DEVICE` routine to select the font to use. Because device fonts are specified differently on different platforms, the syntax of the *fontname* string depends on which platform you are using.

UNIX

Usually, the window system provides a directory of font files that can be used by all applications. List the contents of that directory to find the fonts available on your system. The size of the font selected also affects the size of vector drawn text. On some machines, fonts are kept in subdirectories of `/usr/lib/X11/fonts`. You can use the `xlsfonts` command to list available X Windows fonts.

For example, to select the font 8X13:

```
!P.FONT = 0
DEVICE, SET_FONT = '8X13'
```

Microsoft Windows

The `SET_FONT` keyword should be set to a string with the following form:

```
DEVICE, SET_FONT="font*modifier1*modifier2*...modifiern"
```

where the asterisk (*) acts as a delimiter between the font's name (*font*) and any modifiers. The string is *not* case sensitive. Modifiers are simply “keywords” that change aspects of the selected font. Valid modifiers are:

- For font weight: THIN, LIGHT, BOLD, HEAVY

- For font quality: DRAFT, PROOF
- For font pitch: FIXED, VARIABLE
- For font angle: ITALIC
- For strikeout text: STRIKEOUT
- For underlined text: UNDERLINE
- For font size: Any number is interpreted as the font height in pixels.

For example, if you have Garamond installed as one of your Windows fonts, you could select 24-pixel cell height Garamond italic as the font to use in plotting. The following commands tell IDL to use hardware fonts, change the font, and then make a simple plot:

```
!P.FONT = 0
DEVICE, SET_FONT = "GARAMOND*ITALIC*24"
PLOT, FINDGEN(10), TITLE = "IDL Plot"
```

This feature is compatible with TrueType and Adobe Type Manager (and, possibly, other type scaling programs for Windows). If you have TrueType or ATM installed, the TrueType or PostScript outline fonts are used so that text looks good at any size.

Fonts and the PostScript Device

A special set of device fonts are available when the current Direct Graphics device is PS (PostScript). IDL includes font metric information for 35 standard PostScript fonts, and can create PostScript language files that include text in these fonts. (The 35 fonts known to IDL are listed in the following table; they the standard fonts included in memory in the vast majority of modern PostScript printers.) The PostScript font metric files (*.afm files) are located in the `resource/fonts/ps` subdirectory of the IDL directory.

AvantGarde-Book	Helvetica-Narrow-Oblique
AvantGarde-BookOblique	Helvetica-Oblique
AvantGarde-Demi	NewCenturySchlbk-Bold
AvantGarde-DemiOblique	NewCenturySchlbk-BoldItalic
Bookman-Demi	NewCenturySchlbk-Italic
Bookman-DemiItalic	NewCenturySchlbk-Roman

Table H-39: Names of Supported PostScript Fonts

Bookman-Light	Palatino-Bold
Bookman-LightItalic	Palatino-BoldItalic
Courier	Palatino-Italic
Courier-Bold	Palatino-Roman
Courier-BoldOblique	Symbol
Courier-Oblique	Times-Bold
Helvetica	Times-BoldItalic
Helvetica-Bold	Times-Italic
Helvetica-BoldOblique	Times-Roman
Helvetica-Narrow	ZapfChancery-MediumItalic
Helvetica-Narrow-Bold	ZapfDingats
Helvetica-Narrow-BoldOblique	

Table H-39: Names of Supported PostScript Fonts (Continued)

Using PostScript Fonts

To use a PostScript font in your Direct Graphics output, you must first specify that IDL use the device font system, they switch to the PS device, then choose a font using the SET_FONT keyword to the DEVICE procedure.

The following IDL commands choose the correct font system, set the graphics device, select the font Palatino Roman, open a PostScript file to print to, plot a simple data set, and close the PostScript file:

```
!P.FONT = 0
SET_PLOT, 'PS'
DEVICE, SET_FONT = 'Palatino-Roman', FILE = 'testfile.ps'
PLOT, INDGEN(10), TITLE = 'My Palatino Title'
DEVICE, /CLOSE
```

Note

Subsequent PostScript output will continue to use the font Palatino Roman until you explicitly change the font again, or exit IDL.

You can also specify PostScript fonts using a set of keywords to the DEVICE procedure. The keyword combinations for the fonts included with IDL are listed in the following table.

PostScript Font	DEVICE Keywords
Courier	/COURIER
Courier Bold	/COURIER, /BOLD
Courier Oblique	/COURIER, /OBLIQUE
Courier Bold Oblique	/COURIER, /BOLD, /OBLIQUE
Helvetica	/HELVETICA
Helvetica Bold	/HELVETICA, /BOLD
Helvetica Oblique	/HELVETICA, /OBLIQUE
Helvetica Bold Oblique	/HELVETICA, /BOLD, /OBLIQUE
Helvetica Narrow	/HELVETICA, /NARROW
Helvetica Narrow Bold	/HELVETICA, /NARROW, /BOLD
Helvetica Narrow Oblique	/HELVETICA, /NARROW, /OBLIQUE
Helvetica Narrow Bold Oblique	/HELVETICA, /NARROW, /BOLD, /OBLIQUE
ITC Avant Garde Gothic Book	/AVANTGARDE, /BOOK
ITC Avant Garde Gothic Book Oblique	/AVANTGARDE, /BOOK, /OBLIQUE
ITC Avant Garde Gothic Demi	/AVANTGARDE, /DEMI
ITC Avant Garde Gothic Demi Oblique	/AVANTGARDE, /DEMI, /OBLIQUE
ITC Bookman Demi	/BKMAN, /DEMI
ITC Bookman Demi Italic	/BKMAN, /DEMI, /ITALIC
ITC Bookman Light	/BKMAN, /LIGHT
ITC Bookman Light Italic	/BKMAN, /LIGHT, /ITALIC

Table H-40: The Standard 35 PostScript Fonts

PostScript Font	DEVICE Keywords
ITC Zapf Chancery Medium Italic	/ZAPFCHANCERY, /MEDIUM, /ITALIC
ITC Zapf Dingbats	/ZAPFDINGBATS
New Century Schoolbook	/SCHOOLBOOK
New Century Schoolbook Bold	/SCHOOLBOOK, /BOLD
New Century Schoolbook Italic	/SCHOOLBOOK, /ITALIC
New Century Schoolbook Bold Italic	/SCHOOLBOOK, /BOLD, /ITALIC
Palatino	/PALATINO
Palatino Bold	/PALATINO, /BOLD
Palatino Italic	/PALATINO, /ITALIC
Palatino Bold Italic	/PALATINO, /BOLD, /ITALIC
Symbol	/SYMBOL
Times	/TIMES
Times Bold	/TIMES, /BOLD
Times Italic	/TIMES, /ITALIC
Times Bold Italic	/TIMES, /ITALIC, /BOLD

Table H-40: The Standard 35 PostScript Fonts (Continued)

For example to use the PostScript font Palatino Bold Italic, you could use either of the following DEVICE commands:

```
DEVICE, SET_FONT = 'Palatino*Bold*Italic'
DEVICE, /PALATINO, /BOLD, /ITALIC
```

Changing the PostScript Font Assigned to an Index

You can change the PostScript font assigned to a given font index using the [FONT_INDEX](#) keyword to the DEVICE procedure. Font indices and their use are discussed in [“Embedded Formatting Commands”](#) on page 4433.

Changing the font index assigned to a font can be useful when changing PostScript fonts in the middle of a text string. For example, the following statements map Palatino Bold Italic to font index 4, and then output text using that font and the Symbol font:

```
; Map the font selected by !4 to be PalatinoBoldItalic:
DEVICE, /PALATINO, /BOLD, /ITALIC, FONT_INDEX=4
; Output "Alpha :" in PalatinoBoldItalic followed by an
; Alpha character:
XYOUTS, .3, .5, /NORMAL, "!4Alpha: !9a", FONT=0, SIZE=5.0
```

Adding Your Own PostScript Fonts

Because the 35 PostScript fonts included with IDL are built in to a PostScript printer's memory, the IDL distribution includes only the font metric files, which provide positioning information. In addition, the `.afm` files used by IDL are specially processed to provide the information in a format IDL expects.

You can add your own PostScript fonts to the list of fonts known to IDL if you have access to the PostScript font file (usually named `font.pfb`) to load into your printer and to the `font.afm` file supplied by Adobe. You can convert the standard `.afm` file into a file IDL understands using the IDL routine [PSAFM](#). Consult the file `README.TXT` in the `resource/fonts/ps` subdirectory of the IDL directory for details on adding PostScript fonts to your system.

Choosing a Font Type

Some of the issues involved in choosing between vector, TrueType, and device fonts are explained below.

Appearance

Vector-drawn characters are of medium quality, suitable for most uses. TrueType characters are of relatively high quality, although at some point sizes the triangulation process (described in [“About TrueType Fonts”](#) on page 4419) may cause characters to appear slightly jagged. The appearance of device characters varies from mediocre (characters found in many graphics terminals) to publication quality (PostScript).

Three-Dimensional Transformations

Vector or TrueType fonts should always be used with three-dimensional transformations. Both vector and TrueType characters pass through the same transformations as the rest of the plot, yielding a better looking plot. See [“Three-Dimensional Graphics”](#) in Chapter 18 of the *Using IDL* manual for an example of vector-drawn characters with three-dimensional graphics. Device characters are not subject to the three-dimensional transforms.

Portability

The vector-drawn fonts work using any graphics device and look the same on every device (within the limitations of device resolution) on any system supported by IDL.

TrueType fonts are available only on the X, WIN, PRINTER, PS, and Z Direct Graphics devices, and in IDL's Object Graphics system. If you use only the fonts supplied with IDL, TrueType fonts also look the same on every supported device (again within the limits of the device resolution). If you use TrueType fonts other than those supplied with IDL, your font may not be installed on the system which runs your program. In this case, IDL will substitute a known font for the missing font.

The appearance, size, and availability of device fonts varies greatly from device to device. Many, if not most, of the positioning and font changing commands recognized by the vector-drawing routines are ignored when using device fonts. The exception to this rule is the Direct Graphics PS device; if you use one of the PostScript fonts supported by IDL, the PostScript output from the PS device will be identical between platforms.

Computational Time

Device fonts are generally rendered the most quickly, since the hardware device or operating system handles all computations and caching.

It takes more computer time to draw characters with line vectors and generally results in more input/output. However, this is not an important issue unless the plot contains a large number of characters or the transmission link to the device is very slow.

The initial triangulation step used when displaying TrueType fonts for the first time can be computationally expensive. However, since the font shapes are cached, subsequent uses of the same font are relatively speedy.

Flexibility

Vector-drawn fonts provide a great deal of flexibility. There are many different typefaces available, as shown in the tables at the end of this chapter. In addition, such fonts can be arbitrarily scaled, rotated, and transformed.

TrueType fonts support fewer embedded formatting commands than do the vector fonts, and cannot be scaled, rotated, or transformed.

The abilities of hardware-generated characters differ greatly between devices so it is not possible to make a blanket statement on when they should be used—the best font to use depends on the available hardware. In general, however, the vector or TrueType fonts are easier to use and often provide superior results to what is available from the hardware. See the discussion of the device you are using in [Appendix A, “IDL Graphics Devices”](#) for details on the hardware-generated characters provided by that device.

Print Quality

For producing publication-quality output, we recommend using either the TrueType font system or the Direct Graphics PS device and one of the PostScript fonts supported by IDL.

Embedded Formatting Commands

When you use vector, TrueType, and some device fonts, text strings can include embedded formatting commands that facilitate subscripting, superscripting, and equation formatting. The method used is similar to that developed by Grandle and Nystrom (1980). Embedded formatting commands are always introduced by the exclamation mark, (!). (The string “! !” is used to produce a literal exclamation mark.)

Note

Embedded formatting commands prefaced by the exclamation mark have no special significance for hardware-generated characters unless the ability is provided by the particular device in use. The IDL PostScript device driver accepts many of the standard embedded formatting commands, and is described here. If you wish to use hardware fonts with IDL Direct Graphics devices other than the PostScript device, consult the description of the device in [Appendix A, “IDL Graphics Devices”](#) before trying to use these commands with hardware characters.

You can determine whether embedded formatting commands are available for use with device fonts on your current graphics device by inspecting bit 12 of the *Flags* field of the [!D System Variable](#). Use the IDL statement:

```
IF (!D.FLAGS AND 4096) NE 0 THEN PRINT, 'Bit is set.'
```

to determine whether bit 12 of the *Flags* field is set for the current graphics device.

Changing Fonts within a String

You can change fonts one or more times within a text string using the embedded font commands shown in the table below. The character following the exclamation mark can be either upper or lower case.

Examples of commands used to change fonts in mid-string are included in [“Formatting Command Examples”](#) on page 4437.

Command	Vector Font	TrueType Font	PostScript Font
!3	Simplex Roman (default)	Helvetica	Helvetica
!4	Simplex Greek	Helvetica Bold	Helvetica Bold
!5	Duplex Roman	Helvetica Italic	Helvetica Narrow

Table H-41: Embedded Font Selection Commands

Command	Vector Font	TrueType Font	PostScript Font
!6	Complex Roman	Helvetica Bold Italic	Helvetica Narrow Bold Oblique
!7	Complex Greek	Times	Times Roman
!8	Complex Italic	Times Italic	Times Bold Italic
!9	Math/special characters	Symbol	Symbol
!M	Math/special characters (change effective for one character only)	Symbol	Symbol
!10	Special characters	Symbol *	Zapf Dingbats
!11(!G)	Gothic English	Courier	Courier
!12(!W)	Simplex Script	Courier Italic	Courier Oblique
!13	Complex Script	Courier Bold	Palatino
!14	Gothic Italian	Courier Bold Italic	Palatino Italic
!15	Gothic German	Times Bold	Palatino Bold
!16	Cyrillic	Times Bold Italic	Palatino Bold Italic
!17	Triplex Roman	Helvetica *	Avant Garde Book
!18	Triplex Italic	Helvetica *	New Century Schoolbook
!19		Helvetica *	New Century Schoolbook Bold
!20	Miscellaneous	Helvetica *	Undefined User Font
!X	Revert to the entry font	Revert to the entry font	Revert to the entry font

* The font assigned to this index may be replaced in a future release of IDL.

Table H-41: Embedded Font Selection Commands (Continued)

Positioning Commands

The positioning and other font-manipulation commands are described in the following table. Examples of commands used to position text are included in [“Formatting Command Examples”](#) on page 4437.

Command	Action
!A	Shift above the division line .
!B	Shift below the division line .
!C	“Carriage return,” begins a new line of text. Shift back to the starting position and down one line. This command also performs an implicit “!N” command, returning to the normal level and character size at the beginning of the new line.
!D	Shift down to the first level subscript and decrease the character size by a factor of 0.62.
!E	Shift up to the exponent level and decrease the character size by a factor of 0.44.
!I	Shift down to the index level and decrease the character size by a factor of 0.44.
!L	Shift down to the second level subscript. Decrease the character size by a factor of 0.62.
!N	Shift back to the normal level and original character size.
!R	Restore position. The current position is set from the top of the saved positions stack.
!S	Save position. The current position is saved on the top of the saved positions stack.
!U	Shift to upper subscript level. Decrease the character size by a factor of 0.62.
!X	Return to the entry font.

Table H-42: Vector-Drawn Positioning and Miscellaneous Commands

Command	Action
!Z(u_0, u_1, \dots, u_n)	Display one or more character glyphs according to their unicode value. Each u_i within the parentheses will be interpreted as a 16-bit hexadecimal unicode value. If more than one unicode value is to be included, the values should be separated by commas.
!!	Display the ! symbol.

Table H-42: Vector-Drawn Positioning and Miscellaneous Commands

Formatting Command Examples

The figure below illustrates the relative positions and effects on character size of the level commands. In this figure, the letters “!N” are normal level and size characters.

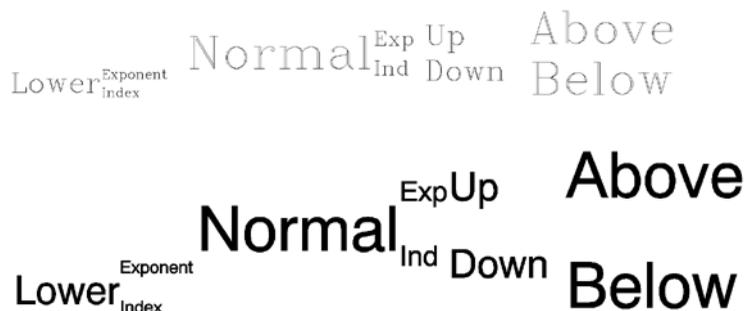


Figure H-9: Positioning Commands With Vector Fonts (Top) and TrueType Fonts (Bottom)

The positioning shown was created with the following command:

```
XYOUTS, 0.1, 0.3, $
'!LLower!S!EExponent!R!IIndex!N Normal!S!EExp!R!IInd!N!S!U Up
!R!D Down!N!S!A Above!R!B Below'
```

Note that the string argument to the XYOUTS procedure must be entered on a single line rather than the two lines shown above.

A Complex Equation

Embedded positioning commands and the vector font system can be used to create the integral shown below:

$$\int_p^x \rho_i U_i^2 dx$$

Figure H-10: An integral created with the vector fonts.

The command string used to produce the integral is:

```
XYOUTS, 0, .2, $
  '!MI!S!A!E!8x!R!B!Ip!N !7q!Ii!N!8U!S!E2!R!Ii!NdX', $
  SIZE = 3, /NORMAL
```

Remember that the case of the letter in an embedded command is not important. The string may be broken down into the following components:

!MI

Changes to the math set and draws the integral sign, uppercase I.

!S

Saves the current position on the position stack.

!A!E!8x

Shifts above the division line and to the exponent level, switches to the Complex Italic font (Font 8), and draws the “x.”

!R!B!lp

Restores the position to the position immediately after the integral sign, shifts below the division line to the index level, and draws the “*p*.”

!N !7q

Returns to the normal level, advances one space, shifts to the Complex Greek font (Font 7), and draws the Greek letter rho, which is designated by “*q*” in this set.

!li!N

Shifts to the index level and draws the “*i*” at the index level. Returns to the normal level.

!8U

Shifts to the Complex Italic font (Font 8) and outputs the upper case “*U*.”

!S!E2

Saves the position and draws the exponent “2.”

!R!li

Restores the position and draws the index “*i*.”

!N dx

Returns to the normal level and outputs “*dx*.”

Note

The equation shown in the figure above could not be created so simply using the TrueType font system, because the large integral symbol is broken into two or more characters in the TrueType fonts.

Vector-Drawn Font Example

IDL uses vector-drawn font when the value of the system variable !P.FONT is -1. This is the default condition. Initially, all characters are drawn using the Simplex Roman font (Font 3). When plotting, font changing commands may be embedded in the title strings keyword arguments (XTITLE, YTITLE, and TITLE) to select other fonts. For example, the following statement uses the Complex Roman font (Font 6) for the *x*-axis title:

```
PLOT, X, XTITLE = '!6X Axis Title'
```

This font remains in effect until explicitly changed. The order in which the annotations are drawn is main title, x -axis numbers, x -axis title, y -axis numbers, and y -axis title. Strings to be output also may contain embedded information selecting subscripting, superscripting, plus other features that facilitate equation formatting.

The following statements were used to produce the figure below. They serve as an example of a plot using vector-drawn characters and of equation formatting.

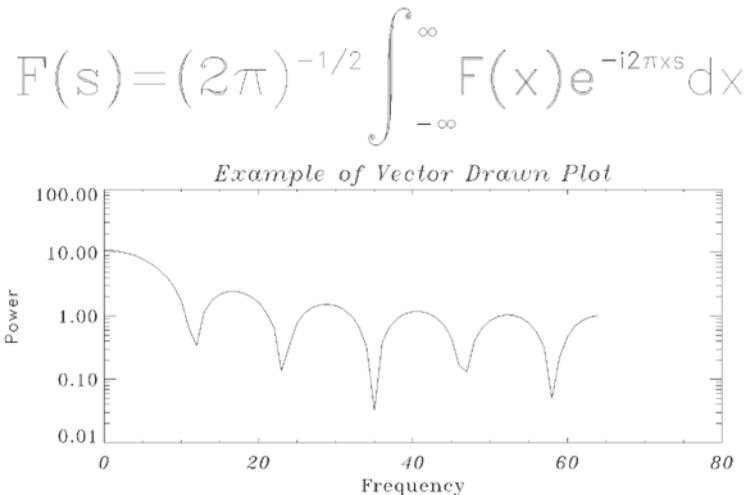


Figure H-11: Example of a Vector-drawn Plot

```
; Define an array:
X = FLTARR(128)
; Make a step function:
X[30:40] = 1.0
; Take FFT and magnitude:
X = ABS(FFT(X, 1))
; Produce a log-linear plot. Use the Triplex Roman font for the
; x title (!17), Duplex Roman for the y title (!5), and Triplex
; Italic for the main title (!18). The position keyword is used to
; shrink the plotting window:
PLOT, X[0:64], /YLOG, XTITLE = '!17Frequency', $
      YTITLE = '!5Power', $
      TITLE = '!18Example of Vector Drawn Plot', $
      POSITION = [.2, .2, .9, .6]
SS = '!6F(s) = (2!4p)!e-1/2!n !mi!s!a!e!m $
      !r!b!i ' + '-!m $
```

```
; String to produce equation:
      !nF(x)e !e-i2!4p!3xs!ndx'
XYOUTS, 0.1, .75, SS, SIZE = 3, $
; Output string over plot. The NOCLIP keyword is needed because
; the previous plot caused the clipping region to shrink:
      /NORMAL, /NOCLIP
```

TrueType Font Samples

The following figures show roman versions of the four TrueType font families included with IDL. The character sets for the bold, italic, and bold italic versions of these fonts are the same as the roman versions.

The SHOWFONT command was used to create these figures. For example, to display the following figure on the screen, you would the command:

```
SHOWFONT, 'Helvetica', 'Helvetica', /TT_FONT
```

For more information, see [“SHOWFONT”](#) on page 1914.

Note

The following font charts are numbered in octal notation. To read the octal number of a character, add the column index (along the top) to ten times the row index. For example, the capital letter “A” is octal 101, and the copyright symbol is octal 251.

Font Helvetica

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04y		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06y	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10y	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12y	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
14x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
20x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
22x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
24x		ı	ç	£	¤	¥	ı	\$	"	©	®	«	¬	-	®	-
26x	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
30x	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
32x	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
34x	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
36x	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Font Times

Decimal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	
14x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
20x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
22x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
24x		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
26x	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
30x	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
32x	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
34x	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
36x	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Font Courier

Decimal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	
14x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	-	□
20x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
22x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
24x		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
26x	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
30x	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
32x	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
34x	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
36x	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Font Symbol

Decimal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	∇	#	∃	‰	&	∫	()	#	+	,	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	≡	A	B	X	Δ	E	Φ	Γ	Π	I	∂	K	Λ	M	N	O
12x	Π	⊖	P	Σ	T	Y	s	Ω	Ξ	Ψ	Z		∴		⊥	
14x		α	β	χ	δ	ε	φ	γ	η	ι	φ	κ	λ	μ	ν	ο
16x	π	θ	ρ	σ	τ	υ	ω	ω	ξ	ψ	ξ	{		}	~	□
20x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
22x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
24x	□	Y	'	≤	/	∞	f	♣	♦	♥	♠	↔	←	↑	→	↓
26x	°	±	"	≥	×	α	∂	•	÷	≠	≡	≈	...		—	↩
30x	ℵ	ℶ	ℷ	ℸ	⊗	⊕	⊖	∩	∪	⊃	⊄	⊆	⊇	∈	∉	
32x	∠	∇	®	©	™	∏	√	·	¬	∧	∨	⊕	←	↑	→	↓
34x	◇	<	®	©	™	Σ	/									
36x	⍓	>	ℓ	ℓ												□

Vector Font Samples

The following figures show samples of various vector-drawn fonts. The SHOWFONT command was used to create these figures. For example, to display the following figure on the screen, you would use the command:

```
SHOWFONT, 3, 'Simplex Roman'
```

To output this figure to a postscript file, you would use the following commands:

```
SET_PLOT, 'PS'
SHOWFONT, 3, 'Simplex Roman'
DEVICE, /CLOSE
```

For more information, see “[SHOWFONT](#)” on page 1914.

Note

The following font charts are numbered in octal notation. To read the octal number of a character, add the column index (along the top) to ten times the row index. For example, the capital letter “A” is octal 101, and the “\$” symbol is octal 44.

Font 3, Simplex Roman

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	^	

Font 4, Simplex Greek

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M	N	Ξ	O
12x	Π	P	Σ	T	Τ	Φ	Χ	Ψ	Ω	∞	?	[\]	^	_
14x	'	α	β	γ	δ	ε	ξ	η	θ	ι	κ	λ	μ	ν	ξ	ο
16x	π	ρ	σ	τ	υ	φ	χ	ψ	ω	∞	?	[\]	^	_

Font 5, Duplex Roman

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	[\]	^	_

Font 6, Complex Roman

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	[\]	^	_
20x																
22x																
24x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
26x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
30x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
32x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
34x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
36x	p	q	r	s	t	u	v	w	x	y	z	[\]	^	_

Font 7, Complex Greek

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M	N	Ξ	O
12x	Π	P	Σ	T	Υ	Φ	X	Ψ	Ω	∞	ι	[\]	^	_
14x	'	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
16x	π	ρ	σ	τ	υ	φ	χ	ψ	ω	∞	ι	[\]	^	_

Font 8, Complex Italic

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06x		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
14x		'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x		p	q	r	s	t	u	v	w	x	y	z	[\]	^	_

Font 9, Math and Special

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	
04x			"		∞	°	§	'	()	*	±	,	∓	•	÷	
06x		∩	∪	∩	∪	←	↓	→	↑		≡		{	≠	}	∞	
10x		∞	~	□	✓	∂	∃	∫	∇	∫	∫				≡	†	
12x		∅		√	√	∴	♠	♠	×				[]	^	---
14x		'	∠	≥	α	∂	∈	♀	∩	∫	j		≤	♂	⊙	‡	
16x		p	q	√	∫	∂	♥	♠	♠	⊥			[]	^	---

Font 11, Gothic English

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	'	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	~	~
14x	'	a	b	r	d	e	f	g	h	i	i	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	~

Font 12, Simplex Script

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	^	°
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	^	°
20x																
22x																
24x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
26x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
30x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
32x	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	^	°
34x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
36x	p	q	r	s	t	u	v	w	x	y	z	{		}	^	°

Font 13, Complex Script

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	'	#	\$	%	&	'	()	*	+	.	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	:	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	^	°
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	^	°
20x																
22x																
24x		!	'	#	\$	%	&	'	()	*	+	.	-	.	/	
26x	0	1	2	3	4	5	6	7	8	9	:	:	<	=	>	?
30x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
32x	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	^	°
34x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
36x	p	q	r	s	t	u	v	w	x	y	z	{		}	^	°

Font 14, Gothic Italian

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	'	#	\$	%	&	'	()	*	+	.	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	:	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	{		}	~	~
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	~

Font 15, Gothic German

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	'	#	\$	%	&	'	()	*	+	.	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	s	u	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	p	q	r	s	t	u	v	w	x	y	z	{	ß	}	þ	~
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{	ß	}	þ	~

Font 16, Cyrillic

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	Ю Ъ	\$	Э	&	'	()	*	+	.	-	.	/		
06x	0	1	2	3	4	5	6	7	8	9	:	я	ъ	=	ы	?
10x	ь	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О
12x	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ы	Э	Ь	Ю	Я
14x	'	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о
16x	п	р	с	т	у	ф	х	ц	ч	ш	щ	ы	э	ь	ю	я
20x																
22x																
24x		!	Ю Ъ	\$	Э	&	'	()	*	+	.	-	.	/		
26x	0	1	2	3	4	5	6	7	8	9	:	я	ъ	=	ы	?
30x	ь	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О
32x	П	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ы	Э	Ь	Ю	Я
34x	'	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о
36x	п	р	с	т	у	ф	х	ц	ч	ш	щ	ы	э	ь	ю	я

Font 17, Triplex Roman

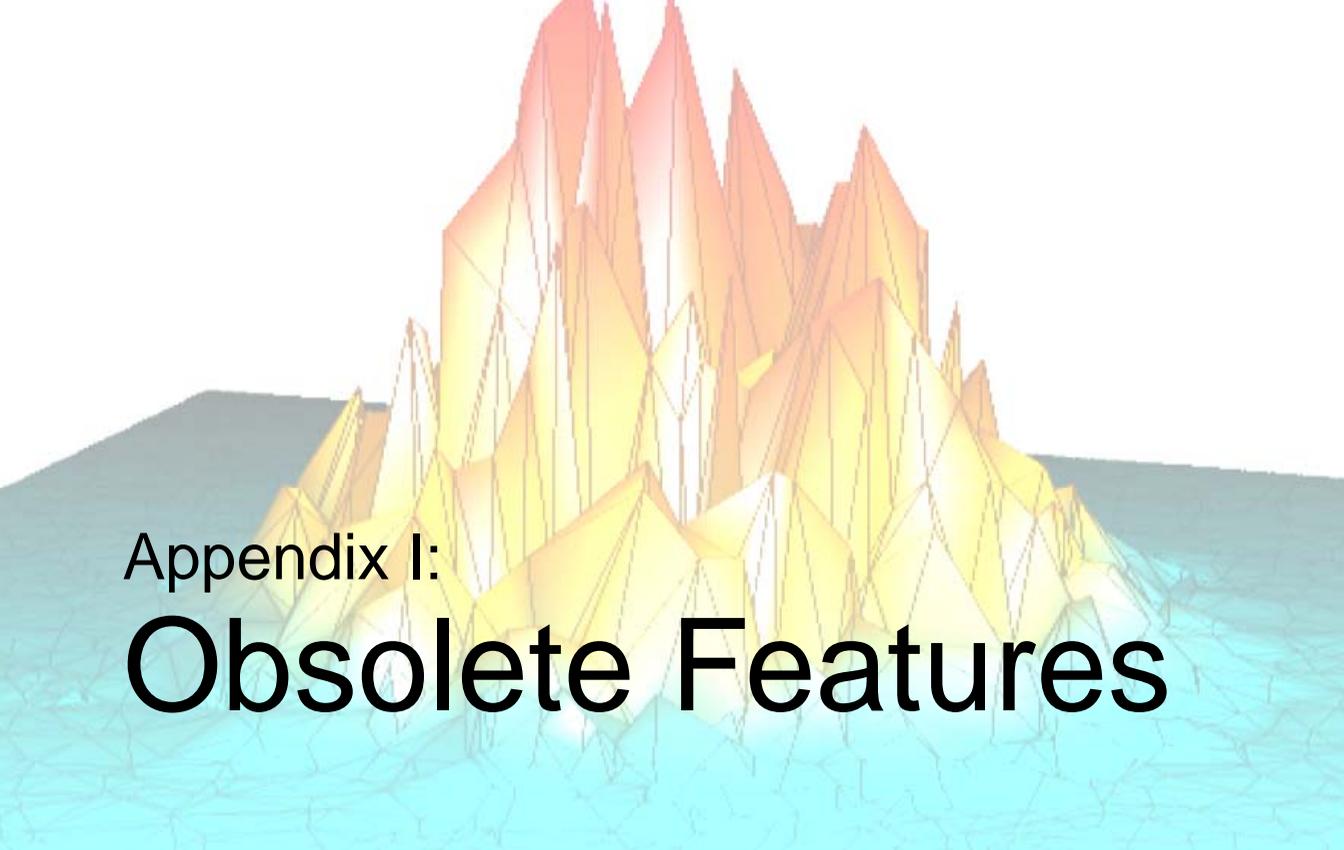
Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W	X	Y	Z	{	\	}	^	°
14x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w	x	y	z	{	\	}	^	°

Font 18, Triplex Italic

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06x	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>:</i>	<i>;</i>	<i><</i>	<i>=</i>	<i>></i>	<i>?</i>
10x	@	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>	<i>J</i>	<i>K</i>	<i>L</i>	<i>M</i>	<i>N</i>	<i>O</i>
12x	<i>P</i>	<i>Q</i>	<i>R</i>	<i>S</i>	<i>T</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	{	\	}	^	°
14x	'	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>l</i>	<i>m</i>	<i>n</i>	<i>o</i>
16x	<i>p</i>	<i>q</i>	<i>r</i>	<i>s</i>	<i>t</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>	{	\	}	^	°

Font 20, Miscellaneous

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x	∞	S	∫	@:	-		~	∞	h	b	*	o	γ	o	♩	#
06x	∞	Ω	m	↗	∞	≡	✕	.	∫	∫	▲	o	6	-	Γ	ℝ
10x	^	•	•	▲	▼	★	↓	×	♁	♁	♁	†	♁	♁	♁	♁
12x	○	○	□	⊕	h	Ψ	☾	*	∞	∞	∞		=	♩	∇	}
14x	∞	♩	■	◀	▶	↑	+	∞	∞	∞	∞	∞	∞	∞	∞	∞
16x	○	○	♩	4	♩	E	♩	∞	∞	∞	∞		=	♩	∇	}



Appendix I: Obsolete Features

The following topics are covered in this appendix:

What Are Obsolete Features?	4456	Features Obsolete in IDL 5.2	4464
Features Obsolete in IDL 6.1	4457	Features Obsolete in IDL 5.1	4465
Features Obsolete in IDL 6.0	4458	Features Obsolete in IDL 5.0	4466
Features Obsolete in IDL 5.6	4459	Features Obsolete in IDL 4.0 or Earlier	4467
Features Obsolete in IDL 5.5	4460	Obsolete Arguments and Keywords	4473
Features Obsolete in IDL 5.4	4461	Obsolete System Variables	4479
Features Obsolete in IDL 5.3	4462	Obsolete Graphics Devices	4481
SDF Routines Obsolete in IDL 5.3	4463		

What Are Obsolete Features?

To improve the overall quality and functionality of IDL, Research Systems, Inc. occasionally replaces existing features with new, improved features. In many cases, existing routines are improved without changing their existing behavior—through improvements of the underlying algorithms, for example, or by adding keyword functionality. In some cases, however, the improved methods are incompatible with the old. In these situations, we consider the routines that we have replaced to be *obsolete*.

For similar reasons, arguments and keywords to existing routines may occasionally become obsolete. In many cases, the functionality activated by the obsolete argument or keyword can be achieved more efficiently using a new argument or keyword. If this is the case, existing code using the obsolete argument or keyword will generally continue to function as it always has, although performance may be better using the new, replacement argument or keyword. In cases where the obsolete argument or keyword activated functionality that is no longer available (perhaps as a result of an operating system upgrade or change), IDL will either quietly ignore the presence of the argument or keyword (if doing so does not affect the resulting output) or warn that an invalid parameter has been specified.

This chapter lists the features that have become obsolete in each version of IDL. These features are not documented in the online help. To view reference information for features obsoleted in IDL version 5.0 and later, see the `obsolete.pdf` file in the `docs` subdirectory of the IDL distribution. Features obsoleted in IDL 4.0 and earlier are not documented in the `obsolete.pdf` file. If a `.pro` file for the routine exists, it is located in the `lib/obsolete` subdirectory of the IDL distribution. You can read the documentation header of a routine in the `obsolete` directory either by opening the `.pro` file or using the `DOC_LIBRARY` routine.

Features Obsoleted in IDL 6.1

The following features were present in IDL Version 6.0 but became obsolete in IDL Version 6.1.

Routine	Replaced by	.pro File?
FINDFILE	FILE_SEARCH	
MSG_CAT_CLOSE	IDLffLangCat	
MSG_CAT_COMPILE	IDLffLangCat	
MSG_CAT_OPEN	IDLffLangCat	
IDLffLanguageCat	IDLffLangCat	

Table I-43: Features Obsoleted in IDL 6.1

Features Obsoleted in IDL 6.0

The following features were present in IDL Version 5.6 but became obsolete in IDL Version 6.0.

For information about routines replaced by the iTools system, see the *iTools User's Guide*.

Routine	Replaced by	.pro File?
LIVE_CONTOUR	ICONTOUR	
LIVE_CONTROL	iTools system	
LIVE_DESTROY	iTools system	
LIVE_EXPORT	iTools system	
LIVE_IMAGE	IIMAGE	
LIVE_INFO	iTools system	
LIVE_LINE	iTools system	
LIVE_LOAD	iTools system	
LIVE_OPLOT	IPLOT	
LIVE_PLOT	IPLOT	
LIVE_PRINT	iTools system	
LIVE_RECT	iTools system	
LIVE_STYLE	iTools system	
LIVE_SURFACE	ISURFACE	
LIVE_TEXT	iTools system	

Table I-44: Features Obsoleted in IDL 6.0

Features Obsoleted in IDL 5.6

The following features were present in IDL Version 5.5 but became obsolete in IDL Version 5.6.

Routine	Replaced by	.pro File?
HDF_VD_GETNEXT	HDF_VG_GETNEXT	
VAX_FLOAT	VAX_FLOAT keyword to OPEN	

Table I-45: Features Obsoleted in IDL 5.6

Features Obsoleted in IDL 5.5

The following features were present in IDL Version 5.4 but became obsolete in IDL Version 5.5.

Routine	Replaced by	.pro File?
DELETE_SYMBOL	n/a	
DELLOG	n/a	
DO_APPLE_SCRIPT	n/a	
ERRORF	ERF	
GET_SYMBOL	n/a	
LJLCT	n/a	
REWIND	n/a	
SET_SYMBOL	n/a	
SETLOG	n/a	
SKIPF	n/a	
TAPRD	n/a	
TAPWRT	n/a	
TRNLOG	n/a	
WEOF	n/a	

Table I-46: Features Obsoleted in IDL 5.5

Features Obsoleted in IDL 5.4

The following features were present in IDL Version 5.3 but became obsolete in IDL Version 5.4.

Routine	Replaced by	.pro File?
POLYFITW	POLY_FIT , MEASURE_ERRORS keyword	<code>polyfitw.pro</code>
RIEMANN	RADON	

Table I-47: Features Obsoleted in IDL 5.4

Features Obsoleted in IDL 5.3

The following features were present in IDL Version 5.2 but became obsolete in IDL Version 5.3.

Routine	Replaced by	.pro File?
HDF_DFSD_* Routines	HDF_SD_* Routines	
RSTRPOS	STRPOS , /REVERSE_SEARCH	rstrpos.pro
STR_SEP	STRSPLIT for single character delimiters STRSPLIT , /REGEX for longer delimiters	str_sep.pro

Table I-48: Features Obsoleted in IDL 5.3

SDF Routines Obsoleted in IDL 5.3

HDF_DFSD_* routines have been obsoleted in IDL 5.3.

What is DFSD and Why Are We Obsoleting It?

DFSD is an SD (Scientific Data Model). DFSD is the older, single-file SD form. The newer SD format, MFSD, is referred to in the IDL API as HDF_SD_*. New IDL code should use HDF_SD_* routines rather than HDF_DFSD_* routines.

Version HDF4.1r2 of HDF has obsoleted the DFSD interface, somewhat forcing us to do so as well. IDL 5.3 uses HDF4.1r3 (Version 4.1, revision 3). It is recommended that users convert their old HDF files to the 4.1r3 format.

The following HDF routines have been obsoleted in IDL 5.3.

- [HDF_DFSD_ADDDATA](#)
- [HDF_DFSD_DIMGET](#)
- [HDF_DFSD_DIMSET](#)
- [HDF_DFSD_ENDSLICE](#)
- [HDF_DFSD_GETDATA](#)
- [HDF_DFSD_GETINFO](#)
- [HDF_DFSD_GETSLICE](#)
- [HDF_DFSD_PUTSLICE](#)
- [HDF_DFSD_READREF](#)
- [HDF_DFSD_SETINFO](#)
- [HDF_DFSD_STARTSLICE](#)

Features Obsoleted in IDL 5.2

The following features were present in IDL Version 5.1 but became obsolete in IDL Version 5.2.

Routine	Replaced by	.pro File?
DEMO_MODE	LMGR	demo_mode.pro

Table I-49: Features Obsoleted in IDL 5.2

Features Obsoleted in IDL 5.1

The following features were present in IDL Version 5.0 but became obsolete in IDL Version 5.1.

Routine	Replaced by	.pro File?
SLICER	SLICER3	slicer3.pro

Table I-50: Features Obsoleted in IDL 5.1

Features Obsoleted in IDL 5.0

The following features were present in IDL Version 4.0 but became obsolete in IDL Version 5.0.

Routine	Replaced by	.pro File?
DDE Routines	n/a	
GETHELP	OUTPUT keyword to HELP	
HANDLE_CREATE	PTR_NEW	
HANDLE_FREE	PTR_FREE	
HANDLE_INFO	PTR_VALID	
HANDLE_MOVE	n/a	
HANDLE_VALUE	dereference operator	
INP, INPW, OUTP, OUTPW	n/a	
PICKFILE	DIALOG_PICKFILE	
Old RPC API	New RPC API	
.SIZE Executive Command	No longer needed	
TIFF_DUMP	n/a	
TIFF_READ	READ_TIFF	
TIFF_WRITE	WRITE_TIFF	
WIDED	n/a	
WIDGET_MESSAGE	DIALOG_MESSAGE	

Table I-51: Features Obsoleted in IDL 5.0

Features Obsoleted in IDL 4.0 or Earlier

The following features became obsolete in IDL version 4.0 or earlier. These features are not documented in the `obsolete.pdf` file. If a `.pro` file for the routine exists, it is located in the `obsolete` subdirectory of the `lib` directory of the IDL distribution. You can read the documentation header of a routine in the `obsolete` directory either by opening the `.pro` file or using the `DOC_LIBRARY` routine.

Routine	Replaced by	.pro File?
ADDSYSVAR	DEFSYSV	<code>addsysvar.pro</code>
ADJCT	XPALETTE	<code>adjct.pro</code>
ANOVA	KW_TEST	<code>anova.pro</code>
ANOVA_UNEQUAL	KW_TEST	<code>anova_unequal.pro</code>
BETAI	IBETA	<code>betai.pro</code>
C_EDIT	XPALETTE	<code>c_edit.pro</code>
CALL_VMS	CALL_EXTERNAL	
CHI_SQR	CHISQR_CVF	<code>chi_sqr.pro</code>
CHI_SQR1	CHISQR_PDF	<code>chi_sqr1.pro</code>
COLOR_EDIT	XPALETTE	<code>color_edit.pro</code>
CONTINGENT	CTI_TEST	<code>contingent.pro</code>
CORREL_MATRIX	CORRELATE	<code>correl_matrix.pro</code>
COSINES	n/a	<code>cosines.pro</code>
CW_BSELECTOR	WIDGET_DROPLIST	<code>cw_bselector.pro</code>
CW_LOADSTATE	NO_COPY keyword to WIDGET_CONTROL	<code>cw_loadstate.pro</code>
CW_SAVESTATE	NO_COPY keyword to WIDGET_CONTROL	<code>cw_savestate.pro</code>
DIFFEQ_23	RK4	<code>diffeq_23.pro</code>

Table I-52: Features Obsoleted in IDL 4.0 or Earlier

Routine	Replaced by	.pro File?
DIFFEQ_45	RK4	diffeq_23.pro
DISP_TEXT	XYOUTS	disp_text.pro
EIGEN_II	EIGENVEC	eigen_ii.pro
EQUAL_VARIANCE	FV_TEST	equal_variance.pro
F_TEST	F_CVF	f_test.pro
F_TEST1	F_PDF	f_test1.pro
FILLCONTOUR	FILL keyword to CONTOUR	fillcontour.pro
FORRD	READU	
FORRD_KEY	READU	
FORWRT	WRITEU	
FRIEDMAN	KW_TEST	friedman.pro
GAUSS	GAUSS_CVF	gauss.pro
GOODFIT	XSQ_TEST	goodfit.pro
HELP_VM	MEMORY keyword to HELP	help_vm.pro
HSV_TO_RGB	COLOR_CONVERT	hsv_to_rgb.pro
JOIN	CLUSTER	join.pro
KMEANS	CLUSTER	kmeans.pro
KRUSKAL_WALLIS	KW_TEST	kruskal_wallis.pro
LATLON	n/a	latlon.pro
LEGO	LEGO keyword to SURFACE	lego.pro
LISTREP	n/a	listrep.pro
LISTWISE	n/a	listwise.pro

Table I-52: Features Obsolete in IDL 4.0 or Earlier (Continued)

Routine	Replaced by	.pro File?
LN03	n/a	ln03.pro
LUBKSB	LUSOL	
LUDCMP	LUDC	
MAKETREE	CLUSTER	maketree.pro
MANN_WHITNEY	RS_TEST	mann_whitney.pro
MENUS	WIDGET_DROPLIST, etc.	menus.pro
MIPSEB_DBLFIXUP	n/a	mipseb_dblfixup.pro
MOVIE	XINTERANIMATE	movie.pro
MPROVE	LUMPROVE	
MULTICOMPARE	Hypothesis Testing Routines	multicompare.pro
NR_BETA	BETA	
NR_BROYDN	BROYDEN	
NR_CHOLDC	CHOLDC	
NR_CHOLSL	CHOLSOL	
NR_DFPMIN	DFPMIN	
NR_ELMHES	ELMHES	nr_elmhes.pro
NR_EXPINT	EXPINT	
NR_FULSTR	FULSTR	
NR_HQR	HQR	nr_hqr.pro
NR_INVERT	INVERT	
NR_LINBCG	LINBCG	
NR_LUBKSB	LUSOL	nr_lubksb.pro
NR_LUDCMP	LUDC	nr_ludcmp.pro
NR_MACHAR	MACHAR	

Table I-52: Features Obsoleted in IDL 4.0 or Earlier (Continued)

Routine	Replaced by	.pro File?
NR_MPROVE	LUMPROVE	
NR_NEWT	NEWTON	
NR_POWELL	POWELL	
NR_QROMB	QROMB	
NR_QROMO	QROMO	
NR_QSIMP	QSIMP	
NR_RK4	RK4	
NR_SPLINE	SPL_INIT	
NR_SPLINT	SPL_INTERP	
NR_SPRSAB	SPRSAB	
NR_SPRSAX	SPRSAX	
NR_SPRSIN	SPRSIN	nr_sprsin.pro
NR_SVBKSB	SVSOL	nr_svbksb.pro
NR_SVD	SVDC	nr_svd.pro
NR_TQLI	TRIQL	
NR_TRED2	TRIRED	
NR_TRIDAG	TRISOL	
NR_WTN	WTN	nr_wtn.pro
NR_ZROOTS	FZ_ROOTS	
ONLY_8BIT	n/a	only_8bit.pro
PALETTE	XPALETTE	palette.pro
PARTIAL2_COR	P_CORRELATE	partial2_cor.pro
PARTIAL_COR	P_CORRELATE	partical_cor.pro
PHASER	n/a	phaser.pro

Table I-52: Features Obsoleted in IDL 4.0 or Earlier (Continued)

Routine	Replaced by	.pro File?
PM	n/a	pm.pro
PMF	n/a	pmf.pro
POLYCONTOUR	FILL keyword to CONTOUR	polycontour.pro
PROMPT	n/a	prompt.pro
PWIDGET	n/a	pwidget.pro
REGRESS1	REGRESS	regress1.pro
REGRESSION	REGRESS	regression.pro
RGB_TO_HSV	COLOR_CONVERT	rgb_to_hsv.pro
RM	n/a	rm.pro
RMF	n/a	rmf.pro
ROT_INT	ROT	rot_int.pro
RSI_GAMMAI	IGAMMA	rsi_gamma.pro
RUNS_TEST	R_TEST	runs_test.pro
SET_NATIVE_PLOT	n/a	set_native_plot.pro
SET_SCREEN	n/a	set_screen.pro
SET_VIEWPORT	n/a	set_viewport.pro
SET_XY	n/a	set_xy.pro
SIGMA	MOMENT	sigma.pro
SIGN_TEST	S_TEST	sign_test.pro
SIMPSON	QSIMP or QROMB	simpson.pro
SPEARMAN	R_CORRELATE	spearman.pro
STDEV	MOMENT	stdev.pro
STEPWISE	REGRESS	stepwise.pro
STUDENT1_T	T_PDF	student1_t.pro

Table I-52: Features Obsolete in IDL 4.0 or Earlier (Continued)

Routine	Replaced by	.pro File?
STUDENT_T	T_CVF	student_t.pro
STUDRANGE	T_PDF	studrange.pro
SURFACE_FIT	SFIT	surface_fit.pro
SVBKS	SVSOL	
SVD	SVDC	
TESTCONTRAST	n/a	testcontrast.pro
TQLI	TRIQL	
TRED2	TRIRED	
TRIDAG	TRISOL	
TVDELETE	WDELETE	
TVRDC	CURSOR	
TVSET	WSET	
TVSHOW	WSHOW	
TVWINDOW	WINDOW	
VMSCODE	n/a	vmrcode.pro
WILCOXON	RS_TEST	wilcoxon.pro
WMENU	WIDGET_DROPLIST, etc.	wmenu.pro
XANIMATE	XINTERANIMATE	xanimate.pro
XBACKREGISTER	TIMER keyword to WIDGET_CONTROL	xbackregister.pro
XDL	n/a	xdl.pro
XMANAGERTOOL	XMTOOL	xmanagertool.pro
XMENU	WIDGET_DROPLIST, etc.	xmenu.pro
XPDMENU	WIDGET_DROPLIST, etc.	xpdmenu.pro
ZROOTS	FZ_ROOTS	

Table I-52: Features Obsolete in IDL 4.0 or Earlier (Continued)

Obsolete Arguments and Keywords

The following arguments and keywords are obsolete and should not be used in new IDL code.

Routine	Argument or Keyword	Description
BYTEORDER	DTOGFLOAT keyword	This keyword was available only on the VMS platform
	GFLOATTOD keyword	This keyword was available only on the VMS platform.
CALL_EXTERNAL	DEFAULT keyword	This keyword was ignored on non-VMS platforms.
	PORTABLE keyword	This keyword was ignored on non-VMS platforms.
	VAX_FLOAT keyword	This keyword was available only on the VMS platform.
DEVICE	DEPTH keyword	This keyword was available only on the VMS platform.
	FONT keyword	This keyword was replaced by the SET_FONT keyword.
DOC_LIBRARY	FILE keyword	This keyword was ignored on non-VMS platforms.
	PATH keyword	This keyword was ignored on non-VMS platforms.
	OUTPUTS keyword	This keyword was available only on the VMS platform.
EXTRACT_SLICE	CUBIC keyword	EXTRACT_SLICE does not support cubic interpolation.
IDLgrMPEG::Save	CREATOR_TYPE keyword	This keyword was ignored on non-Macintosh OS 9 (and earlier) platforms.

Table I-53: Obsolete Arguments and Keywords

Routine	Argument or Keyword	Description
IDLgrVolume::Init	CUTTING_PLANES	This keyword has been replaced by CLIP_PLANES.
IDLITSYS_CREATE_TOOL	PANEL_LOCATION	This keyword is no longer honored. Panel location is handled via standard widget layout mechanisms.
IDLitVisualization::Add	GROUP keyword	This keyword was unintentionally and needlessly exposed.
IDLitVisualization::GetCenterRotation	DATA keyword	This keyword is not implemented.
IDLitVisualization::GetProperty	GROUP_PARENT keyword	This keyword was unintentionally and needlessly exposed.
IVolume	CUTTING_PLANES	This keyword has been replaced by CLIP_PLANES.
LINKIMAGE	DEFAULT keyword	This keyword was ignored on non-VMS platforms.
LIVE_PRINT	SETUP keyword	This keyword was ignored on non-Macintosh OS 9 (and earlier) platforms.
MAKE_DLL	VAX_FLOAT keyword	This keyword was ignored on non-VMS platforms.
ONLINE_HELP	HTML_HELP keyword	This keyword is no longer necessary; ONLINE_HELP parses the value of the BOOK keyword to determine file type.

Table I-53: Obsolete Arguments and Keywords (Continued)

Routine	Argument or Keyword	Description
OPEN	<i>Record_Length</i> argument	This argument was ignored on non-VMS platforms.
	BINARY keyword	Windows file I/O changed to function like UNIX file I/O.
	BLOCK keyword	This keyword was ignored on non-VMS platforms.
	DEFAULT keyword	This keyword was ignored on non-VMS platforms.
	EXTENDSIZE keyword	This keyword was ignored on non-VMS platforms.
	FIXED keyword	This keyword was ignored on non-VMS platforms.
	FORTRAN keyword	This keyword was ignored on non-VMS platforms.
	INITIALSIZE keyword	This keyword was ignored on non-VMS platforms.
	KEYED keyword	This keyword was ignored on non-VMS platforms.

Table I-53: Obsolete Arguments and Keywords (Continued)

Routine	Argument or Keyword	Description
<i>OPEN</i> , continued	LIST keyword	This keyword was ignored on non-VMS platforms.
	MACCREATOR keyword	This keyword was ignored on non-Macintosh OS 9 (and earlier) platforms.
	MACTYPE keyword	This keyword was ignored on non-Macintosh OS 9 (and earlier) platforms.
	NOAUTOMODE keyword	Windows file I/O changed to function like UNIX file I/O.
	NONE keyword	This keyword was ignored on non-VMS platforms.
	NOSDTIO	Keyword was renamed RAWIO.
	PRINT keyword	This keyword was ignored on non-VMS platforms.
	SEGMENTED keyword	This keyword was ignored on non-VMS platforms.
	SHARED keyword	This keyword was ignored on non-VMS platforms.
	STREAM keyword	This keyword was ignored on non-VMS platforms.
	SUBMIT keyword	This keyword was ignored on non-VMS platforms.
	SUPERSEDE keyword	This keyword was ignored on non-VMS platforms.
TRUNCATE_ON_CLOSE keyword	This keyword was ignored on non-VMS platforms.	

Table I-53: Obsolete Arguments and Keywords (Continued)

Routine	Argument or Keyword	Description
OPEN, <i>continued</i>	UDF_BLOCK keyword	This keyword was ignored on non-VMS platforms.
	VARIABLE keyword	This keyword was ignored on non-VMS platforms.
PRINT/PRINTF	REWRITE keyword	This keyword was ignored on non-VMS platforms.
READ_TIFF	ORDER keyword	This keyword was replaced by the ORIENTATION keyword
	UNSIGNED keyword	This keyword became obsolete when IDL began supporting an unsigned 16-bit integer data type.
READ/READF	KEY_ID keyword	This keyword was ignored on non-VMS platforms.
	KEY_MATCH keyword	This keyword was ignored on non-VMS platforms.
	KEY_VALUE keyword	This keyword was ignored on non-VMS platforms.
READU	KEY_ID keyword	This keyword was ignored on non-VMS platforms.
	KEY_MATCH keyword	This keyword was ignored on non-VMS platforms.
	KEY_VALUE keyword	This keyword was ignored on non-VMS platforms.
SAVE	XDR keyword	IDL always writes XDR files.

Table I-53: Obsolete Arguments and Keywords (Continued)

Routine	Argument or Keyword	Description
SPAWN	FORCE keyword	This keyword was incorrectly included in the SPAWN documentation.
	MACCREATOR keyword	This keyword was ignored on non-Macintosh OS 9 (and earlier) platforms.
	NOCLISYM keyword	This keyword was ignored on non-VMS platforms.
	NOLOGNAM keyword	This keyword was ignored on non-VMS platforms.
	NOTIFY keyword	This keyword was ignored on non-VMS platforms.
WIDGET_BASE	APP_MBAR keyword	This keyword was ignored on non-Macintosh OS 9 (and earlier) platforms.
WRITE_TIFF	<i>Order</i> argument	The <i>Order</i> argument has been replaced by the ORIENTATION keyword. Code that uses the <i>Order</i> argument will continue to work as before, but new code should use the ORIENTATION keyword instead.
WRITEU	REWRITE keyword	This keyword was ignored on non-VMS platforms.
XMANAGER	BACKGROUND keyword	This keyword was replaced by the TIMER keyword to WIDGET_CONTROL.
	MODAL keyword	This keyword was replaced by the MODAL keyword to WIDGET_BASE.

Table I-53: Obsolete Arguments and Keywords (Continued)

Obsolete System Variables

The following IDL system variables became obsolete in the change from VAX IDL (IDL version 1) to IDL version 2. While it is highly unlikely that you will find references to these system variables in existing code, we include them here because they are flagged when the OBS_SYSVARS field of the !WARN structure is set equal to one. See [Appendix D, “System Variables”](#) for information on IDL system variables.

System Variable	Replaced by
!BCOLOR	BOTTOM keyword to SURFACE
!COLOR	!P.COLOR
!CXMAX	!X.CRANGE[1]
!CXMIN	!X.CRANGE[0]
!CYMAX	!Y.CRANGE[1]
!CYMIN	!Y.CRANGE[0]
!FANCY	No direct equivalent. Use !P.FONT and !P.CHARSIZE
!FLIP	No equivalent.
!GRID	!P.TICKLEN
!HI	No equivalent.
!IGNORE	!P.NOCLIP
!LINETYPE	!P.LINESTYLE
!LO	No equivalent.
!MTITLE	!P.TITLE
!NOERAS	!P.NOERASE
!NORMALCONT	FOLLOW keyword to CONTOUR
!NSUM	!P.NSUM

Table I-54: Obsolete System Variables

System Variable	Replaced by
!PSYM	!P.PSYM
!SC1	!P.POSITION[0] * !D.X_VSIZE if !P.POSITION[2] is nonzero, or !X.WINDOW[0] * !D.X_VSIZE otherwise.
!SC2	!P.POSITION[2] * !D.X_VSIZE if !P.POSITION[2] is nonzero, or !X.WINDOW[1] * !D.X_VSIZE otherwise.
!SC3	!P.POSITION[1] * !D.X_VSIZE if !P.POSITION[2] is nonzero, or !Y.WINDOW[0] * !D.X_VSIZE otherwise.
!SC4	!P.POSITION[3] * !D.X_VSIZE if !P.POSITION[2] is nonzero, or !Y.WINDOW[1] * !D.X_VSIZE otherwise.
!TERM	DEVICE procedure.
!TYPE	!X.TYPE, !X.STYLE, !Y.TYPE, !Y.STYLE, !P.TICKLEN
!XMAX	!X.RANGE[1]
!XMIN	!X.RANGE[0]
!XTICKS	!X.TICKS
!XTITLE	!X.TITLE
!YMAX	!Y.RANGE[1]
!YMIN	!Y.RANGE[0]
!YTICKS	!Y.TICKS
!YTITLE	!Y.TITLE

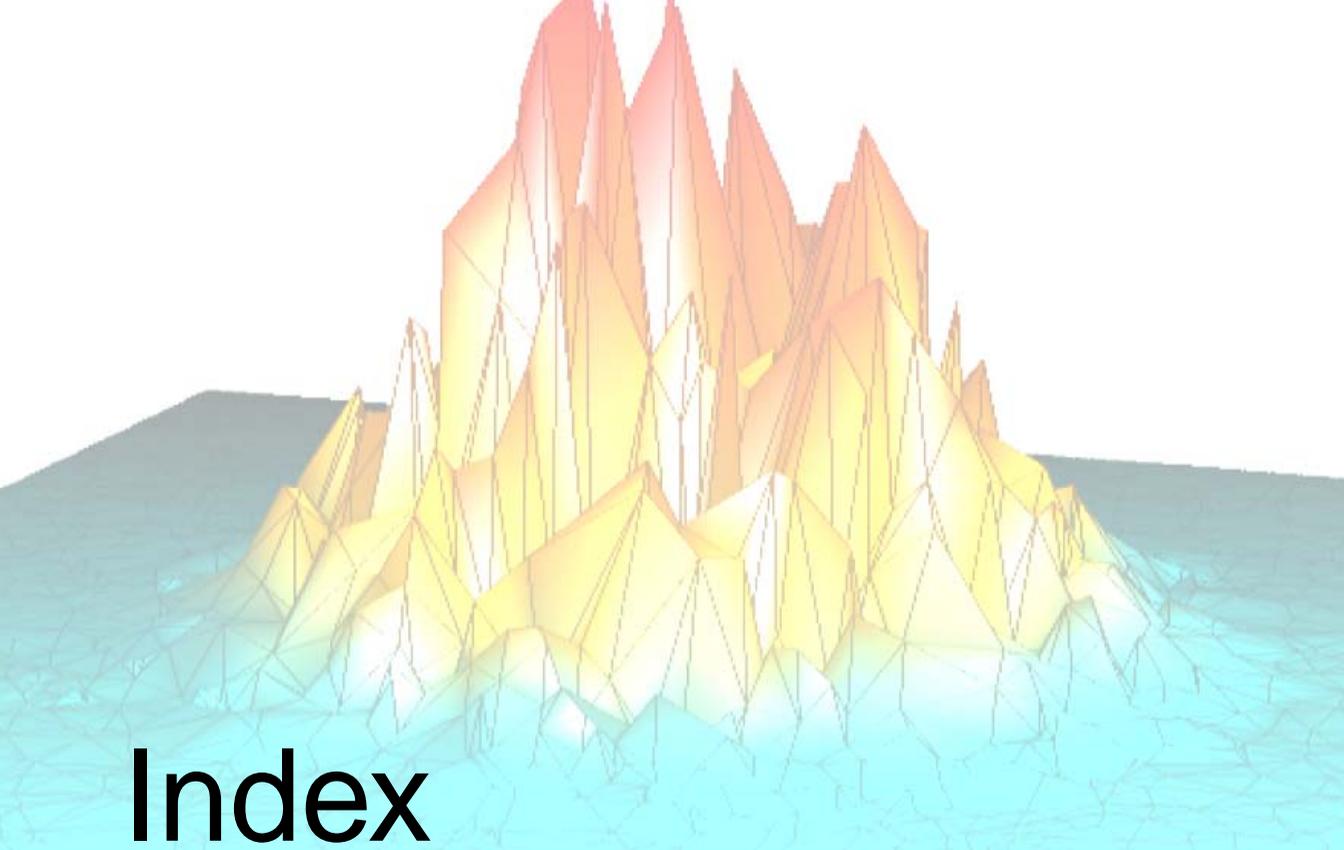
Table I-54: Obsolete System Variables (Continued)

Obsolete Graphics Devices

The following arguments and keywords are obsolete and should not be used in new IDL code.

Graphics Device	Description
The LJ Device (LJ)	The LJ device was only available in IDL for VMS.
The Macintosh Device (MAC)	The MAC device was only available in IDL for Macintosh versions running under OS 9 and earlier. Versions of IDL for Macintosh running under OS X and later use the X device.

Table I-55: Obsolete Graphics Devices



Index

Symbols

- , [4396](#), [4397](#)
- ! character, [4406](#)
- !C system variable, [4375](#)
- !CPU system variable, [4363](#)
- !D system variable, [4375](#)
- !D.TABLE_SIZE system variable
 - array scale, [2178](#)
 - graphic variable, [4377](#)
- !D.WINDOW system variable
 - WDELETE procedure, [2238](#)
 - WINDOW procedure, [2503](#)
 - WSET procedure, [2550](#)
- !DIR system variable, [4365](#)
- !DLM_PATH system variable, environment variable, [4365](#)
- !DPI system variable, [4357](#)
- !DTOR system variable, [4357](#)
- !EDIT_INPUT system variable, environment variable, [4367](#)
- !ERR system variable
 - error handling, [4359](#)
 - WHERE function, [2243](#)
- !ERROR_STATE system variable
 - error handling variable, [4359](#)
 - I/O errors, [1498](#)
 - MSG field, [2037](#)
 - retrieving errors, [2037](#)
- !EXCEPT system variable, [4361](#)
- !HELP_PATH system variable
 - environment variable, [4367](#)
 - path expansion, [635](#)
- !JOURNAL system variable
 - environment variable, [4367](#)
 - JOURNAL procedure, [1110](#)

- !MAKE_DLL system variable, [4368](#)
- !MAP system variable
 - constant variable, [4357](#)
 - coordinate conversion, [1353](#)
- !MORE system variable, [4370](#)
- !MOUSE system variable
 - CURSOR procedure, [371](#)
 - structure information, [4361](#)
- !ORDER system variable, [4379](#)
- !P system variable, [4379](#)
- !P.COLOR system variable, [2611](#)
- !P.FONT system variable, [4415](#)
- !P.MULTI system variable, [4308](#)
- !P.T system variable
 - adjusting, [458](#)
 - creating 3-D view, [359](#)
 - SURFR procedure, [2064](#)
 - T3D procedure, [2089](#)
 - transformation matrix, [4342](#)
- !P.T3D system variable, [359](#)
- !PATH system variable
 - environment variable, [4371](#)
 - expansion, [635](#)
- !PI system variable, [4357](#)
- !PROMPT system variable, [4372](#)
- !QUIET system variable
 - environment variable, [4372](#)
 - error notification, [1426](#)
- !RADEG system variable, [4357](#)
- !VALUES system variable, [4357](#)
- !VERSION system variable, [4372](#)
- !WARN system variable, [4362](#)
- !X system variable, [4383](#)
- !Y system variable, [4383](#)
- !Z system variable, [4383](#)
- # operator, [4395](#)
- ## operator, [4395](#)
- \$ character, line continuation, [4407](#)
- & character, [4408](#)
- && operator, [4396](#)
- ' character, [4407](#)
- * character, [4408](#)
- . character, [4407](#)
- .COMPILE command, [67](#)
- .CONTINUE command, [68](#)
- .EDIT command, [69](#)
- .FULL_RESET_SESSION command, [70](#)
- .GO command, [71](#)
- .idl directory, [105](#)
- .OUT command, [72](#)
- .RESET_SESSION command, [73](#)
- .RETURN command, [75](#)
- .RNEW command, [76](#)
- .RUN command, [78](#)
- .sid image files, [2835](#)
- .SKIP command, [80](#)
- .STEP command, [82](#)
- .STEPOVER command, [83](#)
- .TRACE command, [84](#)
- .Xdefaults file, [2266](#)
- : character, [4408](#)
- ; character, [4407](#)
- < operator, [4394](#)
- > operator, [4394](#)
- ? character, starting online help, [4409](#)
- ?: ternary operator, [4401](#)
- @ character, [4408](#)
- || operator, [4396](#)
- ~ operator, [4396](#)
- '' character, [4407](#)
- / character, in keywords, [2668](#)

Numerics

- 2-D rendering of 3-D volumes, [1614](#)
- 3D
 - coordinate transformations, [336](#)
 - drawing transformation, [458](#)
 - images
 - reconstructed from 2D arrays, [1775](#)
 - viewing coordinate system, [359](#)
 - rendering, [1406](#)

- 3D (*continued*)
 - transformations
 - SCALE3D procedure, [1844](#)
 - SURFR procedure, [2064](#)
 - VERT_T3D function, [2218](#)
 - volume slices, [1936](#)
 - 3D plots, viewing, [2612](#)
 - 3D transforms, implementing, [2089](#)
 - 64-bit integer
 - array creation, [1127](#)
 - data type, converting to, [1266](#)
 - LONG64ARR arrays, [1260](#)
 - vectors, [1260](#)
- A**
- A_CORRELATE function, [86](#)
 - ABS function, [88](#)
 - absolute deviation, [1443](#)
 - absolute value, [88](#)
 - accelerators, disable, [2298](#), [2357](#), [2427](#), [2466](#), [2483](#)
 - ACOS function, [90](#)
 - active command line, [2585](#)
 - ActiveX controls, creating in IDL widget hierarchies, [4215](#)
 - ADAPT_HIST_EQUAL function, [92](#)
 - addition, array elements, [2118](#)
 - addition operator, [4392](#)
 - AddPolygon method, IDLgrTessellator, [4025](#)
 - ADDSYSVAR, *see* obsolete routines
 - adjacency list, Delaunay triangulation, [2132](#)
 - ADJCT, *see* obsolete routines
 - Adobe
 - Font Metrics files, [1619](#)
 - Type Manager
 - device fonts, [4426](#)
 - setting font, [4277](#)
 - Aitoff map projection, [1355](#)
 - Alber's equal area conic map projection, [1355](#)
 - aligning text (XYOUTS), [2656](#)
 - allocated memory, returning amount of, [863](#)
 - ALOG function, [95](#)
 - ALOG10 function, [97](#)
 - alpha channel, [3716](#)
 - AMOEBAS function, [99](#)
 - ampersand, [4408](#)
 - analysis objects
 - IDLanRIOGroup, [2708](#)
 - IDLanROI class, [2680](#)
 - AND operator, [4397](#)
 - Angstrom symbol, [4417](#)
 - animation
 - closing MPEG files, [1466](#)
 - flickering images, [734](#)
 - MPEG frame storage, [1471](#)
 - opening MPEG files, [1467](#)
 - saving MPEG files, [1473](#)
 - widgets (CW_ANIMATE), [384](#)
 - widgets (XINTERANIMATE), [2570](#)
 - XVOLUME, [2644](#)
 - ANNOTATE procedure, [103](#)
 - annotations, of displayed images, [103](#)
 - ANOVA, *see* obsolete routines
 - ANOVA_UNEQUAL, *see* obsolete routines
 - apostrophe, [4407](#)
 - APP_USER_DIR function, [105](#)
 - APP_USER_DIR_QUERY function, [114](#)
 - application user directory, [105](#)
 - approximating models, statistical, [283](#)
 - arc-cosine, [90](#)
 - architecture, current version in use, [4372](#)
 - arc-sine, [132](#)
 - arc-tangent, [137](#)
 - ARG_PRESENT function, [119](#)
 - arguments
 - checking existence of, [119](#)
 - described, [61](#)
 - arguments, described, [2667](#)
 - array operators
 - CHOLDC, [255](#)
 - CHOLSOL, [257](#)

array operators (*continued*)

- COND, 300
- CRAMER, 352
- DETERM, 535
- EIGENVEC, 599
- ELMHES, 602
- GS_ITER, 833
- HQR, 890
- INVERT, 1016
- LA_CHOLDC, 1129
- LA_CHOLMPROVE, 1132
- LA_CHOLSOL, 1136
- LA_DETERM, 1139
- LA_EIGENPROBLEM, 1141
- LA_EIGENVEC, 1153
- LA_ELMHES, 1157
- LA_HQR, 1163
- LA_INVERT, 1166
- LA_LUDC, 1178
- LA_LUMPROVE, 1181
- LA_LUSOL, 1184
- LA_SVD, 1187
- LA_TRIDC, 1191
- LA_TRIQL, 1199
- LA_TRIRED, 1202
- LA_TRISOL, 1204
- LU_COMPLEX, 1273
- LUDC, 1275
- LUMPROVE, 1277
- LUSOL, 1280
- NORM, 1484
- SVDC, 2065
- SVSOL, 2074
- TRIQL, 2146
- TRIRED, 2149
- TRISOL, 2151
- ARRAY_EQUAL function, 121
- ARRAY_INDICES function, 123
- arrays
 - arbitrary resizing, 302
 - changing dimensions of, 1783

arrays (*continued*)

- comparing to scalars, 121
- comparing values, 121
- concatenation, 4400
- converting subscripts, 123
- creating
 - 64-bit integer
 - (L64INDGEN function), 1127
 - (LON64ARR function), 1260
 - any type (MAKE_ARRAY function), 1288
 - byte
 - (BINDGEN function), 178
 - (BYTARR function), 197
 - complex
 - (CINDGEN function), 259
 - (COMPLEXARR function), 295
 - (DCINDGEN function), 495
 - (DCOMPLEXARR function), 500
 - double-precision
 - (DBLARR function), 493
 - (DCINDGEN function), 495
 - (DCOMPLEXARR function), 500
 - (DINDGEN function), 575
 - integer
 - (INDGEN function), 990
 - (INTARR function), 1002
 - longword
 - (LINDGEN function), 1227
 - (LONARR function), 1262
 - single-precision, floating-point
 - (FINDGEN function), 725
 - (FLTARR function), 741
 - string
 - (SINDGEN function), 1923
 - (STRARR function), 2013
 - unsigned 64-bit, (ULON64ARR function), 2191
 - unsigned 64-bit integer, (UL64INDGEN function), 2187
 - unsigned integer, (UINDGEN function), 2181

- arrays (*continued*)
 - creating
 - unsigned longword
 - (ULINDGEN function), 2189
 - (ULONARR function), 2193
 - data type, determining type, SIZE function, 1926
 - extracting sub-arrays, 643
 - filling with a scalar value, 1795
 - finding number of elements in, 1475
 - floating-point, 725
 - incrementing elements, 878
 - interactive editing tool (XVAREEDIT procedure), 2642
 - of structures, creating, 1796
 - operators, *see* array operators
 - resizing, 1771
 - resizing 2D, 633
 - returning
 - maximum value, 1370
 - minimum value, 1430
 - subscripts of non-zero elements, 2242
 - reversing indices, 1810
 - rotating, 1820
 - searching in 2D, 1858
 - searching in 3D, 1861
 - shifting elements, 1887
 - size, 1926
 - sorting, 1970
 - subscripts, returning non-zero elements, 2242
 - summing elements, 2118
 - transposing, 2126
 - unique elements of (UNIQ function), 2199
 - updating, 182
- ARROW procedure, 126
- ASCII_TEMPLATE function, 128
- ASIN function, 132
- assignment operator, 4400
- assignment operators (compound), 4401
- ASSOC, function, 134
- associated variables, 134
- asterisk, 4408
- at sign (character), 4408
- ATAN function, 137
- attributes
 - adding to a Shapefile, 2865
 - of a Shapefile, 2855
- autocorrelation, 86
- autocovariance, 86
- autoregressive time-series forecasting
 - coefficients, 2156
 - values, 2160
- AVANTGARDE keyword, 4251
- average
 - mean, 1443
 - median, 1380
- AVERAGE_LINES keyword, 4252
- axes
 - axis gridstyles, 3548
 - axis thickness, 3555
 - changing type, 2613
 - date labels for, 1207
 - direction, 3547
 - end points, 4390
 - gridstyle system variable, 4384
 - input range, 4385
 - linear, 4390
 - location, 3549
 - logarithmic
 - specifying, 3550
 - system variable, 4390
 - margin system variable, 4385
 - multi-level, 4349
 - object, 3540
 - outer margins, 4385
 - output range, 4384
 - range, 4344
 - range (CRANGE, EXACT, EXTEND, RANGE), 3544
 - scaling, 4386
 - style, 4386

axes (*continued*)

- system variables for, 4383
- thickness system variable, 4387
- thickness, (XYZ)THICK keyword, 4344
- title system variable field, 4389
- titles, 3562
 - graphics keyword, 4350
- axis object, 3540
- AXIS procedure, 141
- azimuth, mapping points, 1298
- azimuthal equidistant map projection, 1355

B

background color

- changing, 4291
 - for graphics windows, 610
- BACKGROUND keyword, 4334
- BACKGROUND system variable field, 4379
- background tasks, widgets, 2343
- backing store
 - about draw widgets, 2369
 - defined, 4286
 - for zoom widgets, 488
 - setting, 2504
 - setting for draw widgets, 398
 - setting for WIDGET_DRAW, 2360
 - setting graphics device, 4271

backprojection

- Hough inverse transform, 882
- Radon inverse transform, 1684
- back-substitution, 2074
- backward index list (for histograms), 874

bar charts, 145

BAR_PLOT procedure, 145

base 10 logarithm, 97

base widgets

- bulletin board bases, 2254
- changing title of, 2344
- column bases, 2259
- events returned by, 2277

base widgets (*continued*)

- exclusive, 2260
- exclusive and non-exclusive, 2254
- keyboard focus events, 2262
- mapping and unmapping, 2325
- nonexclusive, 2265
- positioning
 - WIDGET_BASE, 2254
 - WIDGET_CONTROL, 2343
- resize events, 2273
- row bases, 2268
- top-level, 2257
 - WIDGET_BASE, 2254
- batch files, signaling batch process, 4409
- BEGIN...END statement, 149
- benchmarks, 2110
- Bernoulli distribution, 180
- BESELI function, 152
- BESELJ function, 155
- BESELK function, 160
- BESELY function, 163
- Bessel functions
 - BESELI, 152
 - BESELJ, 155
 - BESELK, 160
 - BESELY, 163
 - recurrence relationship, 157
- BETA function, 166
- BETAI, *see* obsolete routines
- big endian byte ordering
 - converting, 201
 - SOCKET procedure, 1968
 - swapping with little endian, 2076
- bi-level images, 2106
- BILINEAR function, 168
- bilinear interpolation, 168
- BIN_DATE function, 171
- binary interpolation, 1007
- BINARY keyword, 4252
- binary SAVE and RESTORE, 1834
- BINARY_TEMPLATE function, 173

- BINDGEN function, [178](#)
- binomial distribution, [180](#)
- BINOMIAL function, [180](#)
- binomial random deviates
 - RANDOMN function, [1694](#)
 - RANDOMU function, [1700](#)
- bins, histogram, [874](#)
- bit shift operation, [1041](#)
- bitmap
 - byte array, [382](#)
 - editing button labels, [2559](#)
 - files
 - reading (READ_BMP), [1714](#)
 - writing (WRITE_BMP), [2507](#)
 - labels, creating, [382](#)
 - widget button labels, [2280](#)
- BITS_PER_PIXEL keyword, [4252](#)
- Bitwise operators, [4397](#)
- BKMAN keyword, [4252](#)
- BLAS_AXPY procedure, [182](#)
- BLK_CON function, [186](#)
- blob coloring, [1211](#)
- block convolution, [186](#)
- BMP files
 - reading (READ_BMP), [1714](#)
 - writing (WRITE_BMP), [2507](#)
- BOLD keyword, [4252](#)
- BOOK keyword, [4252](#)
- Bookman font, [4252](#)
- Boolean operators *see* Bitwise operators
- Boolean operators *see* Logical operators
- bottom margin, setting, [4385](#)
- BOX_CURSOR procedure, [188](#)
- boxcar average, [1960](#)
- BREAK statement, [190](#)
- BREAKPOINT procedure, [191](#)
- breakpoints
 - removing, [192](#)
 - returning information on, [860](#)
 - setting, [193](#)
- BROYDEN function, [194](#)
- Broyden's method, [194](#)
- buffered output
 - emptying, [604](#)
 - flushing, [743](#)
- buffers
 - flushing, [743](#)
 - flushing on exit, [629](#)
 - type-ahead, [786](#)
- bulletin board bases, [2254](#)
- button
 - groups, [402](#)
 - labels, creating, [382](#)
 - widgets
 - button release events, [2287](#)
 - creating bitmap labels, [2280](#)
 - editing bitmap labels, [2559](#)
 - events returned by, [2294](#)
 - groups, [402](#)
 - setting pointer focus, [2323](#)
 - toggle, [2280](#)
 - WIDGET_BUTTON, [2280](#)
- BYPASS_TRANSLATION keyword, [4253](#)
- BYTARR function, [197](#)
- byte
 - arrays
 - BINDGEN function, [178](#)
 - BYTARR function, [197](#)
 - scaling values into a range of bytes, [206](#)
 - swapping, [201](#)
 - swapping short integers, [203](#)
 - type, converting to, [199](#)
- BYTE function, [199](#)
- byte order, reversing
 - SOCKET procedure, [1967](#)
 - SWAP_ENDIAN_INPLACE procedure, [2078](#)
- byte ordering
 - big endian, [175](#)
 - binary data, [175](#)
 - little endian, [175](#)
 - native method, [175](#)

BYTEORDER procedure, 201
 BYTSCL function, 206

C

C_CORRELATE function, 209

C_EDIT, *see* obsolete routines

caching, 1530, 1530

CALDAT procedure, 212

CALENDAR procedure, 216

CALL_EXTERNAL function, 217

CALL_FUNCTION function, 228

CALL_METHOD, 230

CALL_PROCEDURE procedure, 232

CALL_VMS, *see* obsolete routines

calling

external modules from IDL, 217

IDL functions from a string, 228

IDL methods from a string, 230

IDL procedures from a string, 232

routines written in other languages, 217

routines written in other languages at runtime, 1232

sequence, 60

calling sequence

function methods, 2666

procedure methods, 2666

cancel button, 2311

Cartesian

converting from lat/lon, 1322

converting to lat/lon, 1351

CASE statement, 234

CATCH procedure, 236

messages, 1424

catch, C++ language, 236

CD procedure, 239

CEIL function, 242

central map projection, 1355

CGM driver, 4292

changing

access permissions, 664

changing (*continued*)

directories, 239

CHANNEL keyword, 4335

CHANNEL system variable field, 4379

characters

character sets, 4433

newline, 2484

plotting in graphics windows, 2655

size, 2656

CHARSIZE keyword, 4335

CHARSIZE system variable field

annotations, 4379

axis, 4383

CHARTHICK keyword, 4335

CHARTHICK system variable field, 4379

CHEBYSHEV function, 244

CHECK_MATH function, 245

CHI_SQR, *see* obsolete routines

CHI_SQR1, *see* obsolete routines

children, of widgets, 2385

CHISQR_CVF function, 251

CHISQR_PDF function, 253

Chi-square distribution

compute cutoff, 251

compute probability, 253

chi-square error statistic, minimizing, 1229

Chi-square goodness-of-fit test

computing, 2637

contingency table, 368

chmod, 664

CHOLDC procedure, 255

Cholesky decomposition

constructing (CHOLDC), 255

constructing (LA_CHOLDC), 1129

constructing (LA_CHOLMPROVE), 1132

constructing (LA_CHOLSOL), 1136

solution, 257

CHOLSOL function, 257

CINDGEN function, 259

CIR_3PNT procedure, 261

classes

analysis

- IDLanROI, 2680
- IDLanROIGroup, 2708

COM

- IDLcomActiveX, 4215
- IDLcomIDispatch, 4217

file format

- IDLffDICOM, 2728
- IDLffDXF, 2761
- IDLffJPEG2000, 2790
- IDLffLangCat, 2821
- IDLffMrSID, 2835
- IDLffShape, 2849
- IDLffXMLDOMAttr, 2888
- IDLffXMLDOMCDATASection, 2897
- IDLffXMLDOMCharacterData, 2901
- IDLffXMLDOMComment, 2914
- IDLffXMLDOMDocument, 2918
- IDLffXMLDOMDocumentFragment, 2940
- IDLffXMLDOMDocumentType, 2945
- IDLffXMLDOMElement, 2952
- IDLffXMLDOMEntity, 2965
- IDLffXMLDOMEntityReference, 2972
- IDLffXMLDOMNamedNodeMap, 2976
- IDLffXMLDOMNode, 2986
- IDLffXMLDOMNodeList, 3014
- IDLffXMLDOMNotation, 3020
- IDLffXMLDOMProcessingInstruction, 3026
- IDLffXMLDOMText, 3033
- IDLffXMLSAX, 3039

graphics

- IDLgrAxis, 3540
- IDLgrBuffer, 3572
- IDLgrClipboard, 3600
- IDLgrColorbar, 3627
- IDLgrContour, 3647
- IDLgrFont, 3687
- IDLgrImage, 3695

classes (*continued*)

graphics

- IDLgrLegend, 3719
- IDLgrLight, 3740
- IDLgrModel, 3756
- IDLgrMPEG, 3779
- IDLgrPalette, 3795
- IDLgrPattern, 3809
- IDLgrPlot, 3819
- IDLgrPolygon, 3844
- IDLgrPolyline, 3875
- IDLgrPrinter, 3902
- IDLgrROI, 3927
- IDLgrROIGroup, 3946
- IDLgrScene, 3963
- IDLgrSurface, 3976
- IDLgrSymbol, 4010
- IDLgrTessellator, 4021
- IDLgrText, 4032
- IDLgrView, 4056
- IDLgrViewgroup, 4073
- IDLgrVolume, 4085
- IDLgrVRML, 4114
- IDLgrWindow, 4135

iTools

- command collection, 3095
- component (class) base, 3101
- component collection, 3127
- data collection, 3159
- data undo and redo, 3172
- IDLitCommand, 3083
- IDLitCommandSet, 3095
- IDLitComponent, 3101
- IDLitContainer, 3127
- IDLitData, 3141
- IDLitDataContainer, 3159
- IDLitDataOperation, 3172
- IDLitiMessaging, 3187
- IDLitManipulator, 3205
- IDLitManipulatorContainer, 3235
- IDLitManipulatorManager, 3254

classes (*continued*)

- iTools
 - IDLitManipulatorVisual, 3263
 - IDLitOperation, 3271
 - IDLitParameter, 3292
 - IDLitParameterSet, 3315
 - IDLitReader, 3330
 - IDLitTool, 3343
 - IDLitToolbase, 3404
 - IDLitUI, 3406
 - IDLitVisualization, 3425
 - IDLitWindow, 3491
 - IDLitWriter, 3525
 - manipulator collection, 3235
 - manipulating objects, 3205
 - manipulator base, 3254
 - messaging, 3187
 - naming data objects, 3315
 - operating tasks, 3271
 - parameters, 3292
 - reading files, 3330
 - storing data, 3141
 - tool base, 3343, 3404
 - undo and redo commands, 3083
 - user-interface, 3406
 - visual base, 3425
 - window base, 3491
 - writing files, 3525
- Java, IDLjavaObject, 4223, 4223
- miscellaneous
 - IDL_Container, 4182
 - IDL_Savefile, 4193
 - Trackball, 4231
- clearing breakpoints, 192
- CLIP keyword, 4335
- CLIP system variable field, 4380
- clipboard object, 3600
- clipping meshes, 1388
- clipping window, 4380
- clock, system, 2082
- CLOSE keyword, 4253
- CLOSE procedure, 263
- CLOSE_DOCUMENT keyword, 4253
- CLOSE_FILE keyword, 4253
- closing
 - (image processing) function, 570
 - files (CLOSE procedure), 263
 - graphics output files, 4253
 - Shapefiles, 2869
- CLUST_WTS function, 265
- cluster analysis
 - CLUST_WTS function, 265
 - CLUSTER function, 267
- CLUSTER function, 267
- cluster weights, 265
- CLUSTER_TREE procedure, 270
- CMY color system, 481
- CMYK
 - direct graphic support, 4254
 - object graphic support, 3609
- CMYK keyword, DEVICE procedure, 4254
- coastlines, 1302
- colon character, 4408
- COLOR keyword
 - DEVICE procedure, 4254
 - graphics keyword, 4336
- COLOR system variable field, 4380
- color tables
 - colors1.tbl, 1441
 - creating and modifying with XPALETTE, 2607
 - definitions, 1251
 - gamma correction, 768
 - histogram equalization, 837
 - histogram equalizing, 836
 - HLS (Hue, Lightness, Saturation), 880
 - HSV (Hue, Saturation, Value), 892
 - LHB (Lightness, Hue, Brightness), 1620
 - loading, 2171
 - loading into variables (GET keyword), 2172
 - loading predefined, 1251
 - loading predefined interactively, 2578

- color tables (*continued*)
 - maximum indices for draw widgets, 2355
 - modifying predefined colortable files, 1441
 - setting maximum number of indices, 2503
 - stretching, 2024
 - Tektronix 4115, 2098
 - wrapping (MULTI procedure), 1474
- COLOR_CONVERT procedure, 275
- COLOR_EDIT, *see* obsolete routines
- COLOR_QUAN function, 277
- colorbar object, 3627
- COLORMAP_APPLICABLE function, 281
- colors
 - background graphic keyword, 4334
 - background system variable, 4379
 - changing background, 4291
 - converting between color systems, 275
 - default index, 4380
 - erasing background, 610
 - gamma correction (GAMMA_CT), 768
 - indices
 - controlling interpretation, 4257
 - display, 413
 - index selection, 409
 - slider, 481
 - luminance of (CT_LUMINANCE function), 366
 - maximum number available, 2178
 - maximum number for draw widgets, 2355
 - quantization, 277
 - reducing number in an image, 1781
 - resources, for widgets, 2267
 - setting maximum number of indices, 2503
 - shared colormap, 4279
 - systems
 - displaying, 2171
 - modifying with CW_RGBSLIDER, 481
- COLORS keyword, 4254
- column bases, 2259
- COM objects
 - creating in IDL object hierarchies, 4217
- COM objects (*continued*)
 - IDLcomIDispatch object class, 4217
- combobox widgets, 2295
- COMFIT function, 283
- command input buffer, displaying, 865
- command recall, buffer, 1774
- commands
 - displaying previously-executed, 865
 - executive
 - .COMPILE, 67
 - .CONTINUE, 68
 - .EDIT, 69
 - .FULL_RESET_SESSION, 70
 - .GO, 71
 - .OUT, 72
 - .RESET_SESSION, 73
 - .RETURN, 75
 - .RNEW, 76
 - .RUN, 78
 - .SKIP, 80
 - .STEP, 82
 - .STEPOVER, 83
 - .TRACE, 84
- COMMON statement, 286
- comparing array values, 121
- COMPILE_OPT statement, 287
- compiling
 - functions and procedures, 865
 - ITRESOLVE, 1085
 - RESOLVE_ALL, 1799
 - RESOLVE_ROUTINE, 1802
- complex
 - arrays, creating
 - (CINDGEN function), 259
 - (COMPLEXARR function), 295
 - (DCOMPLEXARR function), 500
 - double precision, 495
 - arrays, rounding, 297
 - conjugate, 305
 - data type, 497
 - COMPLEX function, 292

- complex (*continued*)
 - numbers, returning imaginary part of, 963
 - numbers, returning real part of, 735
 - numbers, returning the magnitude of, 88
 - polynomials, 763
- COMPLEX function, 292
- COMPLEXARR function, 295
- COMPLEXROUND function, 297
- compound assignment operators, 4401
- compound widgets
 - CW_ANIMATE, 384
 - CW_ARCBALL, 396
 - CW_BGROUPE, 402
 - CW_CLR_INDEX, 409
 - CW_COLORSEL, 413
 - CW_DEFROI, 416
 - CW_FIELD, 421
 - CW_FILESEL, 427
 - CW_FORM, 433
 - CW_FSLIDER, 442
 - CW_LIGHT_EDITOR, 448
 - CW_LIGHT_EDITOR_GET, 453
 - CW_LIGHT_EDITOR_SET, 456
 - CW_ORIENT, 458
 - CW_PALETTE_EDITOR, 462
 - CW_PALETTE_EDITOR_GET, 470
 - CW_PALETTE_EDITOR_SET, 471
 - CW_PDMENU, 472
 - CW_RGBSLIDER, 481
 - CW_ZOOM, 487
- compression, JPEG
 - read, 1725
 - write, 2514
- COMPUTE_MESH_NORMALS function, 299
- ComputeBounds method, IDLgrVolume, 4105
- ComputeDimensions method
 - IDLgrColorbar, 3641
 - IDLgrLegend, 3734
- Computer Graphics Metafile, 4292
- concatenation, array, 4400
- COND function, 300
- condition number, 300
- conditional expression, 4401
- CONGRID function, 302
- CONJ function, 305
- conjugate, complex, 305
- connectivity list, 1388
- CONSTRAINED_MIN procedure, 307
- container object, 4182
- context-sensitive menu, widget, 2351
- continental boundaries, 1302
- contingency table, 368
- CONTINGENT, *see* obsolete routines
- CONTINUE statement, 314
- contour object, 3647
- contour plots
 - CONTOUR procedure, 315
 - interactive (iTool) routine, 898
 - overlying with images, 958
 - polar, 1562
 - with images and surface plots, 1912
- CONTOUR procedure, 315
- contrast, gamma correction, 768
- convergence criterion, 1482
- CONVERT_COORD function, 328
- converting
 - colors between color systems, 275
 - coordinate systems, 328
- converting expressions
 - between host and network byte ordering, 201
 - to 64-bit integer type, 1266
 - to byte type, 199
 - to complex type
 - COMPLEX function, 292
 - DCOMPLEX function, 497
 - to double-precision type, 590
 - to integer type, 731
 - to longword type, 1264
 - to single-precision floating-point type, 735
 - to string type, 2026
 - to unsigned 64-bit integer type, 2197

- converting expressions (*continued*)
 - to unsigned integer type, 2183
 - to unsigned longword type, 2195
- convex hulls, 1635
- CONVOL function, 331
- convolution
 - computing, 186
 - filtering, 331
- COORD2TO3 function, 336
- coordinates
 - 3D transformations
 - 2D to 3D, 336
 - scaling, 1844
 - scaling 3D, 1842
 - setting, 2064
 - vertices, 2218
 - clipping, 4335
 - converting
 - 2D to 3D, 336
 - between coordinate systems, 379
 - map coordinates, 1353
 - systems, 328
 - defining 3D systems, 359
 - device, 4336
 - normal, 4339
- COPY keyword, DEVICE procedure, 4254
- COPY_LUN procedure, 338
- copying
 - data between files, 338
 - files, 669
 - pixels from one window to another, 4254
- correction, gamma, 768
- CORREL_MATRIX, *see* obsolete routines
- CORRELATE function, 341
- correlation analysis
 - correlation/covariance matrix, 341
 - Kendall's tau rank, 1679
 - lagged autocorrelation, 86
 - lagged crosscorrelation, 209
 - multiple, 1283
 - partial, 1525
- correlation analysis (*continued*)
 - Pearson's correlation, 341
 - Spearman's rho rank, 1679
- correlation coefficient
 - CORRELATE, 341
 - Kendalls's, 1679
 - M_CORRELATE, 1283
 - multiple, 1283
 - P_CORRELATE, 1525
 - partial, 1525
 - Pearson, 341
 - R_CORRELATE, 1679
 - rank, 1679
 - Spearman's, 1679
- COS function, 343
- COSH function, 345
- cosine
 - COS function, 343
 - hyperbolic, 345
 - inverse, 90
- COSINES, *see* obsolete routines
- count accumulation, 878
- country boundaries, 1302
- COURIER keyword, 4255
- CPU procedure, 347
- CRAMER function, 352
- Cramer's rule, 352
- CRANGE system variable field, 4384
- CREATE_CURSOR function, 354
- CREATE_STRUCT function, 356
- CREATE_VIEW procedure, 359
- creating
 - iTools, 933
 - realizing widgets, 2328
 - symbolic links, 688
 - system variables, 521
 - windows, 2503
- cross correlation, 209
- cross covariance, 209
- CROSSP function, 363
- CRVLENGTH function, 364

- CT_LUMINANCE function, [366](#)
- CTI_TEST function, [368](#)
- cubic convolution interpolation
 - returning, [1008](#)
 - warping, [1568](#)
- cubic spline interpolation
 - establishing type, [1988](#)
 - returning, [1990](#)
- current (active) iTool, [1073](#)
- current IDL session, returning information on, [859](#)
- current working directory, CD procedure, [239](#)
- cursor
 - box, [188](#)
 - changing appearance, [4256](#)
 - creating, [354](#)
 - displaying, [2169](#)
 - graphics on Tektronix terminals, [4263](#)
 - hiding, [2170](#)
 - hourglass, [2323](#)
 - manipulator, [3231](#)
 - positioning, [2169](#)
 - reading position of, [1705](#)
 - registering, [1788](#)
 - returning events from draw widgets, [2358](#)
 - setting to crosshair, [4255](#)
 - specifying pattern, [4255](#)
 - type, [4255](#)
- CURSOR procedure
 - and Tektronix terminals, [4263](#)
 - reference, [371](#)
- CURSOR_CROSSHAIR keyword, [4255](#)
- CURSOR_IMAGE keyword, [4255](#)
- CURSOR_STANDARD keyword, [4256](#)
- CURSOR_XY keyword, [4257](#)
- curve fitting
 - COMFIT function, [283](#)
 - CRVLENGTH function, [364](#)
 - CURVEFIT function, [374](#)
 - GAUSS2DFIT function, [773](#)
 - GAUSSFIT function, [777](#)
 - curve fitting (*continued*)
 - LADFIT function, [1214](#)
 - LINFIT function, [1229](#)
 - LMFIT function, [1238](#)
 - MIN_CURVE_SURF function, [1433](#)
 - POLY_FIT function, [1573](#)
 - REGRESS function, [1790](#)
 - SFIT function, [1873](#)
 - SVDFIT function, [2068](#)
- CURVEFIT function, [374](#)
- cutoff value
 - Chi-square distribution, [251](#)
 - F distribution, [650](#)
 - Gaussian distribution, [769](#)
 - T distribution, [2085](#)
- CV_COORD function, [379](#)
- CVTTOBM function, [382](#)
- CW_ANIMATE function, [384](#)
- CW_ANIMATE_GETP procedure, [390](#)
- CW_ANIMATE_LOAD procedure, [392](#)
- CW_ANIMATE_RUN procedure, [394](#)
- CW_ARCBALL function, [396](#)
- CW_BGROUP function, [402](#)
- CW_BSELECTOR, *see* obsolete routines
- CW_CLR_INDEX function, [409](#)
- CW_COLORSEL function, [413](#)
- CW_DEFROI function, [416](#)
- CW_FIELD function, [421](#)
- CW_FILESEL function, [427](#)
- CW_FORM function, [433](#)
- CW_FSLIDER function, [442](#)
- CW_LIGHT_EDITOR function, [448](#)
- CW_LIGHT_EDITOR_GET procedure, [453](#)
- CW_LIGHT_EDITOR_SET procedure, [456](#)
- CW_LOADSTATE, *see* obsolete routines
- CW_ORIENT function, [458](#)
- CW_PALETTE_EDITOR function, [462](#)
- CW_PALETTE_EDITOR_GET procedure, [470](#)
- CW_PALETTE_EDITOR_SET procedure, [471](#)

CW_PDMENU function, reference, [472](#)
 CW_RGBSLIDER function, [481](#)
 CW_SAVESTATE, *see* obsolete routines
 CW_TMPL procedure, [485](#)
 CW_ZOOM function, [487](#)
 cylindrical coordinates, [379](#)
 cylindrical equidistant map projection, [1355](#)

D

data coordinates, converting to other types, [329](#)
 data entry, field widget, [421](#)
 DATA keyword, [4336](#)
 data types, determining using SIZE, [1926](#)
 date
 converting from string to binary, [171](#)
 converting Julian to calendar, [212](#)
 displaying calendars, [216](#)
 labeling axes with, [1207](#)
 returning current, [2082](#)
 Daubechies wavelet filter, [2554](#)
 Davidon-Fletcher-Powell minimization, [541](#)
 day, returning current, [2082](#)
 DBLARR function, [493](#)
 DCINDGEN function, [495](#)
 DCOMPLEX function, [497](#)
 DCOMPLEXARR function, [500](#)
 DDE routines, *see* obsolete routines
 deallocated memory, returning amount of, [863](#)
 debugging
 BREAKPOINT procedure, [191](#)
 PROFILER procedure, [1609](#)
 shared memory, [1889](#)
 decimating a mesh, [1395](#)
 DECOMPOSED keyword, [4257](#)
 decomposition
 Cholesky, [257](#)
 Cholesky (CHOLDC), [255](#)
 Cholesky (LA_CHOLDC), [1129](#)
 Cholesky (LA_CHOLMPROVE), [1132](#)
 Cholesky (LA_CHOLSOL), [1136](#)
 decomposition (*continued*)
 LU
 LA_LUDC procedure, [1178](#)
 LA_LUSOL function, [1184](#)
 LA_TRIDC function, [1191](#)
 LU_COMPLEX function, [1273](#)
 LUDC procedure, [1275](#)
 LUSOL function, [1280](#)
 singular value
 LA_SVD procedure, [1187](#)
 solving, [2075](#)
 SVDC procedure, [2065](#)
 decrement operator, [4392](#)
 default button, [2314](#)
 default font, [4040](#)
 default visual class, [4320](#)
 DEFINE_KEY procedure, [502](#)
 DEFINE_MSGBLK procedure, [511](#)
 DEFINE_MSGBLK_FROM_FILE procedure, [514](#)
 defining
 command or help path, [635](#)
 keys, [502](#)
 region of interest, [519](#)
 system variables, [521](#)
 DEFROI function, [519](#)
 DEFSYSV procedure, [521](#)
 Delaunay triangulation, [2132](#)
 DELETE_SYMBOL, *see* obsolete routines
 deleting
 a region of interest, [2634](#)
 files or directories, [673](#)
 iTools, [1075](#)
 variables, [523](#)
 windows, [2238](#)
 DELLOG, *see* obsolete routines
 DELVAR procedure, [523](#)
 DEMI keyword, [4257](#)
 DEMO_MODE, *see* obsolete routines
 DENDRO_PLOT procedure, [524](#)
 DENDROGRAM procedure, [529](#)

- density function, 874
- DERIV function, 532
- DERIVSIG function, 533
- de-sensitizing widgets, 2331
- destroying
 - widgets, WIDGET_CONTROL, 2316
 - windows, 2238
- DETERM function, 535
- determinant of a square matrix
 - DETERM, 535
 - LA_DETERM, 1139
- deviation, mean absolute, 1378
- DEVICE
 - keyword, 4336
 - procedure, 537
 - supported, 4244
- device
 - backing store, 4286
 - CGM, 4292
 - coordinates, 329
 - display channels, 4379
 - flags, 4376
 - for graphics output, 4244
 - graphics output, 4244
 - height, 4284
 - HP-GL, 4294
 - Microsoft Windows (WIN), 4318
 - monochrome, 4288
 - name of, 4377
 - Null, 4298
 - number of color table indices, 4377
 - number of colors, 4377
 - PCL, 4299
 - PostScript, 4302
 - Printer, 4301
 - Regis terminals, 4315
 - resolution of, 4378
 - size of display, 4378
 - Tektronix, 4316
 - width, 4283
 - x offset, 4283
- device (*continued*)
 - X Windows, 4319
 - y offset, 4283
 - Z-buffer, 4328
- Device fonts, 4414
- DFPMIN procedure, 541
- DIAG_MATRIX function, 545
- diagonal matrix, 545
- DIALOG_MESSAGE function, 547
- DIALOG_PICKFILE function, 550
- DIALOG_PRINTERSETUP function, 557
- DIALOG_PRINTJOB function, 559
- DIALOG_READ_IMAGE function, 561
- DIALOG_WRITE_IMAGE function, 564
- dialogs
 - message dialog box, 547
 - modal, 547
- dicer, 1936
- DICOM
 - conformance summary, 2730
 - IDLffDICOM object, 2728
 - querying DICOM files, 1656
 - reading DICOM files, 1717
- DIFFEQ_23, *see* obsolete routines
- DIFFEQ_45, *see* obsolete routines
- differentiation, CONVOL function, 331
- digital dissolve effect, 577
- digital smoothing polynomial, 1837
- DIGITAL_FILTER function, 567
- DILATE function, 569
- dilation operator, 569
- DINDGEN function, 575
- Direct Graphics, font use, 4415
- direct graphics, CMYK color model, 4254
- DIRECT_COLOR keyword, 4257
- DirectColor visuals, 4257
- direction, of light source, 1867
- directories
 - changing, 239
 - changing permissions, 664
 - creating, 691

- directories (*continued*)
 - deleting, 673
 - expanding pathnames, 679
 - main directory system variable, 4365
 - making, 691
 - popping, 1591
 - printing, 1599
 - pushing, 1629
 - searching for files, 721
 - searching for help files, 4367
- DISP_TEXT, *see* obsolete routines
- displaying images
 - flickering (FLICK), 734
 - TrueColor, 2167
 - TV, 2165
 - with intensity scaling, 2178
- displaying text
 - ASCII files, 2561
 - in a graphics window, 2655
- displays, size, 4378
- DISSOLVE procedure, 577
- DIST function, 579
- distance, between points, 1298
- DISTANCE_MEASURE procedure, 581
- dithering
 - about, 4286
 - Floyd-Steinberg, 4260
 - monochrome, 4288
 - ordered, 4267
 - threshold, 4281
- division operator, 4393
- DLM
 - building sharable libraries, 1292
 - loading, 586
 - registering, 587
- DLM_LOAD procedure, 586
- DLM_REGISTER procedure, 587
- DO_APPLE_SCRIPT, *see* obsolete routines
- DOC_LIBRARY procedure, 588
- documentation headers, extracting, 588
- dollar sign, 4407
- Doppler frequency, 2220
- DOUBLE function, 590
- double-clicks, 2422
- double-precision
 - arrays, creating
 - (DBLARR function), 493
 - (DINDGEN function), 575
 - type, converting to, 590
- drag events
 - for floating-point slider widgets, 443
 - for RGB slider widgets, 482
 - for slider widgets, 2442
 - in draw widgets
 - mouse motion, 2358
 - setting, 2317
- draw widgets, 2353
 - backing store, 2369
 - changing size
 - horizontal, 2317
 - vertical, 2318
 - events
 - determining if set, 2386
 - returned by, 2365
 - returning, 2316
 - motion events, 2358
 - obtaining window number of, 2363
 - returning events
 - button press, 2316
 - motion, 2317
 - viewport
 - draw, 2316
 - motion, 2317
 - viewport, position, 2332
 - viewport, position, widget, 2320
- DRAW_ROI procedure, 592
- drawing
 - arrows, 126
 - continents, 1302
 - lines (PLOTS procedure), 1555
 - objects (ANNOTATE procedure), 103

droplist widgets, [2370](#)
 events returned by, [2378](#)
 returning
 current selection, [2387](#)
 number of elements, [2387](#)
 setting, [2332](#)
 droplist widgets returned events, [2304](#)
 DXF library, supported version, [2761](#)
 DXF object
 displaying, [2564](#)
 IDLffDXF class, [2761](#)
 manipulation, [2564](#)
 dynamic memory
 returning amount in use, [863](#)
 usage, [1383](#)
 dynamically loadable module. *See* DLM
 dynamically loadable modules. *See* DLM
 dynamically loaded modules, keyword, [860](#)

E

earth, interpolating irregularly-sampled data
 over, [2132](#)
 edge detection, CONVOL function, [331](#)
 edge enhancement
 ROBERTS function, [1815](#)
 SOBEL function, [1963](#)
 editable, property sheets, [2426](#)
 EDM, Euclidean Distance Map. *See* Euclidean
 norm
 EFONT procedure, [594](#)
 EIGEN_II, *see* obsolete routines
 EIGENQL function, [596](#)
 eigenvalues
 computing, [1141](#)
 Hessenberg array, [602](#)
 Hessenberg array, returning (HQR), [890](#)
 Hessenberg array, returning (LA_HQR),
 [1163](#)
 non-symmetric array, [599](#)
 symmetric array (EIGENQL), [596](#)
 eigenvalues (*continued*)
 symmetric array (LA_EIGENQL), [1147](#)
 tridiagonal array, [2146](#)
 EIGENVEC function, [599](#)
 eigenvectors
 (EIGENQL), [596](#)
 (LA_EIGENQL), [1147](#)
 non-symmetric array (EIGENVEC), [599](#)
 non-symmetric array (LA_EIGENVEC),
 [1153](#)
 tridiagonal array, [2146](#)
 EJECT keyword, [4258](#)
 elements, number of, [1475](#)
 ELMHES function, [602](#)
 EMPTY procedure, [604](#)
 emptying
 file buffers, [743](#)
 graphics buffers, [604](#)
 ENABLE_SYSRTN procedure, [605](#)
 ENCAPSULATED keyword, [4258](#)
 encapsulated PostScript, [4307](#)
 ENCODING keyword, [4259](#)
 endian
 big, [175](#), [1968](#)
 byte ordering, [175](#)
 little, [175](#), [1968](#)
 end-of-file, [607](#)
 entities
 inserting into a Shapefile, [2882](#)
 retrieving from a Shapefile, [2874](#)
 types of in ShapeFile, [2851](#)
 environment variables, [792](#)
 adding or changing, [1869](#)
 returning value of, [792](#)
 EOF function, [607](#)
 EPS machine-specific parameter, [1287](#)
 EPSI files, [4270](#)
 EPSNEG machine-specific parameter, [1287](#)
 EQ operator, defined, [4399](#)
 EQUAL_VARIANCE, *see* obsolete routines

- Erase method
 - IDLgrBuffer, [3581](#)
 - IDLgrWindow, [4152](#)
- ERASE procedure, [610](#)
- erasing IDL windows, [610](#)
- ERF function, [612](#)
- ERFC function, [614](#)
- ERFCX function, [616](#)
- ERODE function, [618](#)
- erosion operator, morphologic, [618](#)
- error messages
 - generating (MESSAGE procedure), [1424](#)
 - modal dialog box, [547](#)
 - returning text of (STRMESSAGE function), [2037](#)
- ERRORF, *see* obsolete routines
- errors
 - error bars, [624](#)
 - error bars (OPLOTERR), [1523](#)
 - error bars (PLOTERR), [1553](#)
 - handling
 - CATCH procedure, [236](#)
 - ON_ERROR procedure, [1497](#)
 - ON_IOERROR procedure, [1498](#)
 - OPEN procedure, [1513](#)
 - messages, generating (MESSAGE procedure), [1424](#)
 - messages, modal dialog box, [547](#)
 - messages, returning text of (STRMESSAGE function), [2037](#)
 - placing error status in variable, [1513](#)
- ERRPLOT procedure, [624](#)
- Euclidean Distance Map. *See* Euclidean norm
- Euclidean norm
 - distance map, [1449](#)
 - of vector, [1484](#)
- events
 - basic structure returned by all widgets, [2380](#)
 - button release, [2287](#)
 - clearing, [2312](#)
 - processing, [2379](#)
- events (*continued*)
 - returned by
 - button widgets, [2294](#)
 - draw widgets, [2365](#)
 - droplist widgets, [2378](#)
 - list widgets, [2422](#)
 - slider widgets, [2449](#)
 - text widgets, [2489](#)
 - top-level base widgets, [2277](#)
 - returning
 - base resize events, [2273](#)
 - handler procedure name, [2388](#)
 - keyboard focus events
 - WIDGET_BASE, [2262](#)
 - WIDGET_DRAW, [2358](#)
 - WIDGET_TABLE, [2467](#)
 - WIDGET_TEXT, [2483](#)
 - sending to widgets, [2330](#)
 - top-level base kill events, [2272](#)
- events returned by
 - droplist widgets, [2304](#)
 - tab widgets, [2459](#)
 - tree widgets, [2501](#)
- example files, surf_track.pro, [4238](#)
- exclamation point
 - embedded formatting commands, [4433](#)
 - system variables, [4406](#)
- EXECUTE function
 - CALL_FUNCTION speed, [228](#)
 - reference, [626](#)
- EXIT procedure, [629](#)
- exiting IDL, EXIT procedure, [629](#)
- EXP function, [631](#)
- EXPAND procedure, [633](#)
- EXPAND_PATH function, [635](#)
- expanding pathnames, [679](#)
- EXPINT function, [640](#)
- exponential
 - integral, [640](#)
 - natural, [631](#)

exponential (*continued*)

random deviates

RANDOMN function, [1695](#)

RANDOMU function, [1700](#)

exponentiation operator, [4393](#)

expressions

determining data type, SIZE function, [1926](#)

regular, [2020](#)

returning information on, [859](#)

external, sharable object, [217](#)

EXTRAC function, [643](#)

EXTRACT_SLICE function, [646](#)

F

F distribution

cutoff value, [650](#)

probability, [652](#)

F_CVF function, [650](#)

F_PDF function, [652](#)

F_TEST, *see* obsolete routines

F_TEST1, *see* obsolete routines

FACTORIAL function, [654](#)

Fast Fourier transform, [656](#)

FFT function, [656](#)

field

plots, [739](#)

widget, [421](#)

file

symbolic links

creating, [688](#)

following, [695](#)

file pointer, moving, [1933](#)

file units

allocating, [788](#)

returning information about, [861](#)

See also logical unit numbers

setting file position pointer, [1560](#)

FILE_BASENAME function, [661](#)

FILE_CHMOD procedure, [664](#)

FILE_COPY procedure, [669](#)

FILE_DELETE procedure, [673](#)

FILE_DIRNAME function, [676](#)

FILE_EXPAND_PATH function, [679](#)

FILE_INFO function, reference, [681](#)

FILE_LINES function, [685](#)

FILE_LINK procedure, [688](#)

FILE_MKDIR procedure, [691](#)

FILE_MOVE procedure, [692](#)

FILE_READLINK function, [695](#)

FILE_SAME function, [697](#)

FILE_SEARCH function, [700](#)

FILE_TEST function, [717](#)

FILE_WHICH function, [721](#)

FILENAME keyword, [4260](#)

FILEPATH function, [723](#)

files

changing permissions, [664](#)

closing

CLOSE procedure, [263](#)

DEVICE keyword, [4253](#)

freeing file units, [748](#)

comparing, [697](#)

copying, [669](#)

current pointer position, [751](#)

deleting, [673](#)

deriving

base name, [661](#)

directory name, [676](#)

displaying ASCII, [2561](#)

expanding pathnames, [679](#)

filenames, [4260](#)

finding, [550](#), [1530](#), [1530](#)

finding in IDL distribution, [723](#)

freeing logical unit numbers, [748](#)

iTools

reading objects, [3330](#)

writing objects, [3525](#)

moving, [692](#)

opening, OPEN procedure, [1511](#)

pointer position, POINT_LUN procedure,

[1560](#)

- files (*continued*)
 - printing to, 1596
 - protection classes, 664
 - reading
 - ASCII data, 1709
 - binary data from, 1768
 - data, 1706
 - unformatted binary data, 1768
 - returning information on open, 859
 - searching directories, 721
 - selecting, 550
 - size of, 751
 - special functions (IOCTL function), 1018
 - writing
 - formatted output, 1596
 - unformatted binary data, 2548
- FILL_DIST system variable field, 4375
- FILLCONTOUR, *see* obsolete routines
- filling
 - plotting symbols, 2205
 - polygons
 - POLYFILL procedure, 1577
 - POLYFILLV, 1582
- filtering
 - convolution, 186
 - digital, 567
 - digital filters, 567
 - filenames, 552
 - frequency domain, 656
 - Hanning windows, 843
 - histogram equalization, 871
 - Lee filter algorithm, 1219
 - mean, 1960
 - median, 1380
 - morphologic dilation, 569
 - morphologic erosion, 618
 - Roberts, 1815
 - Sobel, 1963
- FINDFILE, *see* obsolete routines
- FINDGEN function, 725
- finding, files, 550
- finite, numbers, 727
- FINITE function, reference, 727
- FIX function, 731
- FLAGS system variable field, 4376
- FLICK procedure, 734
- FLOAT function, 735
- floating-point
 - arithmetic, 1286
 - arrays
 - FINDGEN, 725
 - FLTARR, 741
 - converting type to, 735
 - mantissa, 1286
 - native format, 201
 - precision, 1287
 - slider widgets, 442
 - XDR format, 201
- FLOOR function, 737
- flow
 - control, 4283
 - field, plotting 3D, 739
 - field, plotting velocity, 2213
- FLOW3 procedure, 739
- FLOYD keyword, 4260
- FLTARR function, 741
- FLUSH procedure, 743
- focus events, keyboard
 - WIDGET_CONTROL, 2324
 - WIDGET_INFO, 2390
- focus events, keyboard WIDGET_BASE, 2262
- focus events, keyboard WIDGET_TABLE, 2467
- focus events, keyboard WIDGET_TEXT, 2483
- FONT keyword, 4337
- font object
 - IDLgrFont, 3687
 - modifiers, 3692
- FONT system variable field, 4380
- FONT_INDEX keyword, 4260
- FONT_SIZE keyword, 4261

fonts

- character sets, [4433](#)
- default for widgets, [2314](#)
- device, [4414](#)
- Direct Graphics, [4415](#)
- displaying vector fonts, [1914](#)
- displaying X Windows fonts, [2568](#)
- editing, [594](#)
- examples of TrueType fonts, [4442](#)
- examples of vector fonts, [4445](#)
- finding current X windows font, [4261](#)
- finding names of, [4261](#)
- finding number of, [4261](#)
- hardware, [4414](#)
- Hershey, [4414](#)
- Object Graphics, [4415](#)
- outline, [4414](#)
- positioning commands, [4435](#)
- PostScript, [1618](#)
- TrueType
 - overview, [4414](#)
 - specifying with DEVICE, [4277](#)
 - vector, [4414](#)

FOR statement, [744](#)

foreground color, [2611](#)

formal parameters, [61](#)

FORMAT_AXIS_VALUES function, [745](#)

forms, creating, [433](#)

FORRD, *see* obsolete routines

FORRD_KEY, *see* obsolete routines

Fortran file formats, [1513](#)

forward difference, [2158](#)

FORWARD_FUNCTION statement, [747](#)

FORWRT, *see* obsolete routines

four-dimensional displays, [1584](#)

Fourier transform, [656](#)

FREE_LUN procedure, [748](#)

freeing, heap variables

- HEAP_FREE procedure, [854](#)
- PTR_FREE procedure, [1622](#)

freeing, objects, [1489](#)

FRIEDMAN, *see* obsolete routines

FSTAT function, reference, [750](#)

FSTAT structure, [750](#)

FULSTR function, [753](#)

FUNCT procedure, [755](#)

function keys

defining

example, [509](#)

reference, [502](#)

for different keyboards, [1870](#)

returning definitions, [859](#)

function methods, calling sequence for, [2666](#)

FUNCTION statement, [757](#)

functions

calling sequence for, [60](#)

compiled, [1825](#)

displaying compiled, [865](#)

FV_TEST function, [758](#)

FX_ROOT function, [760](#)

FZ_ROOTS function, [763](#)

G

gamma correction, [768](#)

GAMMA function, [766](#)

gamma function

incomplete, [939](#)

logarithm of, [1246](#)

gamma random deviates

RANDOMN function, [1695](#)

RANDOMU function, [1700](#)

GAMMA_CT procedure, [768](#)

garbage collection, [857](#)

GAUSS, *see* obsolete routines

GAUSS_CVF function, [769](#)

GAUSS_PDF function, [771](#)

GAUSS2DFIT function, [773](#)

GAUSSFIT function, [777](#)

Gaussian

distribution

cutoff value, [769](#)

- Gaussian (*continued*)
 - distribution
 - probability, [771](#)
 - elimination method, [1016](#)
 - integral, [782](#)
 - iterated quadrature, [993](#), [997](#)
 - two-dimensional fit, [773](#)
- GAUSSINT function, [782](#)
- Gauss-Krueger map projection, [1357](#)
- Gauss-Markov linear model, [1160](#)
- Gauss-Seidel iteration, [833](#)
- GE operators, [4399](#)
- general perspective map projection, [1356](#)
- GET_CURRENT_FONT keyword, [4261](#)
- GET_DECOMPOSED keyword, [4261](#)
- GET_DRIVE_LIST function, [784](#)
- GET_FONTNAMES keyword, [4261](#)
- GET_FONTNUM keyword, [4261](#)
- GET_GRAPHICS_FUNCTION keyword, [4262](#)
- GET_KBRD function, [786](#)
- GET_LUN procedure, [748](#), [788](#)
- GET_PAGE_SIZE keyword, [4262](#)
- GET_SCREEN_SIZE function, [790](#)
- GET_SCREEN_SIZE keyword, [4262](#)
- GET_SYMBOL, *see* obsolete routines
- GET_VISUAL_DEPTH keyword, [4262](#)
- GET_VISUAL_NAME keyword, [4262](#)
- GET_WINDOW_POSITION keyword, [4263](#)
- GET_WRITE_MASK keyword, [4263](#)
- GetByName method
 - IDLgrModel, [3767](#)
 - IDLgrScene, [3970](#)
 - IDLgrView, [4067](#)
 - IDLgrViewgroup, [4079](#)
- GetDeviceInfo method
 - IDLgrBuffer, [3583](#)
 - IDLgrClipboard, [3617](#)
 - IDLgrVRML, [4125](#)
 - IDLgrWindow, [4154](#)
- GetDimensions method, IDLgrWindow, [4156](#)
- GETENV function, [792](#)
- GetFontnames method
 - IDLgrBuffer, [3585](#)
 - IDLgrPrinter, [3917](#)
 - IDLgrVRML, [4127](#)
 - IDLgrWindow, [4158](#)
- GETHELP, *see* obsolete routines
- GetRGB method, IDLgrPalette, [3801](#)
- GetTextDimensions method
 - IDLgrBuffer, [3588](#)
 - IDLgrClipboard, [3622](#)
 - IDLgrPrinter, [3920](#)
 - IDLgrVRML, [4130](#)
 - IDLgrWindow, [4161](#)
- GIF files
 - reading, [1719](#)
 - writing, [2510](#)
- GIN_CHARS keyword, [4263](#)
- gnomic map projection, [1355](#)
- gnomonic map projection, [1355](#)
- GOODFIT, *see* obsolete routines
- GOTO statement, reference, [795](#)
- Gouraud shading, [1867](#)
- graphics
 - cursor positioning, [371](#)
 - devices
 - DEVICE procedure, [537](#)
 - erasing, [610](#)
 - list of supported, [4244](#)
 - returning information about current, [860](#)
 - setting, [1865](#)
 - functions
 - getting, [4262](#)
 - setting, [4277](#)
 - image file formats
 - BMP
 - reading, [1714](#)
 - writing, [2507](#)
 - GIF, [1719](#), [2510](#)
 - Interfile, [1723](#)
 - JPEG

- graphics (*continued*)
 - reading, 1725
 - writing, 2514
 - NRIF, 2519
 - PICT
 - reading, 1736
 - writing, 2521
 - SRF
 - reading, 1744
 - writing, 2530
 - TIFF
 - reading, 1750
 - writing, 2534
 - X11 bitmap, 1762
 - XWD, 1764
 - keywords (collected), 4333
 - GRAPHICS_TIMES procedure, 2110
 - great circle, 1298
 - GRID_INPUT procedure, 796
 - GRID_TPS function, 801
 - GRID3 function, 805
 - GRIDDATA function, 808
 - gridding
 - irregular intervals, 2144
 - irregularly gridded, 2132
 - spherical
 - SPH_SCAT function, 1982
 - TRIGRID function, 2136
 - GRIDSTYLE system variable field, 4384
 - growth trends, 283
 - GS_ITER function, 833
 - GT operator, 4399
 - guard digits, 1286
- ## H
- H_EQ_CT procedure, 836
 - H_EQ_INT procedure, 837
 - H5_BROWSER function, 840
 - halftoning, 4286
 - halting program execution, 2012
 - Hammer-Aitoff map projection, 1355
 - HANDLE_CREATE, *see* obsolete routines
 - HANDLE_FREE, *see* obsolete routines
 - HANDLE_INFO, *see* obsolete routines
 - HANDLE_MOVE, *see* obsolete routines
 - HANDLE_VALUE, *see* obsolete routines
 - HANNING function, 843
 - hardware fonts, 4414
 - HDF_BROWSER function, 846
 - HDF_READ function, 850
 - HDF_VD_GETNEXT, *see* obsolete routines
 - HDF5 files, viewing, 840
 - heap variables
 - creating, 1623
 - destroying, 1622
 - freeing, HEAP_FREE, 854
 - garbage collection, 857
 - HEAP_FREE procedure, 854
 - HEAP_GC procedure, 857
 - help, ONLINE_HELP procedure, 1500
 - HELP procedure, 859
 - HELP_VM, *see* obsolete routines
 - HELVETICA keyword, 4263
 - Hershey fonts, 4414
 - Hershey, Dr. A. V., 4416
 - Hessenberg array
 - eigenvalues (HQR), 890
 - eigenvalues (LA_HQR), 1163
 - returning (ELMHES), 602
 - returning (LA_ELMHES), 1157
 - Hessenberg array or matrix (LA_ELMHES), 1157
 - Hewlett-Packard Graphics Language, *see* HP-GL
 - hidden object classes, 1488, 2677
 - hiding cursor, 2170
 - HILBERT function, 867
 - HIST_2D function, 869
 - HIST_EQUAL function, 871
 - histogram
 - equalization
 - H_EQ_CT function, 836

histogram (*continued*)
 equalization
 interactive (H_EQ_INT function), 837
 plotting mode, 4341
 view of ROI, 2629
 HISTOGRAM function, 874
 HLS color system
 adjusting with slider, 481
 converting, 275
 displaying, 2171
 HLS procedure, 880
 Hough
 backprojection, 882
 transform, 882
 HOUGH function, 882
 hourglass cursor
 for widgets, 2323
 saving, 2382
 Householder
 method, 2149
 reductions, 596
 HP-GL
 driver, 4294
 files, 4289
 HQR function, 890
 HSV color system
 adjusting with slider, 481
 converting, 275
 displaying, 2171
 HSV procedure, 892
 HSV_TO_R, see obsolete routines
 HTML, 1438
 hyperbolic
 cosine, 345
 sine, 1924
 tangent, 2096
 HyperText Markup Language, 1438
 hypothesis testing
 Chi-square model validation, 2637
 contingency test for independence, 368
 F-variances test, 758

hypothesis testing (*continued*)
 Kruskal-Wallis H-test, 1124
 Lomb frequency test, 1248
 Mann-Whitney U-test, 1828
 median delta test, 1374
 normality test
 FV_TEST, 758
 TM_TEST, 2116
 runs test for randomness, 1682
 sign test, 1831
 t-means test, 2116
 Wilcoxon rank-sum test, 1828

/

I/O, *see* input/output
 IBETA function, 894
 IBETA machine-specific parameter, 1286
 Iconify method, IDLgrWindow, 4163
 iconifying
 widgets, 2323
 windows, 2552
 icons, editing, 2559
 ICONTOUR procedure, 898
 IDENTITY function, 923
 IDL, for Windows, 4318
 IDL_Container
 class, 4182
 methods, 4182
 Add, 4184
 Cleanup, 4185
 Count, 4186
 Get, 4187
 Init, 4189
 IsContained, 4190
 Move, 4191
 Remove, 4192
 IDL_Savefile
 class, 4193
 methods, 4193
 Cleanup, 4197

IDL_Savefile (*continued*)

- methods, 4193
 - Contents, 4198
 - Init, 4201
 - Names, 4203
 - Restore, 4206
 - Size, 4213

IDL_VALIDATE_NAME function, 926

IDLAnROI

- class, 2680
- methods, 2681
 - AppendData, 2686
 - Cleanup, 2688
 - ComputeGeometry, 2689
 - ComputeMask, 2691
 - ContainsPoints, 2694
 - GetProperty, 2696
 - Init, 2697
 - RemoveData, 2699
 - ReplaceData, 2701
 - Rotate, 2704
 - Scale, 2705
 - SetProperty, 2706
 - Translate, 2707
- properties, 2682

IDLAnROIGroup

- class, 2708
- methods, 2708
 - Add, 2712
 - Cleanup, 2713
 - ComputeMask, 2714
 - ComputeMesh, 2717
 - ContainsPoints, 2719
 - GetProperty, 2721
 - Init, 2722
 - Rotate, 2723
 - Scale, 2724
 - Translate, 2725
- properties, 2710

IDLcomActiveX

- class, 4215

IDLcomActiveX (*continued*)

- methods, 4215
- IDLcomIDispatch
- class, 4217
 - methods, 4218
 - GetProperty, 4220
 - Init, 4221
 - SetProperty, 4222
 - properties, 4219
- IDLffDICOM
- class, 2728
 - methods, 2729
 - Cleanup, 2735
 - DumpElements, 2736
 - GetChildren, 2737
 - GetDescription, 2739
 - GetElement, 2741
 - GetGroup, 2743
 - GetLength, 2745
 - GetParent, 2747
 - GetPreamble, 2749
 - GetReference, 2750
 - GetValue, 2752
 - GetVR, 2755
 - Init, 2757
 - Read, 2759
 - Reset, 2760
 - properties, 2734
- IDLffDXF, 2786
- class, 2761
 - methods, 2762
 - Cleanup, 2764
 - GetContents, 2765
 - GetEntity, 2768
 - GetPalette, 2781
 - Init, 2782, 2782
 - PutEntity, 2783
 - Read, 2784
 - RemoveEntity, 2785
 - Reset, 2786
 - SetPalette, 2787

IDLffDXF (*continued*)

methods, 2762
 Write, 2788

IDLffJPEG2000

class, 2790
 methods, 2790
 Cleanup, 2806
 GetData, 2807
 GetProperty, 2811
 GetTileProperty, 2812
 Init, 2814
 SetData, 2816
 SetProperty, 2820
 properties, 2795

IDLffLangCat

class, 2821
 methods, 2822
 AppendCatalog, 2827
 Cleanup, 2829
 GetProperty, 2830
 Init, 2831
 Query, 2833
 SetProperty, 2834
 properties, 2823

IDLffLanguageCat, *see* obsolete routines**IDLffMrSID**

class, 2835
 methods, 2835
 Cleanup, 2838
 GetDimsAtLevel, 2839
 GetImageData, 2841
 GetProperty, 2844
 Init, 2847
 properties, 2837

IDLffShape

class, 2849
 methods, 2849
 AddAttribute, 2865
 Cleanup, 2868
 Close, 2869
 DestroyEntity, 2870

IDLffShape (*continued*)

methods, 2849
 GetAttributes, 2872
 GetEntity, 2874
 GetProperty, 2876
 Init, 2878
 Open, 2880
 PutEntity, 2882
 SetAttributes, 2884
 properties, 2861

IDLffXMLDOMAttr

class, 2888
 methods, 2888
 Cleanup, 2891
 GetName, 2892
 GetSpecified, 2893
 GetValue, 2894
 Init, 2895
 SetValue, 2896
 properties, 2890

IDLffXMLDOMCDATASection

class, 2897
 methods, 2897
 Cleanup, 2899
 Init, 2900
 properties, 2898

IDLffXMLDOMCharacterData

class, 2901
 methods, 2901
 AppendData, 2904
 Cleanup, 2905
 DeleteData, 2906
 GetData, 2907
 GetLength, 2908
 Init, 2909
 InsertData, 2910
 ReplaceData, 2911
 SetData, 2912
 SubstringData, 2913
 properties, 2903

- IDLffXMLDOMComment
 - class, [2914](#)
 - methods, [2914](#)
 - Cleanup, [2916](#)
 - Init, [2917](#), [2917](#)
 - properties, [2915](#)
- IDLffXMLDOMDocument
 - class, [2918](#)
 - methods, [2918](#)
 - Cleanup, [2921](#)
 - CreateAttribute, [2922](#)
 - CreateCDATASection, [2923](#)
 - CreateComment, [2924](#)
 - CreateElement, [2926](#)
 - CreateEntityReference, [2927](#)
 - CreateProcessingInstruction, [2928](#)
 - CreateTextNode, [2929](#)
 - DocumentFragment, [2925](#)
 - GetDoctype, [2930](#)
 - GetDocumentElement, [2931](#)
 - GetElementsByTagName, [2932](#)
 - Init, [2933](#)
 - Load, [2936](#)
 - Save, [2939](#)
 - properties, [2920](#)
- IDLffXMLDOMDocumentFragment
 - class, [2940](#)
 - methods, [2941](#)
 - Cleanup, [2943](#)
 - Init, [2944](#)
 - properties, [2942](#)
- IDLffXMLDOMDocumentType
 - class, [2945](#)
 - methods, [2945](#)
 - Cleanup, [2947](#)
 - GetEntities, [2948](#)
 - GetName, [2949](#)
 - GetNotations, [2950](#)
 - Init, [2951](#)
 - properties, [2946](#)
- IDLffXMLDOMDOMELEMENT, methods, [2952](#)
- IDLffXMLDOMDOMEntity, methods, [2965](#)
- IDLffXMLDOMELEMENT
 - class, [2952](#)
 - methods
 - Cleanup, [2955](#)
 - GetAttribute, [2956](#)
 - GetAttributeNode, [2957](#)
 - GetElementsByTagName, [2958](#)
 - GetTagName, [2959](#)
 - Init, [2960](#)
 - RemoveAttribute, [2961](#)
 - RemoveAttributeNode, [2962](#)
 - SetAttribute, [2963](#)
 - SetAttributeNode, [2964](#)
 - properties, [2954](#)
- IDLffXMLDOMEntity
 - class, [2965](#)
 - methods
 - Cleanup, [2967](#)
 - GetNotationName, [2968](#)
 - GetPublicId, [2969](#)
 - GetSystemId, [2970](#)
 - Init, [2971](#)
 - properties, [2966](#)
- IDLffXMLDOMEntityReference
 - class, [2972](#)
 - methods, [2972](#)
 - Cleanup, [2974](#)
 - Init, [2975](#)
 - properties, [2973](#)
- IDLffXMLDOMNamedNodeMap
 - class, [2976](#)
 - methods, [2976](#)
 - Cleanup, [2979](#)
 - GetLength, [2980](#)
 - GetNamedItem, [2981](#)
 - Init, [2982](#)
 - Item, [2983](#)
 - RemoveNamedItem, [2984](#)
 - SetNamedItem, [2985](#)

- IDLffXMLDOMNamedNodeMap (*continued*)
 - properties, 2978
- IDLffXMLDOMNode
 - class, 2986
 - method, SetNodeValue, 3013
 - methods, 2986
 - AppendChild, 2989
 - Cleanup, 2990
 - CloneNode, 2991
 - GetAttributes, 2992
 - GetChildNodes, 2993
 - GetFirstChild, 2994
 - GetLastChild, 2995
 - GetNextSibling, 2996
 - GetNodeName, 2997
 - GetNodeType, 2999
 - GetNodeValue, 3001
 - GetOwnerDocument, 3003
 - GetParentNode, 3004
 - GetPreviousSibling, 3005
 - HasChildNodes, 3006
 - Init, 3007
 - InsertBefore, 3008
 - RemoveChild, 3010
 - ReplaceChild, 3011
 - properties, 2988
- IDLffXMLDOMNodeList
 - class, 3014
 - methods, 3014
 - Cleanup, 3016
 - GetLength, 3017
 - Init, 3018
 - Item, 3019
 - properties, 3015
- IDLffXMLDOMNotation
 - class, 3020
 - methods, 3020
 - Cleanup, 3022
 - GetPublicID, 3023
 - GetSystemID, 3024
 - Init, 3025
- IDLffXMLDOMNotation (*continued*)
 - properties, 3021
- IDLffXMLDOMProcessingInstruction
 - class, 3026
 - methods, 3026
 - Cleanup, 3028
 - GetData, 3029
 - GetTarget, 3030
 - Init, 3031
 - SetData, 3032
 - properties, 3027
- IDLffXMLDOMText
 - class, 3033
 - methods, 3033
 - Cleanup, 3035
 - Init, 3036
 - IsIgnorableWhitespace, 3037
 - SplitText, 3038
 - properties, 3034
- IDLffXMLSAX
 - class, 3039
 - methods, 3040
 - AttributeDecl, 3046
 - Characters, 3048
 - Cleanup, 3049
 - Comment, 3050
 - ElementDecl, 3051
 - EndCDATA, 3052
 - EndDocument, 3053
 - EndDTD, 3054
 - EndElement, 3055
 - EndEntity, 3056
 - EndPrefixMapping, 3057
 - Error, 3058
 - ExternalEntityDecl, 3060
 - FatalError, 3061
 - GetProperty, 3062
 - IgnorableWhitespace, 3063
 - Init, 3064
 - InternalEntityDecl, 3065
 - NotationDecl, 3066

IDLffXMLSAX (*continued*)

- methods, 3040
 - ParseFile, 3067
 - ProcessingInstruction, 3068
 - SetProperty, 3069
 - SkippedEntity, 3070
 - StartCDATA, 3071
 - StartDocument, 3072
 - StartDTD, 3073
 - StartElement, 3074
 - StartEntity, 3076
 - StartPrefixmapping, 3077
 - StopParsing, 3078
 - UnparsedEntityDecl, 3079
 - Warning, 3080
- properties, 3042

IDLgrAxis

- class, 3540
- methods, 3541, 3757
 - Cleanup, 3565
 - GetCTM, 3566
 - GetProperty, 3568
 - Init, 3569
 - SetProperty, 3571
- properties, 3542

IDLgrBuffer

- class, 3572
- methods, 3573
 - Cleanup, 3579
 - Draw, 3580
 - Erase, 3581
 - GetContiguousPixels, 3582
 - GetDeviceInfo, 3583
 - GetFontnames, 3585
 - GetProperty, 3587
 - GetTextDimensions, 3588
 - Init, 3590
 - PickData, 3592
 - Read, 3595
 - Select, 3596
 - SetProperty, 3599

IDLgrBuffer (*continued*)

- properties, 3574

IDLgrClipboard

- class, 3600
- methods, 3601
 - Cleanup, 3607
 - Draw, 3608
 - GetContiguousPixels, 3616
 - GetDeviceInfo, 3617
 - GetProperty, 3621
 - GetTextDimensions, 3622
 - Init, 3624
- properties, 3602

IDLgrColorbar

- class, 3627
- methods, 3628
 - Cleanup, 3640
 - ComputeDimensions, 3641
 - GetProperty, 3643
 - Init, 3644
 - SetProperty, 3646
- properties, 3630

IDLgrContour

- class, 3647
- methods, 3648
 - AdjustLabelOffsets, 3677
 - Cleanup, 3678
 - GetCTM, 3679
 - GetLabelInfo, 3681
 - GetProperty, 3683
 - Init, 3684
 - SetProperty, 3686
- properties, 3650

IDLgrFont

- class, 3687
- methods, 3687
 - Cleanup, 3690
 - GetProperty, 3691
 - Init, 3692
 - SetProperty, 3694
- properties, 3688

- IDLgrImage
 - class, 3695
 - methods, 3696
 - Cleanup, 3712
 - GetCTM, 3713
 - GetProperty, 3715
 - Init, 3716
 - SetProperty, 3718
 - properties, 3698
- IDLgrLegend
 - class, 3719, 3719
 - methods, 3720
 - Cleanup, 3733
 - ComputeDimensions, 3734
 - GetProperty, 3736
 - Init, 3737
 - SetProperty, 3739
 - properties, 3722
- IDLgrLight
 - class, 3740
 - methods, 3741
 - Cleanup, 3749
 - GetCTM, 3750
 - GetProperty, 3752
 - Init, 3753
 - SetProperty, 3755
 - properties, 3742
- IDLgrModel
 - class, 3756
 - methods, 3757
 - Add, 3764
 - Cleanup, 3765
 - Draw, 3766
 - GetByName, 3767
 - GetCTM, 3769
 - GetProperty, 3771
 - Init, 3772
 - Reset, 3774
 - Rotate, 3775
 - Scale, 3776
 - SetProperty, 3777
- IDLgrModel (*continued*)
 - methods, 3757
 - Translate, 3778
 - properties, 3758
 - Translate, 3778
- IDLgrMPEG
 - class, 3779
 - methods, 3780
 - Cleanup, 3788
 - GetProperty, 3789
 - Init, 3790
 - Put, 3792
 - Save, 3793
 - SetProperty, 3794
 - properties, 3781
- IDLgrPalette
 - class, 3795
 - methods, 3795
 - Cleanup, 3800
 - GetProperty, 3802
 - GetRGB, 3801
 - Init, 3803
 - LoadCT, 3805
 - NearestColor, 3806
 - SetProperty, 3808
 - SetRGB, 3807
 - properties, 3797
- IDLgrPattern
 - class, 3809
 - methods, 3809
 - Cleanup, 3814
 - GetProperty, 3815
 - Init, 3816
 - SetProperty, 3818
 - properties, 3811
- IDLgrPlot
 - class, 3819
 - methods, 3820
 - Cleanup, 3837
 - GetCTM, 3838
 - GetProperty, 3840

IDLgrPlot (*continued*)

methods, 3820
 Init, 3841
 SetProperty, 3843
 properties, 3821

IDLgrPolygon

class, 3844
 methods, 3845
 Cleanup, 3868
 GetCTM, 3869
 GetProperty, 3871
 Init, 3872
 SetProperty, 3874
 properties, 3846

IDLgrPolyline

class, 3875
 methods, 3876
 Cleanup, 3895
 GetCTM, 3896
 GetProperty, 3898
 Init, 3899
 SetProperty, 3901
 properties, 3877

IDLgrPrinter

class, 3902
 methods, 3903
 Cleanup, 3910
 Draw, 3911
 GetContiguousPixels, 3916
 GetFontnames, 3917
 GetProperty, 3919
 GetTextDimensions, 3920
 Init, 3922
 NewDocument, 3924
 NewPage, 3925
 SetProperty, 3926
 properties, 3904

IDLgrROI

class, 3927
 methods, 3928
 Cleanup, 3939

IDLgrROI (*continued*)

methods, 3928
 GetProperty, 3940
 Init, 3941
 PickVertex, 3943
 SetProperty, 3945
 properties, 3929

IDLgrROIGroup

class, 3946
 methods, 3947
 Add, 3955
 Cleanup, 3956
 GetProperty, 3957
 Init, 3958
 PickRegion, 3960
 SetProperty, 3962
 properties, 3948

IDLgrScene

class, 3963
 methods, 3963
 Add, 3968
 Cleanup, 3969
 GetByName, 3970
 GetProperty, 3972
 Init, 3973
 SetProperty, 3975
 properties, 3965

IDLgrSurface

class, 3976
 methods, 3977
 Cleanup, 4003
 GetCTM, 4004
 GetProperty, 4006
 Init, 4007
 SetProperty, 4009
 properties, 3978

IDLgrSymbol

class, 4010
 methods, 4010
 Cleanup, 4016
 GetProperty, 4017

IDLgrSymbol (*continued*)

methods, 4010
 Init, 4018
 SetProperty, 4020
 properties, 4012

IDLgrTessellator

class, 4021
 methods, 4022
 AddPolygon, 4025
 Cleanup, 4027
 Init, 4028
 Reset, 4029
 Tessellate, 4030

IDLgrText

class, 4032
 methods, 4033
 Cleanup, 4049
 GetCTM, 4050
 GetProperty, 4052
 Init, 4053
 SetProperty, 4055
 properties, 4034

IDLgrView

class, 4056
 methods, 4057
 Add, 4065
 Cleanup, 4066
 GetByName, 4067
 GetProperty, 4069
 Init, 4070
 SetProperty, 4072
 properties, 4058

IDLgrViewgroup

class, 4073
 methods, 4073
 Add, 4077
 Cleanup, 4078
 GetByName, 4079
 GetProperty, 4081
 Init, 4082
 SetProperty, 4084

IDLgrViewgroup (*continued*)

properties, 4075

IDLgrVolume

class, 4085
 methods, 4086
 Cleanup, 4104
 ComputeBounds, 4105
 GetCTM, 4106
 GetProperty, 4108
 Init, 4109
 PickVoxel, 4111
 SetProperty, 4113
 properties, 4087

IDLgrVRML

class, 4114
 methods, 4116
 Draw, 4124
 GetDeviceInfo, 4125
 GetFontnames, 4127
 GetProperty, 4129
 GetTextDimensions, 4130
 Init, 4132
 SetProperty, 4134
 properties, 4117

IDLgrWindow

class, 4135
 maximum size, 4135
 methods, 4136
 Cleanup, 4149
 Draw, 4150
 Erase, 4152
 GetContiguousPixels, 4153
 GetDeviceInfo, 4154
 GetDimensions, 4156
 GetFontnames, 4158
 GetProperty, 4160
 GetTextDimensions, 4161
 Iconify, 4163
 Init, 4164
 PickData, 4166
 Read, 4169

IDLgrWindow (*continued*)

methods, 4136
 Select, 4170
 SetCurrentCursor, 4173
 SetCurrentZoom, 4175
 SetProperty, 4176
 Show, 4177
 ZoomIn, 4178
 ZoomOut, 4179
 properties, 4138

IDLitCommand

class, 3083
 methods, 3083
 AddItem, 3086
 Cleanup, 3088
 GetItem, 3089
 GetProperty, 3090
 GetSize, 3091
 Init, 3092
 SetProperty, 3094
 properties, 3085

IDLitCommandSet

class, 3095
 methods, 3095
 Cleanup, 3098
 GetSize, 3099
 Init, 3100
 properties, 3097

IDLitComponent

class, 3101
 methods, 3102
 Cleanup, 3106
 EditUserDefProperty, 3107
 GetFullIdentifier, 3109
 GetProperty, 3110
 GetPropertyAttribute, 3111
 GetPropertyByIdentifier, 3112
 Init, 3113
 QueryProperty, 3115
 RegisterProperty, 3116
 Restore, 3122

IDLitComponent (*continued*)

methods, 3102
 SetProperty, 3123
 SetPropertyAttribute, 3124
 SetPropertyByIdentifier, 3125
 UpdateComponentVersion, 3126
 properties, 3103

IDLitContainer

class, 3127
 methods, 3127
 Add, 3130
 AddByIdentifier, 3131
 Cleanup, 3132
 FindIdentifiers, 3133
 Get, 3135
 GetByIdentifier, 3137
 Init, 3138
 Remove, 3139
 RemoveByIdentifier, 3140
 properties, 3129

IDLitData

class, 3141
 methods, 3142
 AddDataObserver, 3145
 Cleanup, 3146
 Copy, 3147
 GetByType, 3148
 GetData, 3149
 GetProperty, 3150
 GetSize, 3151
 Init, 3152
 NotifyDataChange, 3154
 NotifyDataComplete, 3155
 RemoveDataObserver, 3156
 SetData, 3157
 SetProperty, 3158
 properties, 3143

IDLitDataContainer

class, 3159
 methods, 3159
 Add, 3162

IDLitDataContainer (*continued*)

methods, 3159
 Cleanup, 3163
 GetData, 3164
 GetIdentifiers, 3165
 GetProperty, 3166
 Init, 3167
 SetData, 3169
 SetProperty, 3171
 properties, 3161

IDLitDataOperation

class, 3172
 methods, 3173
 Cleanup, 3176
 DoExecuteUI, 3177
 Execute, 3179
 GetProperty, 3181
 Init, 3182
 SetProperty, 3184
 UndoExecute, 3185
 properties, 3175

IDLitIMessaging

class, 3187
 methods, 3188
 AddOnNotifyObserver, 3190
 DoOnNotify, 3192
 ErrorMessage, 3194
 GetTool, 3196
 ProbeStatusMessage, 3197
 ProgressBar, 3198
 PromptUserText, 3200
 PromptUserYesNo, 3201
 RemoveOnNotifyObserver, 3202
 SignalError, 3203
 StatusMessage, 3204
 properties, 3189

IDLitManipulator

class, 3205
 methods, 3206
 Cleanup, 3213
 CommitUndoValues, 3214

IDLitManipulator (*continued*)

methods, 3206
 GetCursorType, 3216
 GetProperty, 3218
 Init, 3219
 OnKeyboard, 3221
 OnLoseCurrentManipulator, 3223
 OnMouseDown, 3224
 OnMouseMove, 3226
 OnMouseUp, 3228
 RecordUndoValues, 3229
 RegisterCursor, 3231
 SetCurrentManipulator, 3233
 SetProperty, 3234
 properties, 3208

IDLitManipulatorContainer

class, 3235
 methods, 3235
 Add, 3238
 GetCurrent, 3239
 GetCurrentManipulator, 3240
 GetProperty, 3241
 Init, 3242
 OnKeyboard, 3244
 OnMouseDown, 3246
 OnMouseMove, 3248
 OnMouseUp, 3250
 SetCurrent, 3251
 SetCurrentManipulator, 3252
 SetProperty, 3253
 properties, 3237

IDLitManipulatorManager

class, 3254
 methods, 3254
 Add, 3257
 AddManipulatorObserver, 3258
 GetDefaultManipulator, 3259
 Init, 3260
 RemoveManipulatorObserver, 3262
 properties, 3256

IDLitManipulatorVisual

class, [3263](#)
 methods, [3263](#)
 Cleanup, [3266](#)
 GetProperty, [3267](#)
 Init, [3268](#)
 SetProperty, [3270](#)
 properties, [3265](#)

IDLitOperation

class, [3271](#)
 methods, [3272](#)
 Cleanup, [3277](#)
 DoAction, [3278](#)
 GetProperty, [3280](#)
 Init, [3281](#)
 RecordFinalValues, [3283](#)
 RecordInitialValues, [3285](#)
 RedoOperation, [3287](#)
 SetProperty, [3289](#)
 UndoOperation, [3290](#)
 properties, [3275](#)

IDLitParameter

class, [3292](#)
 methods, [3293](#)
 Cleanup, [3295](#)
 GetParameter, [3296](#)
 GetParameterAttribute, [3297](#)
 GetParameterSet, [3299](#)
 Init, [3300](#)
 OnDataChangeUpdate, [3301](#)
 OnDataDisconnect, [3303](#)
 QueryParameter, [3305](#)
 RegisterParameter, [3307](#)
 SetData, [3309](#)
 SetParameterAttribute, [3311](#)
 SetParameterSet, [3313](#)
 properties, [3294](#)

IDLitParameterSet

class, [3315](#)
 methods, [3315](#)
 Add, [3318](#)

IDLitParameterSet (*continued*)

methods, [3315](#)
 Cleanup, [3320](#)
 Copy, [3321](#)
 Get, [3322](#)
 GetByName, [3324](#)
 GetParameterName, [3326](#)
 Init, [3327](#)
 Remove, [3329](#)
 properties, [3317](#)

IDLitReader

class, [3330](#)
 methods, [3330](#)
 Cleanup, [3333](#)
 GetData, [3334](#)
 GetFileExtensions, [3335](#)
 GetFilename, [3336](#)
 GetProperty, [3337](#)
 Init, [3338](#)
 IsA, [3340](#)
 SetFilename, [3341](#)
 SetProperty, [3342](#)
 properties, [3332](#)

IDLITSYS_CREATETOOL function, [933](#)

IDLitTool

class, [3343](#)
 methods, [3344](#)
 ActivateManipulator, [3350](#)
 Add, [3351](#)
 AddService, [3352](#)
 Cleanup, [3353](#)
 CommitActions, [3354](#)
 DisableUpdates, [3355](#)
 DoAction, [3356](#)
 DoSetProperty, [3357](#)
 DoUIService, [3359](#)
 EnableUpdates, [3360](#)
 FindIdentifiers, [3361](#)
 GetCurrentManipulator, [3364](#)
 GetFileReader, [3365](#)
 GetFileWriter, [3366](#)

IDLitTool (*continued*)

methods, 3344

- GetManipulators, 3367
- GetOperations, 3368
- GetProperty, 3369
- GetSelectedItems, 3370
- GetService, 3371
- GetVisualization, 3372
- Init, 3374
- RefreshCurrentWindow, 3376
- Register, 3377
- RegisterCustomization, 3380
- RegisterFileReader, 3381
- RegisterFileWriter, 3383
- RegisterManipulator, 3386
- RegisterOperation, 3388
- RegisterStatusBarSegment, 3392
- RegisterVisualization, 3393
- SetProperty, 3395
- UnRegister, 3396
- UnRegisterCustomization, 3397
- UnRegisterFileReader, 3398
- UnRegisterFileWriter, 3399
- UnRegisterManipulator, 3400
- UnRegisterOperation, 3401
- UnRegisterStatusBarSegment, 3402
- UnRegisterVisualization, 3403

properties, 3347

IDLitToolbase

class, 3404

methods, 3405

IDLitUI

class, 3406

methods, 3406

- AddOnNotifyObserver, 3409
- Cleanup, 3411
- DoAction, 3412
- GetProperty, 3413
- GetTool, 3414
- GetWidgetByName, 3415
- Init, 3416

IDLitUI (*continued*)

methods, 3406

- RegisterUIService, 3417
- RegisterWidget, 3419
- RemoveOnNotifyObserver, 3421
- SetProperty, 3422
- UnRegisterUIService, 3423
- UnRegisterWidget, 3424

properties, 3408

IDLitVisualization

class, 3425

methods, 3426

- Add, 3432
- Aggregate, 3434
- BeginManipulation, 3435
- Cleanup, 3436
- EndManipulation, 3437
- Get, 3438
- GetCenterRotation, 3440
- GetCurrentSelectionVisual, 3442
- GetDataSpace, 3443
- GetDataString, 3444
- GetDefaultSelectionVisual, 3445
- GetManipulatorTarget, 3446
- GetProperty, 3447
- GetRequestedAxesStyle, 3448
- GetSelectionVisual, 3449
- GetTypes, 3450
- GetXYZRange, 3451
- Init, 3453
- Is3D, 3454
- IsIsotropic, 3455
- IsManipulatorTarget, 3456
- IsSelected, 3457
- Move, 3458
- On2DRotate, 3459
- OnAxesRequestChange, 3460
- OnAxesStyleRequestChange, 3461
- OnDataChange, 3462
- OnDataComplete, 3463
- OnDataRangeChange, 3464

IDLitVisualization (*continued*)

- methods, 3426
 - OnDimensionChange, 3465
 - OnWorldDimensionChange, 3466
 - Remove, 3467
 - RequestsAxes, 3468
 - Restore, 3469
 - Rotate, 3470
 - Scale, 3472
 - Select, 3474
 - Set3D, 3476
 - SetAxesRequest, 3477
 - SetAxesStyleRequest, 3479
 - SetCurrentSelectionVisual, 3481
 - SetData, 3482
 - SetDefaultSelectionVisual, 3483
 - SetParameterSet, 3484
 - SetProperty, 3485
 - UpdateSelectionVisual, 3486
 - VisToWindow, 3487
 - WindowToVis, 3489
- properties, 3429

IDLitWindow

- class, 3491
- methods, 3491
 - Add, 3494
 - AddWindowEventObserver, 3495
 - Cleanup, 3496
 - ClearSelections, 3497
 - DoHitTest, 3498
 - GetEventMask, 3500
 - GetProperty, 3502
 - GetSelectedItems, 3503
 - Init, 3504
 - OnKeyboard, 3506
 - OnMouseDown, 3509
 - OnMouseMove, 3511
 - OnMouseUp, 3513
 - OnScroll, 3515
 - Remove, 3516
 - RemoveWindowEventObserver, 3517

IDLitWindow (*continued*)

- methods, 3491
 - SetCurrentZoom, 3518
 - SetEventMask, 3519
 - SetManipulatorManager, 3521
 - SetProperty, 3522
 - ZoomIn, 3523
 - ZoomOut, 3524
 - properties, 3493
- IDLitWriter
- class, 3525
 - methods, 3525
 - Cleanup, 3528
 - GetFileExtensions, 3529
 - GetFilename, 3530
 - GetProperty, 3531
 - Init, 3532
 - IsA, 3534
 - SetData, 3535
 - SetFilename, 3536
 - SetProperty, 3537
 - properties, 3527
- IDLjavaObject
- class, 4223
 - methods, 4223
 - GetProperty, 4226
 - Init, 4227
 - SetProperty, 4229
 - properties, 4225, 4225
- IEXP machine-specific parameter, 1287
- IF...THEN...ELSE statement, 937
- IGAMMA function, 939
- IIMAGE procedure, 942
- image object, 3695
- IMAGE_CONT procedure, 958
- IMAGE_STATISTICS procedure, 960
- images
 - annotating, 103
 - bi-level, 2106
 - color channel, 945, 3700
 - copying areas, 4254

- images (*continued*)
 - defining region of interest, 519
 - displaying
 - color table, 2171
 - returning array, 2174
 - scrolling display, 1956
 - zooming, 487
 - displaying (FLICK), 734
 - displaying (TV), 2165
 - displaying (TVSCL), 2178
 - displaying with intensity scaling, 2178
 - dissolve effect, 577
 - interactive (iTool) routine, 942, 965
 - JPEG, 1725, 2514
 - magnified, 2659, 2661
 - monochrome, 4288
 - MPEG files
 - closing, 1466
 - opening, 1467
 - saving, 1473
 - storing image frame, 1471
 - object, 3695
 - profiling, 1606
 - profiling, interactive, 1612
 - reading from display, 2174
 - region labeling, 1211
 - Roberts edge enhancement, 1815
 - rotating, 1820
 - searching for objects, 1858
 - sharing data, 3708
 - smoothing, 1960
 - Sobel edge enhancement, 1963
 - thinning, 2106
 - transfer direction, 4379
 - TrueColor, 2175
 - warping, 1567
 - warping to maps
 - MAP_IMAGE function, 1313
 - MAP_PATCH function, 1318
 - with surface and contour plots, 1912
 - zooming, 487
- IMAGINARY function, 963
- imaginary part of complex numbers, 963
- IMAP procedure, 965
- INCHES keyword, 4264
- incomplete
 - beta function, 894
 - gamma function, 939
- increment operator, 4392
- incrementing array elements, 878
- INDEX_COLOR keyword, 4264
- INDGEN function, 990
- Infinity norm, 1484
- initialization of objects, 2670
- INP, *see* obsolete routines
- input/output
 - associated variables, 134
 - bitmap files, 1714
 - BMP files, 2507
 - closing files, 263
 - emptying buffers
 - EMPTY, 604
 - FLUSH, 743
 - errors, 1498
 - formatted, using PRINT/PRINTF, 1596
 - GIF files, 1719, 2510
 - Interfile files, 1723
 - JPEG files
 - reading, 1725
 - writing, 2514
 - NRIF files, 2519
 - opening files, 1511
 - PGM files
 - reading, 1741
 - writing, 2526
 - PICT files
 - reading, 1736
 - writing, 2521
 - PPM files
 - reading, 1741
 - writing, 2526

input/output (*continued*)

- reading
 - ASCII files, 1709
 - formatted data, 1706
 - formatted data from a string, 1766
 - from a prompt, 1707
 - unformatted binary data, 1768
- SRF files
 - reading, 1744
 - writing, 2530
- TIFF files
 - reading, 1750
 - writing, 2534
- unformatted binary data, writing, 2548
- wave files
 - reading, 1760
 - writing, 2546
- X11 Bitmaps, 1762
- XWD files, 1764
- INT_2D function, 993
- INT_3D function, 997
- INT_TABULATED function, 1000
- INTARR function, 1002
- integer
 - arrays
 - (INDGEN), 990
 - converting to, 731
 - arrays(INTARR), 1002
 - data type, converting to, 731
- integration
 - INT_2D, 993
 - INT_3D, 997
 - INT_TABULATED, 1000
 - QROMB, 1640
 - QROMO, 1645
 - QSIMP, 1648
 - RK4, 1812
 - tabulated functions, 1000
 - univariate functions, 1640, 1645, 1648
- Interfile files, reading, 1723
- Internet socket support, 1965

- INTERPOL function, 1004
- INTERPOLATE function, 1007
- interpolation
 - bilinear, 168
 - cubic convolution, 1008
 - cubic convolution in warping, 1568
 - cubic spline
 - SPL_INIT function, 1988
 - SPLINE function, 1992
 - SPLINE_P procedure, 1994
 - dependent variable to volume, 1630
 - INTERPOLATE function, 1007
 - irregularly-sampled data over earth, 2132
 - KRIG2D, 1117
 - MIN_CURVE_SURF, 1433
 - of irregularly-gridded data
 - KRIG2D function, 1117
 - MIN_CURVE_SURF function, 1433
 - TRIGRID function, 2136
 - POLAR_SURFACE, 1564
 - quintic, 2139
 - scattered data to regular, 808
 - spherical, 1982
 - SPL_INIT, 1988
 - SPL_INTERP, 1990
 - thin-plate-spline
 - GRID_TPS, 801
 - MIN_CURVE_SURF, 1433
- INTERVAL_VOLUME procedure, 1011
- invalid widget ID's, 2381
- inverse
 - cosine, 90
 - of a complex array or matrix, 1273
 - sine, 132
 - subspace iteration, 599
 - tangent, 137
- INVERT function, 1016
- IOCTL function, 1018
- IPLLOT procedure, 1021
- IRND machine-specific parameter, 1286
- irregularly-gridded data, 2132, 2136

- ISHFT function, [1041](#)
 - ISO Latin 1 encoding, [4417](#)
 - ISOCONTOUR procedure, [1043](#)
 - ISOLATIN1 keyword, [4264](#)
 - ISOSURFACE procedure, [1048](#)
 - isosurfaces, displaying, [1884](#)
 - ISURFACE procedure, [1052](#)
 - IT machine-specific parameter, [1286](#)
 - ITALIC keyword, [4264](#)
 - ITCURRENT procedure, [1073](#)
 - ITDELETE procedure, [1075](#)
 - iterative
 - biconjugate gradient, [1224](#)
 - Gaussian quadrature
 - double integral, [993](#)
 - trivariate function integral, [997](#)
 - improvement of a solution, [1277](#)
 - ITGETCURRENT function, [1077](#)
 - iTools
 - classes
 - component (class) base, [3101](#)
 - component collection, [3127](#)
 - data collection, [3159](#)
 - data undo and redo, [3172](#)
 - manipulating objects, [3205](#)
 - manipulator base, [3254](#)
 - manipulator collection, [3235](#)
 - messaging, [3187](#)
 - naming data objects, [3315](#)
 - operating tasks, [3271](#)
 - parameters, [3292](#)
 - reading files, [3330](#)
 - storing data, [3141](#)
 - tool base, [3343](#), [3404](#)
 - undo and redo commands, [3083](#)
 - user-interface, [3406](#)
 - visual base, [3425](#)
 - window base, [3491](#)
 - writing files, [3525](#)
 - command collection, [3095](#)
 - creating, [933](#)
 - iTools (*continued*)
 - current (active), [1073](#)
 - deleting, [1075](#)
 - displaying properties, [2424](#)
 - registering, [1079](#)
 - resetting, [1083](#)
 - retrieving current, [1077](#)
 - routines
 - contours, [898](#)
 - creating tools, [933](#)
 - current (active) tool, [1073](#)
 - deleting tools, [1075](#)
 - image, [942](#), [965](#)
 - plot, [1021](#)
 - property sheet, [2424](#)
 - registering tools, [1079](#)
 - resetting tools, [1083](#)
 - retrieving current tool, [1077](#)
 - surface, [1052](#)
 - volume, [1087](#)
 - ITREGISTER procedure, [1079](#)
 - ITRESET procedure, [1083](#)
 - ITRESOLVE procedure, [1085](#)
 - IVOLUME procedure, [1087](#)
- ## J
- Java, objects, [4223](#)
 - JFIF, *see* JPEG
 - JOIN, *see* obsolete routines
 - JOURNAL procedure, [1110](#)
 - JPEG files
 - reading, [1725](#)
 - writing, [2514](#)
 - JULDAY function, [1112](#)
 - Julian date, converting to calendar, [212](#)
 - Julian date definition, [2111](#)
 - Julian dates/time, generating, [2111](#)

K

Kendall's tau rank correlation, [1679](#)
 kernel, convolving an array with, [331](#)
 keyboard
 defining keys, [502](#)
 defining keys for different keyboards, [1870](#)
 focus events
 base widget, [2262](#)
 widget control, [2324](#)
 widget info, [2390](#)
 widget table, [2467](#)
 widget text, [2483](#)
 numeric keypads, [1872](#)
 returning characters from, [786](#)
 KEYWORD_SET function, [1115](#)
 keywords
 arguments, checking existence of, [119](#)
 described, [61](#), [2667](#)
 graphics, [4333](#)
 meaning of slash character, [2668](#)
 setting, [2667](#)
 KMEANS, *see* obsolete routines
 KRIG2D function, [1117](#)
 kriging, [1117](#)
 KRUSKAL_WALLIS, *see* obsolete routines
 Kruskal-Wallis H-Test, [1124](#)
 kurtosis
 KURTOSIS function, [1122](#)
 MOMENT function, [1443](#)
 KURTOSIS function, [1122](#)
 KW_TEST function, [1124](#)

L

L64INDGEN function, [1127](#)
 LA_CHOLDC procedure, [1129](#)
 LA_CHOLMPROVE function, [1132](#)
 LA_CHOLSOL function, [1136](#)
 LA_DETERM function, [1139](#)
 LA_EIGENPROBLEM function, [1141](#)

LA_EIGENQL function, [1147](#)
 LA_EIGENVEC function, [1153](#)
 LA_ELMHES function, [1157](#)
 LA_GM_LINEAR_MODEL function, [1160](#)
 LA_HQR function, [1163](#)
 LA_INVERT function, [1166](#)
 LA_LEAST_SQUARE_EQUALITY function, [1168](#)
 LA_LEAST_SQUARES function, [1171](#)
 LA_LINEAR_EQUATION function, [1175](#)
 LA_LUDC procedure, [1178](#)
 LA_LUMPROVE function, [1181](#)
 LA_LUSOL function, [1184](#)
 LA_SVD procedure, [1187](#)
 LA_TRIDC procedure, [1191](#)
 LA_TRIMPROVE function, [1195](#)
 LA_TRIQL procedure, [1199](#)
 LA_TRIRED procedure, [1202](#)
 LA_TRISOL function, [1204](#)
 label widgets, WIDGET_LABEL, [2407](#)
 LABEL_DATE function, [1207](#)
 LABEL_REGION function, [1211](#)
 labeling, regions regions, [1211](#)
 LADFIT function, [1214](#)
 lagged
 autocorrelation, [87](#)
 cross correlation, [210](#)
 LAGUERRE function, [1217](#)
 Laguerre polynomials, [1217](#)
 Laguerre's method, [763](#)
 Lambert's conformal conic map projection, [1355](#)
 Lambert's equal-area map projection, [1356](#)
 LANDSCAPE keyword, [4264](#)
 landscape orientation
 for IDL plots (LANDSCAPE keyword), [4264](#)
 PostScript positioning, [4305](#)
 laser printers, [4302](#)
 LATLON, *see* obsolete routines
 LE operator, [4399](#)

- least absolute deviation, [1214](#)
- least squares fit
 - CURVEFIT function, [374](#)
 - GAUSSFIT function, [777](#)
 - POLY_FIT function, [1573](#)
 - SVDFIT function, [2068](#)
- LEEFILT function, [1219](#)
- legend object, [3719](#)
- LEGENDRE function, [1221](#)
- Legendre polynomials, [1221](#)
- LEGO, *see* obsolete routines
- length of strings, [2031](#)
- LIGHT keyword, [4265](#)
- light object, [3740](#)
- light source
 - IDLgrLight object, [3740](#)
 - shading, [1867](#)
- LINBCG function, [1224](#)
- LINDGEN function, [1227](#)
- line
 - drawing
 - method for contours, [315](#)
 - PLOTS procedure, [1555](#)
 - editing, enabling and disabling, [4367](#)
 - interval, [4375](#)
 - styles, [4338](#)
- linear
 - interpolation, [1007](#)
 - linear-log plots, [1545](#)
 - regression, [1790](#)
- linear algebra
 - CHOLDC, [255](#)
 - CHOLSOL, [257](#)
 - COND, [300](#)
 - CRAMER, [352](#)
 - DETERM, [535](#)
 - EIGENVEC, [599](#)
 - ELMHES, [602](#)
 - GS_ITER, [833](#)
 - HQR, [890](#)
 - INVERT, [1016](#)
- linear algebra (*continued*)
 - LA_CHOLDC, [1129](#)
 - LA_CHOLMPROVE, [1132](#)
 - LA_CHOLSOL, [1136](#)
 - LA_DETERM, [1139](#)
 - LA_EIGENPROBLEM, [1141](#)
 - LA_EIGENVEC, [1153](#)
 - LA_ELMHES, [1157](#)
 - LA_HQR, [1163](#)
 - LA_INVERT, [1166](#)
 - LA_LUDC, [1178](#)
 - LA_LUMPROVE, [1181](#)
 - LA_LUSOL, [1184](#)
 - LA_SVD, [1187](#)
 - LA_TRIDC, [1191](#)
 - LA_TRIQL, [1199](#)
 - LA_TRIRED, [1202](#)
 - LA_TRISOL, [1204](#)
 - LINBCG, [1224](#)
 - LU_COMPLEX, [1273](#)
 - LUDC, [1275](#)
 - LUMPROVE, [1277](#)
 - LUSOL, [1280](#)
 - NORM, [1484](#)
 - SVDC, [2065](#)
 - SVSOL, [2074](#)
 - TRIQL, [2146](#)
 - TRIRED, [2149](#)
 - TRISOL, [2151](#)
- linear model, Gauss-Markov, [1160](#)
- linear programming solutions, [1916](#)
- lines, counting, [685](#)
- LINestyle keyword, [4337](#)
- LINestyle system variable field, [4380](#)
- linestyles, table of, [4338](#)
- LINFIT function, [1229](#)
- LINKIMAGE procedure
 - alternative to, [217](#)
 - using, [1232](#)
- linking
 - C code with IDL, [1292](#)

- linking (*continued*)
 - dynamically, [1292](#)
- list widgets
 - determining
 - selected element, , [2391](#)
 - topmost element, [2391](#)
 - double-clicks, [2422](#)
 - events returned by, [2422](#)
 - number, [2391](#)
 - selecting multiple items
 - setting, [2416](#)
 - WIDGET_INFO, [2390](#)
 - setting, [2333](#)
 - WIDGET_LIST, [2414](#)
- LISTREP, *see* obsolete routines
- LISTWISE, *see* obsolete routines
- little endian byte ordering
 - SOCKET procedure, [1968](#)
 - swapping with big endian, [2076](#)
- LIVE_CONTOUR, *see* obsolete routines
- LIVE_CONTROL, *see* obsolete routines
- LIVE_DESTROY, *see* obsolete routines
- LIVE_EXPORT, *see* obsolete routines
- LIVE_IMAGE, *see* obsolete routines
- LIVE_INFO, *see* obsolete routines
- LIVE_LINE, *see* obsolete routines
- LIVE_LOAD, *see* obsolete routines
- LIVE_OPLOT, *see* obsolete routines
- LIVE_PLOT, *see* obsolete routines
- LIVE_PRINT, *see* obsolete routines
- LIVE_RECT, *see* obsolete routines
- LIVE_STYLE, *see* obsolete routines
- LIVE_SURFACE, *see* obsolete routines
- LIVE_TEXT, *see* obsolete routines
- LJLCT, *see* obsolete routines
- LL_ARC_DISTANCE function, [1236](#)
- LMFIT function, [1238](#)
- LMGR function, [1243](#)
- LN03, *see* obsolete routines
- LNGAMMA function, [1246](#)
- LNP_TEST function, [1248](#)
- LoadCT method, IDLgrPalette, [3805](#)
- LOADCT procedure, [1251](#)
- loading color tables, [2171](#)
- LOCALE_GET function, [1253](#)
- logarithm
 - base 10, [97](#)
 - natural, [95](#)
 - of the gamma function, [1246](#)
- logarithmic axes, [325](#)
- logging an IDL session, [1110](#)
- Logical operators, [4396](#)
- logical unit number, SOCKET procedure, [1966](#)
- logical unit numbers
 - !D system variable field, [4377](#)
 - allocating, [788](#)
 - freeing, [748](#)
 - FSTAT function, [750](#)
 - getting, [1513](#)
 - journal file, [4367](#)
 - obtaining status information, [750](#)
 - returning information about, [861](#)
 - setting file position pointer, [1560](#)
- LOGICAL_AND function, [1254](#)
- LOGICAL_OR function, [1256](#)
- LOGICAL_TRUE function, [1258](#)
- log-linear plots
 - AXIS, [142](#)
 - CONTOUR, [325](#)
 - PLOT, [1545](#)
 - SHADE_SURF, [1879](#)
 - SURFACE, [2062](#)
- Lomb Normalized Periodogram, [1248](#)
- LON64ARR function, [1260](#)
- LONARR function, [1262](#)
- LONG function, [1264](#)
- LONG64 function, [1266](#)
- longjmp, C language, [236](#)
- longword
 - arrays
 - LINDGEN function, [1227](#)
 - LONARR function, [1262](#)

longword (*continued*)

arrays

ULON64ARR, 2191

data type, converting to, 1264

unsigned arrays, 2189

lossy compression

READ_JPEG procedure, 1725

WRITE_JPEG procedure, 2514

lower margin, setting, 4385

lowercase, converting strings to, 2032

LSODE function, 1268

LT operator, 4399

LU decomposition

LA_LUDC procedure, 1178

LA_LUSOL function, 1184

LA_TRIDC function, 1191

LU_COMPLEX function, 1273

LUDC procedure, 1275

LUSOL function, 1280

LU_COMPLEX function, 1273

LUBKSB, *see* obsolete routines

LUDC procedure, 1275

LUDCMP, *see* obsolete routines

luminance, 366

LUMPROVE function, 1277

LUN

freeing, 748

TCP/IP socket, 1965

LUSOL function, 1280

M

M_CORRELATE function, 1283

MACHAR function, 1286

MACHEP machine-specific parameter, 1286

machine-specific parameters, 1286

magnifying arrays, 1771

magnitude, returning for complex number, 88

magnitude-based ranks, 1703

MAKE_ARRAY function, 1288

MAKE_DLL procedure, 1292

MAKETREE, *see* obsolete routines

MANN_WHITNEY, *see* obsolete routines

Mann-Whitney U-Test, 1828

map coordinates, transforming

Cartesian to lat/lon, 1351

lat/lon to Cartesian, 1322

map projections

Aitoff, 1355

Alber's equal area conic, 1355

azimuthal equidistant, 1355

cylindrical equidistant, 1355

drawing boundaries over, 1302

drawing continent boundaries, 1358

drawing parallels and meridians, 1307

gnomonic (central, gnomonic), 1355

Hammer-Aitoff, 1355

Lambert's conformal conic, 1355

Lambert's equal area, 1356

MAP_PROJ_INIT function, 1335

Mercator, 1356

Miller, 1356

Mollweide, 1356

orthographic, 1356

satellite, 1356

setting, 1353

sinusoidal, 1357

stereographic, 1357

Transverse Mercator (UTM), 1357

warping images to maps

MAP_IMAGE function, 1313

MAP_PATCH function, 1318

MAP_2POINTS function, 1298

MAP_CONTINENTS procedure, 1302

MAP_GRID procedure, 1307

MAP_IMAGE function, 1313

MAP_PATCH function, 1318

MAP_PROJ_FORWARD function, 1322

MAP_PROJ_IMAGE function, 1327

MAP_PROJ_INFO procedure, 1332

MAP_PROJ_INIT function, 1335

MAP_PROJ_INVERSE function, 1351

MAP_SET procedure, [1353](#)

mapping widgets, [2263](#)

MARGIN system variable field, [4385](#)

margins

setting for multi-plot window, [4385](#)

setting for single plot, [4385](#)

marquee selector, [188](#)

mathematical operators, table of, [4392](#)

matrices

DIAG_MATRIX, [545](#)

MATRIX_MULTIPLY, [1365](#)

MATRIX_POWER, [1368](#)

multiplication example, [4395](#)

matrix operators

CHOLDC, [255](#)

CHOLSOL, [257](#)

COND, [300](#)

CRAMER, [352](#)

DETERM, [535](#)

EIGENVEC, [599](#)

ELMHES, [602](#)

GS_ITER, [833](#)

HQR, [890](#)

INVERT, [1016](#)

LA_CHOLDC, [1129](#)

LA_CHOLMPROVE, [1132](#)

LA_CHOLSOL, [1136](#)

LA_DETERM, [1139](#)

LA_EIGENPROBLEM, [1141](#)

LA_EIGENVEC, [1153](#)

LA_ELMHES, [1157](#)

LA_HQR, [1163](#)

LA_INVERT, [1166](#)

LA_LUDC, [1178](#)

LA_LUMPROVE, [1181](#)

LA_LUSOL, [1184](#)

LA_SVD, [1187](#)

LA_TRIDC, [1191](#)

LA_TRIQL, [1199](#)

LA_TRIRED, [1202](#)

LA_TRISOL, [1204](#)

matrix operators (*continued*)

LU_COMPLEX, [1273](#)

LUDC, [1275](#)

LUMPROVE, [1277](#)

LUSOL, [1280](#)

NORM, [1484](#)

SVDC, [2065](#)

SVSOL, [2074](#)

TRIQL, [2146](#)

TRIRED, [2149](#)

TRISOL, [2151](#)

See also sparse arrays

MATRIX_MULTIPLY function, [1365](#)

MATRIX_POWER function, [1368](#)

MAX function, [1370](#)

MAXEXP machine-specific parameter, [1287](#)

maximum operator, [4394](#)

maximum size, draw area, [4135](#)

maximum value

for slider widgets, [2443](#)

of an array, [1370](#)

MD_TEST function, [1374](#)

mean

absolute deviation, [1378](#)

MOMENT function, [1443](#)

of distribution, [1124](#)

MEAN function, [1376](#)

MEANABSDEV function, [1378](#)

median

Median Delta Test, [1374](#)

MOMENT function, [1443](#)

smoothing, [1380](#)

MEDIAN function, [1380](#)

MEDIUM keyword, [4265](#)

memory

conserving, [2099](#)

dynamic memory in use, [863](#)

MEMORY function, [1383](#)

menu bars, [2263](#)

menus

displaying context-sensitive, [2351](#)

menus (*continued*)

- menu bars, 2263
- pulldown menu button, 2286
- MENUS, *see* obsolete routines
- Mercator map projection, 1356
- merging meshes, 1401
- meridians, drawing, 1307
- mesh plots, 2058
- MESH_CLIP function, 1387
- MESH_DECIMATE function, 1392
- MESH_ISSOLID function, 1399
- MESH_MERGE function, 1400
- MESH_NUMTRIANGLES function, 1405
- MESH_OBJ procedure, 1406
- MESH_SMOOTH function, 1413
- MESH_SURFACEAREA function, 1418
- MESH_VALIDATE function, 1420
- MESH_VOLUME function, 1422
- message dialogs, 547
- MESSAGE procedure, 1424
- messages, suppressing informational, 4372
- Metafile, 4244
- methods

IDL_Container

- Add, 4184
- Cleanup, 4185
- Count, 4186
- Get, 4187
- Init, 4189
- IsContained, 4190
- Move, 4191
- Remove, 4192

IDL_Savefile

- Cleanup, 4197
- Contents, 4198
- Init, 4201
- Names, 4203
- Restore, 4206
- Save, 4213

IDLanROI

- AppendData, 2686
- Cleanup, 2688

methods (*continued*)

IDLanROI

- ComputeGeometry, 2689
- ComputeMask, 2691
- ContainsPoints, 2694
- GetProperty, 2696
- Init, 2697
- RemoveData, 2699
- ReplaceData, 2701
- Rotate, 2704
- Scale, 2705
- SetProperty, 2706
- Translate, 2707

IDLanROIgroup

- Add, 2712
- Cleanup, 2713
- ComputeMask, 2714
- ComputeMesh, 2717
- ContainsPoints, 2719
- GetProperty, 2721
- Init, 2722
- Rotate, 2723
- Scale, 2724
- Translate, 2725

IDLcomIDispatch

- GetProperty, 4220
- Init, 4221
- SetProperty, 4222

IDLffDICOM

- Cleanup, 2735
- DumpElements, 2736
- GetChildren, 2737
- GetDescription, 2739
- GetElement, 2741
- GetGroup, 2743
- GetLength, 2745
- GetParent, 2747
- GetPreamble, 2749
- GetReference, 2750
- GetValue, 2752
- GetVR, 2755
- Init, 2757

methods (*continued*)

IDLffDICOM
 Read, 2759
 Reset, 2760
 IDLffDXF
 Cleanup, 2764
 GetContents, 2765
 GetEntity, 2768
 GetPalette, 2781
 Init, 2782, 2782
 PutEntity, 2783
 Read, 2784
 RemoveEntity, 2785
 Reset, 2786
 SetPalette, 2787
 Write, 2788
 IDLffJPEG2000
 Cleanup, 2806
 GetData, 2807
 GetProperty, 2811
 GetTileProperty, 2812
 Init, 2814
 SetData, 2816
 SetProperty, 2820
 IDLffLangCat
 AppendCatalog, 2827
 Cleanup, 2829
 GetProperty, 2830
 Init, 2831
 Query, 2833
 SetProperty, 2834
 IDLffMrSID
 Cleanup, 2838
 GetDimsAtLevel, 2839
 GetImageData, 2841
 GetProperty, 2844
 Init, 2847
 IDLffShape
 AddAttribute, 2865
 Cleanup, 2868
 Close, 2869

methods (*continued*)

IDLffShape
 DestroyEntity, 2870
 GetAttributes, 2872
 GetEntity, 2874
 GetProperty, 2876
 Init, 2878
 Open, 2880
 PutEntity, 2882
 SetAttributes, 2884
 IDLffXMLDOMAttr
 Cleanup, 2891
 GetName, 2892
 GetSpecified, 2893
 GetValue, 2894
 Init, 2895
 SetValue, 2896
 IDLffXMLDOMCDATASection
 Cleanup, 2899
 Init, 2900
 IDLffXMLDOMCharacterData
 AppendData, 2904
 Cleanup, 2905
 DeleteData, 2906
 GetData, 2907
 GetLength, 2908
 Init, 2909
 InsertData, 2910
 ReplaceData, 2911
 SetData, 2912
 SubstringData, 2913
 IDLffXMLDOMComment
 Cleanup, 2916
 Init, 2917, 2917
 IDLffXMLDOMDocument
 Cleanup, 2921
 CreateAttribute, 2922
 CreateCDATASection, 2923
 CreateComment, 2924
 CreateElement, 2926
 CreateEntityReference, 2927

methods (*continued*)

- IDLffXMLDOMDocument
 - CreateProcessingInstruction, 2928
 - CreateTextNode, 2929
 - DocumentFragment, 2925
 - GetDoctype, 2930
 - GetDocumentElement, 2931
 - GetElementsByTagName, 2932
 - Init, 2933
 - Load, 2936
 - Save, 2939
- IDLffXMLDOMDocumentFragment
 - Cleanup, 2943
 - Init, 2944
- IDLffXMLDOMDocumentType
 - Cleanup, 2947
 - GetEntities, 2948
 - GetName, 2949
 - GetNotations, 2950
 - Init, 2951
- IDLffXMLDOMElement
 - Cleanup, 2955
 - GetAttribute, 2956
 - GetAttributeNode, 2957
 - GetElementsByTagName, 2958
 - GetTagName, 2959
 - Init, 2960
 - RemoveAttribute, 2961
 - RemoveAttributeNode, 2962
 - SetAttribute, 2963
 - SetAttributeNode, 2964
- IDLffXMLDOMEntity
 - Cleanup, 2967
 - GetNotationname, 2968
 - GetPublicId, 2969
 - GetSystemId, 2970
 - Init, 2971
- IDLffXMLDOMEntityReference
 - Cleanup, 2974
 - Init, 2975

methods (*continued*)

- IDLffXMLDOMNamedNodeMap
 - Cleanup, 2979
 - GetLength, 2980
 - GetNamedItem, 2981
 - Init, 2982
 - Item, 2983
 - RemoveNamedItem, 2984
 - SetNamedItem, 2985
- IDLffXMLDOMNode
 - AppendChild, 2989
 - Cleanup, 2990
 - CloneNode, 2991
 - GetAttributes, 2992
 - GetChildNodes, 2993
 - GetFirstChild, 2994
 - GetLastChild, 2995
 - GetNextSibling, 2996
 - GetNodeName, 2997
 - GetNodeType, 2999
 - GetNodeValue, 3001
 - GetOwnerDocument, 3003
 - GetParentNode, 3004
 - GetPreviousSibling, 3005
 - HasChildNodes, 3006
 - Init, 3007
 - InsertBefore, 3008
 - RemoveChild, 3010
 - ReplaceChild, 3011
 - SetNodeValue, 3013
- IDLffXMLDOMNodeList
 - Cleanup, 3016
 - GetLength, 3017
 - Init, 3018
 - Item, 3019
- IDLffXMLDOMNotation
 - Cleanup, 3022
 - GetPublicID, 3023
 - GetSystemID, 3024
 - Init, 3025

methods (*continued*)

IDLffXMLDOMProcessingInstruction

Cleanup, [3028](#)

GetData, [3029](#)

GetTarget, [3030](#)

Init, [3031](#)

SetData, [3032](#)

IDLffXMLDOMText

Cleanup, [3035](#)

Init, [3036](#)

IsIgnorableWhitespace, [3037](#)

SplitText, [3038](#)

IDLffXMLSAX

AttributeDecl, [3046](#)

Characters, [3048](#)

Cleanup, [3049](#)

Comment, [3050](#)

ElementDecl, [3051](#)

EndCDATA, [3052](#)

EndDocument, [3053](#)

EndDTD, [3054](#)

EndElement, [3055](#)

EndEntity, [3056](#)

EndPrefixMapping, [3057](#)

Error, [3058](#)

ExternalEntityDecl, [3060](#)

FatalError, [3061](#)

GetProperty, [3062](#)

IgnorableWhitespace, [3063](#)

Init, [3064](#)

InternalEntityDecl, [3065](#)

NotationDecl, [3066](#)

ParseFile, [3067](#)

ProcessingInstruction, [3068](#)

SetProperty, [3069](#)

SkippedEntity, [3070](#)

StartCDATA, [3071](#)

StartDocument, [3072](#)

StartDTD, [3073](#)

StartElement, [3074](#)

StartEntity, [3076](#)

methods (*continued*)

IDLffXMLSAX

StartPrefixmapping, [3077](#)

StopParsing, [3078](#)

UnparsedEntityDecl, [3079](#)

Warning, [3080](#)

IDLgrAxis

Cleanup, [3565](#)

GetCTM, [3566](#)

GetProperty, [3568](#)

Init, [3569](#)

SetProperty, [3571](#)

IDLgrBuffer

Cleanup, [3579](#)

Draw, [3580](#)

Erase, [3581](#)

GetContiguousPixels, [3582](#)

GetDeviceInfo, [3583](#)

GetFontnames, [3585](#)

GetProperty, [3587](#)

GetTextDimensions, [3588](#)

Init, [3590](#)

PickData, [3592](#)

Read, [3595](#)

Select, [3596](#)

SetProperty, [3599](#)

IDLgrClipboard

Cleanup, [3607](#)

Draw, [3608](#)

GetContiguousPixels, [3616](#)

GetDeviceInfo, [3617](#)

GetProperty, [3621](#)

GetTextDimensions, [3622](#)

Init, [3624](#)

IDLgrColorbar

Cleanup, [3640](#)

ComputeDimensions, [3641](#)

GetProperty, [3643](#)

Init, [3644](#)

SetProperty, [3646](#)

methods (*continued*)

- IDLgrContour
 - AdjustLabelOffsets, [3677](#)
 - Cleanup, [3678](#)
 - GetCTM, [3679](#)
 - GetLabelInfo, [3681](#)
 - GetProperty, [3683](#)
 - Init, [3684](#)
 - SetProperty, [3686](#)
- IDLgrFont
 - Cleanup, [3690](#)
 - GetProperty, [3691](#)
 - Init, [3692](#)
 - SetProperty, [3694](#)
- IDLgrImage
 - Cleanup, [3712](#)
 - GetCTM, [3713](#)
 - GetProperty, [3715](#)
 - Init, [3716](#)
 - SetProperty, [3718](#)
- IDLgrLegend
 - Cleanup, [3733](#)
 - ComputeDimensions, [3734](#)
 - GetProperty, [3736](#)
 - Init, [3737](#)
 - SetProperty, [3739](#)
- IDLgrLight
 - Cleanup, [3749](#)
 - GetCTM, [3750](#)
 - GetProperty, [3752](#)
 - Init, [3753](#)
 - SetProperty, [3755](#)
- IDLgrModel
 - Add, [3764](#)
 - Cleanup, [3765](#)
 - Draw, [3766](#)
 - GetByName, [3767](#)
 - GetCTM, [3769](#)
 - GetProperty, [3771](#)
 - Init, [3772](#)
 - Reset, [3774](#)

methods (*continued*)

- IDLgrModel
 - Rotate, [3775](#)
 - Scale, [3776](#)
 - SetProperty, [3777](#)
- IDLgrMPEG
 - Cleanup, [3788](#)
 - GetProperty, [3789](#)
 - Init, [3790](#)
 - Put, [3792](#)
 - Save, [3793](#)
 - SetProperty, [3794](#)
- IDLgrPalette
 - Cleanup, [3800](#)
 - GetProperty, [3802](#)
 - GetRGB, [3801](#)
 - Init, [3803](#)
 - LoadCT, [3805](#)
 - NearestColor, [3806](#)
 - SetProperty, [3808](#)
 - SetRGB, [3807](#)
- IDLgrPattern
 - Cleanup, [3814](#)
 - GetProperty, [3815](#)
 - Init, [3816](#)
 - SetProperty, [3818](#)
- IDLgrPlot
 - Cleanup, [3837](#)
 - GetCTM, [3838](#)
 - GetProperty, [3840](#)
 - Init, [3841](#)
 - SetProperty, [3843](#)
- IDLgrPolygon
 - Cleanup, [3868](#)
 - GetCTM, [3869](#)
 - GetProperty, [3871](#)
 - Init, [3872](#)
 - SetProperty, [3874](#)
- IDLgrPolyline
 - Cleanup, [3895](#)
 - GetCTM, [3896](#)

methods (*continued*)

- IDLgrPolyline
 - GetProperty, 3898
 - Init, 3899
 - SetProperty, 3901
- IDLgrPrinter
 - Cleanup, 3910
 - Draw, 3911
 - GetContiguousPixels, 3916
 - GetFontnames, 3917
 - GetProperty, 3919
 - GetTextDimensions, 3920
 - Init, 3922
 - NewDocument, 3924
 - NewPage, 3925
 - SetProperty, 3926
- IDLgrROI
 - Cleanup, 3939
 - GetProperty, 3940
 - Init, 3941
 - PickVertex, 3943
 - SetProperty, 3945
- IDLgrROIGroup
 - Add, 3955
 - Cleanup, 3956
 - GetProperty, 3957
 - Init, 3958
 - PickRegion, 3960
 - SetProperty, 3962
- IDLgrScene
 - Add, 3968
 - Cleanup, 3969
 - GetByName, 3970
 - GetProperty, 3972
 - Init, 3973
 - SetProperty, 3975
- IDLgrSurface
 - Cleanup, 4003
 - GetCTM, 4004
 - GetProperty, 4006
 - Init, 4007

methods (*continued*)

- IDLgrSurface
 - SetProperty, 4009
- IDLgrSymbol
 - Cleanup, 4016
 - GetProperty, 4017
 - Init, 4018
 - SetProperty, 4020
- IDLgrTessellator
 - AddPolygon, 4025
 - Cleanup, 4027
 - Init, 4028
 - Reset, 4029
 - Tessellate, 4030
- IDLgrText
 - Cleanup, 4049
 - GetCTM, 4050
 - GetProperty, 4052
 - Init, 4053
 - SetProperty, 4055
- IDLgrView
 - Add, 4065
 - Cleanup, 4066
 - GetByName, 4067
 - GetProperty, 4069
 - Init, 4070
 - SetProperty, 4072
- IDLgrViewgroup
 - Add, 4077
 - Cleanup, 4078
 - GetByName, 4079
 - GetProperty, 4081
 - Init, 4082
 - SetProperty, 4084
- IDLgrVolume
 - Cleanup, 4104
 - ComputeBounds, 4105
 - GetCTM, 4106
 - GetProperty, 4108
 - Init, 4109
 - PickVoxel, 4111

methods (*continued*)

- IDLgrVolume
 - SetProperty, 4113
- IDLgrVRML
 - Draw, 4124
 - GetDeviceInfo, 4125
 - GetFontnames, 4127
 - GetProperty, 4129
 - GetTextDimensions, 4130
 - Init, 4132
 - SetProperty, 4134
- IDLgrWindow
 - Cleanup, 4149
 - Draw, 4150
 - Erase, 4152
 - GetContiguousPixels, 4153
 - GetDeviceInfo, 4154
 - GetDimensions, 4156
 - GetFontnames, 4158
 - GetProperty, 4160
 - GetTextDimensions, 4161
 - Iconify, 4163
 - Init, 4164
 - PickData, 4166
 - Read, 4169
 - Select, 4170
 - SetCurrentCursor, 4173
 - SetCurrentZoom, 4175
 - SetProperty, 4176
 - Show, 4177
 - ZoomIn, 4178
 - ZoomOut, 4179
- IDLitCommand
 - AddItem, 3086
 - Cleanup, 3088
 - GetItem, 3089
 - GetProperty, 3090
 - GetSize, 3091
 - Init, 3092
 - SetProperty, 3094

methods (*continued*)

- IDLitCommandSet
 - Cleanup, 3098
 - GetSize, 3099
 - Init, 3100
- IDLitComponent
 - Cleanup, 3106
 - EditUserDefProperty, 3107
 - GetFullIdentifier, 3109
 - GetProperty, 3110
 - GetPropertyAttribute, 3111
 - GetPropertyByIdentifier, 3112
 - Init, 3113
 - QueryProperty, 3115
 - RegisterProperty, 3116
 - Restore, 3122
 - SetProperty, 3123
 - SetPropertyAttribute, 3124
 - SetPropertyByIdentifier, 3125
 - UpdateComponentVersion, 3126
- IDLitContainer
 - Add, 3130
 - AddByIdentifier, 3131
 - Cleanup, 3132
 - FindIdentifiers, 3133
 - Get, 3135
 - GetByIdentifier, 3137
 - Init, 3138
 - Remove, 3139
 - RemoveByIdentifier, 3140
- IDLitData
 - AddDataObserver, 3145
 - Cleanup, 3146
 - Copy, 3147
 - GetByType, 3148
 - GetData, 3149
 - GetProperty, 3150
 - GetSize, 3151
 - Init, 3152
 - NotifyDataChange, 3154
 - NotifyDataComplete, 3155

methods (*continued*)

IDLitData

- RemoveDataObserver, 3156
- SetData, 3157
- SetProperty, 3158

IDLitDataContainer

- Add, 3162
- Cleanup, 3163
- GetData, 3164
- GetIdentifiers, 3165
- GetProperty, 3166
- Init, 3167
- SetData, 3169
- SetProperty, 3171

IDLitDataOperation

- Cleanup, 3176
- DoExecuteUI, 3177
- Execute, 3179
- GetProperty, 3181
- Init, 3182
- SetProperty, 3184
- UndoExecute, 3185

IDLitIMessaging

- AddOnNotifyObserver, 3190
- DoOnNotify, 3192
- ErrorMessage, 3194
- GetTool, 3196
- ProbeStatusMessage, 3197
- ProgressBar, 3198
- PromptUserText, 3200
- PromptUserYesNo, 3201
- RemoveOnNotifyObserver, 3202
- SignalError, 3203
- StatusMessage, 3204

IDLitManipulator

- Cleanup, 3213
- CommitUndoValues, 3214
- GetCursorType, 3216
- GetProperty, 3218
- Init, 3219
- OnKeyboard, 3221

methods (*continued*)

IDLitManipulator

- OnLoseCurrentManipulator, 3223
- OnMouseDown, 3224
- OnMouseMove, 3226
- OnMouseUp, 3228
- RecordUndoValues, 3229
- RegisterCursor, 3231
- SetCurrentManipulator, 3233
- SetProperty, 3234

IDLitManipulatorContainer

- Add, 3238
- GetCurrent, 3239
- GetCurrentManipulator, 3240
- GetProperty, 3241
- Init, 3242
- OnKeyboard, 3244
- OnMouseDown, 3246
- OnMouseMove, 3248
- OnMouseUp, 3250
- SetCurrent, 3251
- SetCurrentManipulator, 3252
- SetProperty, 3253

IDLitManipulatorManager

- Add, 3257
- AddManipulatorObserver, 3258
- GetDefaultManipulator, 3259
- Init, 3260
- RemoveManipulatorObserver, 3262

IDLitManipulatorVisual

- Cleanup, 3266
- GetProperty, 3267
- Init, 3268
- SetProperty, 3270

IDLitOperation

- Cleanup, 3277
- DoAction, 3278
- GetProperty, 3280
- Init, 3281
- RecordFinalValues, 3283
- RecordInitialValues, 3285

methods (*continued*)

IDLitOperation

- RedoOperation, [3287](#)
- SetProperty, [3289](#)
- UndoOperation, [3290](#)

IDLitParameter

- Cleanup, [3295](#)
- GetParameter, [3296](#)
- GetParameterAttribute, [3297](#)
- GetParameterSet, [3299](#)
- Init, [3300](#)
- OnDataChangeUpdate, [3301](#)
- OnDataDisconnect, [3303](#)
- QueryParameter, [3305](#)
- RegisterParameter, [3307](#)
- SetData, [3309](#)
- SetParameterAttribute, [3311](#)
- SetParameterSet, [3313](#)

IDLitParameterSet

- Add, [3318](#)
- Cleanup, [3320](#)
- Copy, [3321](#)
- Get, [3322](#)
- GetByName, [3324](#)
- GetParameterName, [3326](#)
- Init, [3327](#)
- Remove, [3329](#)

IDLitReader

- Cleanup, [3333](#)
- GetData, [3334](#)
- GetFileExtensions, [3335](#)
- GetFilename, [3336](#)
- GetProperty, [3337](#)
- Init, [3338](#)
- IsA, [3340](#)
- SetFilename, [3341](#)
- SetProperty, [3342](#)

IDLitTool

- ActivateManipulator, [3350](#)
- Add, [3351](#)
- AddService, [3352](#)

methods (*continued*)

IDLitTool

- Cleanup, [3353](#)
- CommitActions, [3354](#)
- DisableUpdates, [3355](#)
- DoAction, [3356](#)
- DoSetProperty, [3357](#)
- DoUIService, [3359](#)
- EnableUpdates, [3360](#)
- FindIdentifiers, [3361](#)
- GetCurrentManipulator, [3364](#)
- GetFileReader, [3365](#)
- GetFileWriter, [3366](#)
- GetManipulators, [3367](#)
- GetOperations, [3368](#)
- GetProperty, [3369](#)
- GetSelectedItems, [3370](#)
- GetService, [3371](#)
- GetVisualization, [3372](#)
- Init, [3374](#)
- RefreshCurrentWindow, [3376](#)
- Register, [3377](#)
- RegisterCustomization, [3380](#)
- RegisterFileReader, [3381](#)
- RegisterFileWriter, [3383](#)
- RegisterManipulator, [3386](#)
- RegisterOperation, [3388](#)
- RegisterStatusBarSegment, [3392](#)
- RegisterVisualization, [3393](#)
- SetProperty, [3395](#)
- UnRegister, [3396](#)
- UnRegisterCustomization, [3397](#)
- UnRegisterFileReader, [3398](#)
- UnRegisterFileWriter, [3399](#)
- UnRegisterManipulator, [3400](#)
- UnRegisterOperation, [3401](#)
- UnRegisterStatusBarSegment, [3402](#)
- UnRegisterVisualization, [3403](#)

IDLitUI

- AddOnNotifyObserver, [3409](#)
- Cleanup, [3411](#)

methods (*continued*)

IDLitUI

- DoAction, [3412](#)
- GetProperty, [3413](#)
- GetTool, [3414](#)
- GetWidgetByName, [3415](#)
- Init, [3416](#)
- RegisterUIService, [3417](#)
- RegisterWidget, [3419](#)
- RemoveOnNotifyObserver, [3421](#)
- SetProperty, [3422](#)
- UnRegisterUIService, [3423](#)
- UnRegisterWidget, [3424](#)

IDLitVisualization

- Add, [3432](#)
- Aggregate, [3434](#)
- BeginManipulation, [3435](#)
- Cleanup, [3436](#)
- EndManipulation, [3437](#)
- Get, [3438](#)
- GetCenterRotation, [3440](#)
- GetCurrentSelectionVisual, [3442](#)
- GetDataSpace, [3443](#)
- GetDataString, [3444](#)
- GetDefaultSelectionVisual, [3445](#)
- GetManipulatorTarget, [3446](#)
- GetProperty, [3447](#)
- GetRequestedAxesStyle, [3448](#)
- GetSelectionVisual, [3449](#)
- GetTypes, [3450](#)
- GetXYZRange, [3451](#)
- Init, [3453](#)
- Is3D, [3454](#)
- IsIsotropic, [3455](#)
- IsManipulatorTarget, [3456](#)
- IsSelected, [3457](#)
- Move, [3458](#)
- On2DRotate, [3459](#)
- OnAxesRequestChange, [3460](#)
- OnAxesStyleRequestChange, [3461](#)
- OnDataChange, [3462](#)

methods (*continued*)

IDLitVisualization

- OnDataComplete, [3463](#)
- OnDataRangeChange, [3464](#)
- OnDimensionChange, [3465](#)
- OnWorldDimensionChange, [3466](#)
- Remove, [3467](#)
- RequestsAxes, [3468](#)
- Restore, [3469](#)
- Rotate, [3470](#)
- Scale, [3472](#)
- Select, [3474](#)
- Set3D, [3476](#)
- SetAxesRequest, [3477](#)
- SetAxesStyleRequest, [3479](#)
- SetCurrentSelectionVisual, [3481](#)
- SetData, [3482](#)
- SetDefaultSelectionVisual, [3483](#)
- SetParameterSet, [3484](#)
- SetProperty, [3485](#)
- UpdateSelectionVisual, [3486](#)
- VisToWindow, [3487](#)
- WindowToVis, [3489](#)

IDLitWindow

- Add, [3494](#)
- AddWindowEventObserver, [3495](#)
- Cleanup, [3496](#)
- ClearSelections, [3497](#)
- DoHitTest, [3498](#)
- GetEventMask, [3500](#)
- GetProperty, [3502](#)
- GetSelectedItems, [3503](#)
- Init, [3504](#)
- OnKeyboard, [3506](#)
- OnMouseDown, [3509](#)
- OnMouseMotion, [3511](#)
- OnMouseUp, [3513](#)
- OnScroll, [3515](#)
- Remove, [3516](#)
- RemoveWindowEventObserver, [3517](#)
- SetCurrentZoom, [3518](#)

methods (*continued*)

- IDLitWindow
 - SetEventMask, [3519](#)
 - SetManipulatorManager, [3521](#)
 - SetProperty, [3522](#)
 - ZoomIn, [3523](#)
 - ZoomOut, [3524](#)
- IDLitWriter
 - Cleanup, [3528](#)
 - GetFileExtensions, [3529](#)
 - GetFilename, [3530](#)
 - GetProperty, [3531](#)
 - Init, [3532](#)
 - IsA, [3534](#)
 - SetData, [3535](#)
 - SetFilename, [3536](#)
 - SetProperty, [3537](#)
- IDLjavaObject
 - GetProperty, [4226](#)
 - Init, [4227](#)
 - SetProperty, [4229](#)
- TrackBall
 - Init, [4234](#)
 - Reset, [4235](#)
 - Update, [4237](#)
- Microsoft Internet Explorer, [1438](#)
- Microsoft Windows
 - display device (WIN)
 - accepted keywords, [4318](#)
 - support for, [4244](#)
- Miller map projection, [1356](#)
- MIN function, [1430](#)
- MIN_CURVE_SURF function
 - reference, [1433](#)
 - smoothing with, [315](#)
- MINEXP machine-specific parameter, [1287](#)
- minimization
 - Davidon-Fletcher-Powell method, [541](#)
 - Powell method, [1592](#)
- minimum and maximum operators, [4394](#)
- minimum curvature surface, [1433](#)
- minimum value
 - for slider widgets (MINIMUM keyword), [2443](#)
 - of an array, [1430](#)
- MINOR system variable field, [4385](#)
- MIPSEB_DBLFIXUP, *see* obsolete routines
- missing data
 - in CONTOUR plots, [321](#)
 - in irregular grids
 - TRI_SURF, [2129](#)
 - TRIGRID, [2138](#)
 - in map projections, [1316](#)
 - in plots
 - OPlot, [1520](#)
 - PLOTS, [1544](#)
 - SHADE_SURF, [1878](#)
 - SURFACE, [2060](#)
 - in reconstructed images, [1777](#)
 - in rotated images, [1819](#)
 - in velocity fields, [2216](#)
 - in warped images, [1569](#)
- MK_HTML_HELP procedure, reference, [1438](#)
- model object, [3756](#)
- MODIFYCT procedure, [1441](#)
- modules
 - compiled, [865](#)
 - dynamically loaded, [860](#)
- modulo operator, [4393](#)
- Mollweide map projection, [1356](#)
- MOMENT function, [1443](#)
- MORPH_CLOSE function, [1446](#)
- MORPH_DISTANCE function, [1449](#)
- MORPH_GRADIENT function, [1452](#)
- MORPH_HITORMISS function, [1455](#)
- MORPH_OPEN function, [1458](#)
- MORPH_THIN function, [1461](#)
- MORPH_TOPHAT function, [1463](#)
- morphology
 - dilation operator, [569](#)
 - erosion operator, [618](#)

mouse

- double-clicks, [2422](#)
- reading position of, [1705](#)
- reading position with the CURSOR procedure, [371](#)
- returning events from draw widgets, [2358](#)

MOVIE, *see* obsolete routines

movies

- MPEG
 - closing, [1466](#)
 - opening, [1467](#)
 - saving, [1473](#)
 - storing image frame, [1471](#)

moving, files (FILE_MOVE procedure), [692](#)

moving averages

- SMOOTH function, [1960](#)
- TS_SMOOTH function, [2162](#)

MPEG object, [3779](#)

MPEG_CLOSE procedure, [1466](#)

MPEG_OPEN function, [1467](#)

MPEG_PUT procedure, [1471](#)

MPEG_SAVE procedure, [1473](#)

MPROVE, *see* obsolete routines

MrSID image files

- deleting, [2838](#)
- dimensions, [2839](#)
- extracting data, [2841](#)
- loading, [2835](#)
- query, [2835](#)
- query properties, [2844](#)

MSG_CAT_CLOSE, *see* obsolete routines

MSG_CAT_COMPILE, *see* obsolete routines

MSG_CAT_OPEN, *see* obsolete routines

Müller's method, [760](#)

MULTI procedure, [1474](#)

MULTI system variable field, [4380](#)

MULTICOMPARE, *see* obsolete routines

multiple correlation coefficient, [1283](#)

multiple plots on a page, [4380](#)

multiplication, * operator, [4393](#)

multiplication of matrices, [1365](#)

multivariate analysis

- contingency table, [368](#)
- Kruskal-Wallis H-test, [1124](#)
- multiple correlation, [1283](#)
- partial correlation, [1525](#)

multivariate functions

- CTI_TEST, [368](#)
- KW_TEST, [1124](#)
- M_CORRELATE, [1283](#)
- P_CORRELATE, [1525](#)

N

N_COLORS system variable field, [4377](#)

N_ELEMENTS function, reference, [1475](#)

N_PARAMS function, reference, [1477](#)

N_TAGS function, [1478](#)

NAME system variable field, [4377](#)

named, variables, [61](#), [61](#), [2667](#)

names, of structure tags, [2092](#)

NARROW keyword, [4265](#)

native format (floating-point values), [201](#)

natural exponential function, [631](#)

natural logarithm, [95](#)

NCAR binary encoding, [4265](#)

NCAR keyword, [4265](#)

NCAR Raster Interchange Format files, writing, [2519](#)

NE operator, about, [4399](#)

NearestColor method, IDLgrPalette, [3806](#)

negation operator, [4392](#)

NEGEP machine-specific parameter, [1286](#)

nesting

- setting TRACEBACK keyword, [866](#)
- showing for procedures and functions, [859](#)

Netscape, [1438](#)

new page, [610](#)

NewDocument method, IDLgrPrinter, [3924](#)

newline character, [2484](#)

NewPage method, IDLgrPrinter, [3925](#)

NEWTON function, [1481](#)

- Newton's method, [1000](#), [1481](#)
 - NGRD machine-specific parameter, [1286](#)
 - NOCLIP keyword, [4338](#)
 - NOCLIP system variable field, [4381](#)
 - NODATA keyword, [4338](#)
 - NOERASE keyword, [4339](#)
 - NOERASE system variable field, [4381](#)
 - noise, filtering, [1380](#)
 - nonlinear equations
 - BROYDEN, [194](#)
 - CONSTRAINED_MIN, [307](#)
 - FX_ROOT, [760](#)
 - FZ_ROOTS, [763](#)
 - NEWTON, [1481](#)
 - nonparametric tests
 - LNP_TEST, [1248](#)
 - MD_TEST, [1374](#)
 - R_TEST, [1682](#)
 - RS_TEST, [1828](#)
 - S_TEST, [1831](#)
 - XSQ_TEST, [2637](#)
 - NORM function, [1484](#)
 - normal
 - coordinates, converting to other types, [329](#)
 - distribution (Gaussian)
 - cutoff value, [769](#)
 - probability, [771](#)
 - random deviates, [1700](#)
 - NORMAL keyword, [4339](#)
 - normally-distributed random numbers, [1693](#)
 - NOT operator, [4397](#)
 - NR_BETA, *see* obsolete routines
 - NR_BROYDN, *see* obsolete routines
 - NR_CHOLDC, *see* obsolete routines
 - NR_CHOLSL, *see* obsolete routines
 - NR_DFPMIN, *see* obsolete routines
 - NR_ELMHES, *see* obsolete routines
 - NR_EXPINT, *see* obsolete routines
 - NR_FULSTR, *see* obsolete routines
 - NR_HQR, *see* obsolete routines
 - NR_INVERT, *see* obsolete routines
 - NR_LINBCG, *see* obsolete routines
 - NR_LUBKSB, *see* obsolete routines
 - NR_LUDCMP, *see* obsolete routines
 - NR_MACHAR, *see* obsolete routines
 - NR_MPROVE, *see* obsolete routines
 - NR_NEWT, *see* obsolete routines
 - NR_POWELL, *see* obsolete routines
 - NR_QROMB, *see* obsolete routines
 - NR_QROMO, *see* obsolete routines
 - NR_QSIMP, *see* obsolete routines
 - NR_RK4, *see* obsolete routines
 - NR_SPLINE, *see* obsolete routines
 - NR_SPLINT, *see* obsolete routines
 - NR_SPRSAB, *see* obsolete routines
 - NR_SPRSAX, *see* obsolete routines
 - NR_SPRSIN, *see* obsolete routines
 - NR_SVBKSB, *see* obsolete routines
 - NR_SVD, *see* obsolete routines
 - NR_TQLI, *see* obsolete routines
 - NR_TRED2, *see* obsolete routines
 - NR_TRIDAG, *see* obsolete routines
 - NR_WTN, *see* obsolete routines
 - NR_ZROOTS, *see* obsolete routines
 - NRIF, files, writing, [2519](#)
 - NSUM system variable field, [4382](#)
 - Null display device (NULL), [4298](#)
 - number of array elements, [1475](#)
 - numbers, random
 - normally distributed, [1693](#)
 - uniformly-distributed, [1698](#)
 - numeric keypads, [1872](#)
 - numerical integration, [1648](#)
- ## O
- OBJ_CLASS function, reference, [1487](#)
 - OBJ_DESTROY procedure, [1489](#)
 - OBJ_ISA function, reference, [1490](#)
 - OBJ_NEW function, reference, [1491](#)
 - OBJ_VALID function, reference, [1493](#)
 - OBJARR function, reference, [1495](#)

- object classes, undocumented, [1488](#), [2677](#)
- object graphics, CMYK color model, [3609](#)
- objects
 - creating, [1491](#)
 - creating arrays, [1495](#)
 - destroying, OBJ_DESTROY function, [1489](#)
 - determining
 - class names, [1487](#)
 - subclasses, [1490](#)
- iTools
 - command collection, [3095](#)
 - component (class) base, [3101](#)
 - component collection, [3127](#)
 - data collection, [3159](#)
 - data undo and redo, [3172](#)
 - manipulating, [3205](#)
 - manipulator base, [3254](#)
 - manipulator collection, [3235](#)
 - messaging, [3187](#)
 - naming data, [3315](#)
 - operating tasks, [3271](#)
 - parameters, [3292](#)
 - reading files, [3330](#)
 - storing data, [3141](#)
 - tool base, [3343](#), [3404](#)
 - undo and redo commands, [3083](#)
 - user-interface, [3406](#)
 - visual base, [3425](#)
 - window base, [3491](#)
 - writing files, [3525](#)
- Java classes, IDLjavaObject, [4223](#)
- Object Graphics, font use, [4415](#)
- testing existence, [1493](#)
- OBLIQUE keyword, [4266](#)
- obsolete routines and system variables, [4456](#)
- obsolete SDF routines, [4463](#)
- OMARGIN system variable field, [4385](#)
- ON_ERROR procedure
 - messages, [1424](#)
 - reference, [1497](#)
- ON_IOERROR procedure, [1498](#)
 - messages, [1424](#)
- online help
 - calling from programs, [1500](#)
 - viewing from own program
 - DOC_LIBRARY, [588](#)
 - MK_HTML_HELP, [1438](#)
- ONLINE_HELP procedure, reference, [1500](#)
- ONLINE_HELP_PDF_INDEX procedure, [1506](#)
- ONLY_8BIT, *see* obsolete routines
- opacities, [2225](#)
- OPEN procedures, [1511](#)
- opening, Shapefiles, [2880](#)
- opening files
 - getting information on open files, [859](#)
 - OPEN procedures, [1511](#)
- opening operation, in image processing, [570](#)
- operating system, current version in use, [4372](#)
- operators
 - &&, [4396](#)
 - ||, [4396](#)
 - ~, [4396](#)
 - addition, [4392](#)
 - AND, [4397](#)
 - array concatenation, [4400](#)
 - assignment, [4400](#)
 - Bitwise, [4397](#)
 - compound assignment, [4401](#)
 - decrement, [4392](#)
 - division, [4393](#)
 - EQ, [4399](#)
 - exponentiation, [4393](#)
 - GE, [4399](#)
 - GT, [4399](#)
 - increment, [4392](#)
 - LE, [4399](#)
 - Logical, [4396](#)
 - LT, [4399](#)
 - mathematical, table of, [4392](#)
 - matrix multiplication, [4395](#)

- operators (*continued*)
- maximum, [4394](#)
 - minimum, [4394](#)
 - minimum and maximum, [4394](#)
 - modulo, [4393](#)
 - multiplication, [4393](#)
 - NE, [4399](#)
 - NOT, [4397](#)
 - OR, [4397](#)
 - other, [4400](#)
 - relational, [4399](#)
 - subtraction and negation, [4392](#)
 - XOR, [4398](#)
- OPLOT procedure, [1520](#)
- OPLOTERR procedure, [1523](#)
- optimal feasible vector, [1917](#)
- optimization
- AMOEBAS function, [99](#)
 - CONSTRAINED_MIN, [307](#)
 - DFPMIN, [541](#)
 - POWELL, [1592](#)
- OPTIMIZE keyword, [4266](#)
- optional parameters in user-written functions, [1477](#)
- OR operator, [4397](#)
- ORDERED keyword, [4267](#)
- ordinary differential equations, LSODE function, [1268](#)
- ordinary differential equations, RK4, [1812](#)
- ORIENTATION keyword, [4339](#)
- ORIGIN system variable field, [4377](#)
- orthographic map projection, [1356](#)
- outer margins, setting, [4385](#)
- outline fonts, [4414](#)
- outlines of continents, [1302](#)
- outlying data regression, [1214](#)
- OUTPUT, *see* obsolete routines
- output
- BMP files, [2507](#)
 - GIF files, [2510](#)
 - JPEG files, [2514](#)
- output (*continued*)
- NRIF files, [2519](#)
 - PGM files, [2526](#)
 - PICT files, [2521](#)
 - PPM files, [2526](#)
 - SRF files, [2530](#)
 - TIFF files, [2534](#)
 - wave files, [2546](#)
- OUTPUT keyword, [4267](#)
- overflow, integer, [1287](#)
- overplotting, [1520](#)
- ## P
- P_CORRELATE function, [1525](#)
- page break, [610](#)
- PALATINO keyword, [4267](#)
- palette object, [3795](#)
- PALETTE, *see* obsolete routines
- pan offset, [4377](#)
- parallels, drawing
- MAP_GRID procedure, [1307](#)
 - MAP_SET procedure, [1359](#)
- parameters
- finding number of, [1477](#)
 - formal, [61](#)
- parents, of widgets, [2392](#)
- partial correlation coefficient, [1525](#)
- PARTIAL_COR, *see* obsolete routines
- PARTIAL2_COR, *see* obsolete routines
- PARTICLE_TRACE procedure, [1527](#)
- path
- caching, [1530](#), [1530](#)
 - definition string, [635](#)
 - maintaining in memory, [1530](#)
- path separation delimiters, [1537](#)
- path specification, [700](#)
- PATH_CACHE procedure, [1530](#)
- PATH_SEP function, [1537](#)
- pattern object, [3809](#)

- PCL
 - driver, [4299](#)
 - files, [4289](#)
- PCOMP function, [1538](#)
- Pearson correlation coefficient, [341](#)
- period (character), [4407](#)
- permutation, [654](#)
- perspective, [2089](#)
- PGM files
 - reading, [1741](#)
 - writing, [2526](#)
- phase, [137](#)
- PHASER, *see* obsolete routines
- PickData method
 - IDLgrBuffer, [3592](#)
 - IDLgrWindow, [4166](#)
- PICKFILE, *see* obsolete routines
- PickRegion method, IDLgrROIgroup, [3960](#)
- PickVertex method, IDLgrROI, [3943](#)
- PickVoxel method, IDLgrVolume, [4111](#)
- PICT files
 - reading, [1736](#)
 - writing, [2521](#)
- pixels
 - returning value of, using RDPIX procedure, [1705](#)
- PIXELS keyword, [4267](#)
- plane of vector-drawn text, [2656](#)
- plot object, [3819](#)
- PLOT procedure, [1543](#)
- PLOT_3DBOX procedure, [1547](#)
- PLOT_FIELD procedure, [1551](#)
- PLOT_IO, *see* YLOG keyword to PLOT
- PLOT_OI, *see* XLOG keyword to PLOT
- PLOT_OO, *see* (XY)LOG keywords to PLOT
- PLOT_TO keyword, [4267](#)
- PLOTERR procedure, [1553](#)
- plots
 - interactive (iTool) routine, [1021](#)
 - margins, [4385](#)
 - outer margins, [4385](#)
 - plots (*continued*)
 - viewing in 3D, [2612](#)
- PLOTS procedure, [1555](#)
- PLOTTER_ON_OFF keyword, [4268](#)
- plotting
 - 2D fields, [1551](#)
 - 3D fields, [739](#)
 - 3D transformations, [4342](#)
 - adjusting, [458](#)
 - coordinate conversion, [336](#)
 - scaling, [1844](#)
 - setting, [2064](#)
 - specifying, [1842](#)
 - vertices, [2218](#)
- axes
 - graphics keywords, [4350](#)
 - thickness, [4344](#)
- bar plots, [145](#)
- closing files (CLOSE_FILE keyword), [4253](#)
- color, [2611](#)
- contour plots, [958](#)
 - creating, [315](#)
- drawing axes (AXIS procedure), [141](#)
- error bars
 - ERRPLOT procedure, [624](#)
 - OPLOTERR procedure, [1523](#)
 - PLOTERR procedure, [1553](#)
- filename for output (FILENAME keyword), [4260](#)
- flow field, [739](#)
- functions of 2 variables, [1547](#)
- height of output, [4284](#)
- histogram, [4341](#)
- landscape orientation, [4264](#)
- line thickness
 - graphic keyword, [4342](#)
 - system variable, [4383](#)
- lines, [1555](#)
- linestyles
 - graphic keyword, [4337](#)
 - system variable, [4380](#)

- plotting (*continued*)
 - logarithmic axes
 - linear-log, 1545
 - log-linear, 325
 - AXIS, 142
 - CONTOUR, 325
 - PLOT, 1545
 - SHADE_SURF, 1879
 - SURFACE, 2062
 - missing data, 1520, 1544
 - multiple plots on a page
 - placing, 4308
 - specifying, 4380
 - output, positioning, 4291
 - overplotting, 958
 - vector data and plot, 1520
 - PLOT procedure, 1543
 - points, 1555
 - polar
 - oPLOT procedure, 1521
 - PLOT procedure, 1545
 - portrait orientation, 4269
 - position of window
 - graphic keyword, 4339
 - system variable, 4382
 - region, 4382
 - selecting a plotting device, 1865
 - shaded surfaces, 1876
 - subtitles
 - graphic keyword, 4341
 - system variable, 4382
 - symbol size, 4341
 - symbols
 - graphic keyword, 4340
 - system variable, 4382
 - text, 2655
 - three-dimensional lines, 1556
 - titles
 - graphic keyword, 4343
 - system variable, 4383
 - user-defined symbols, 2205
- plotting (*continued*)
 - velocity field, 739
 - velocity fields, 2215
 - weather fronts, 2239
 - width of output, 4283
 - wire-mesh surfaces, 2058
 - without data, 4338
 - without erasing
 - graphic keyword, 4339
 - system variable, 4381
 - XY plots, 1543
 - Z-coordinate for, graphic keyword, 4351
- PM, *see* obsolete routines
- PMF, *see* obsolete routines
- PNG library, supported version, 1738
- PNT_LINE function, 1558
- POINT_LUN procedure, 1560
- pointers
 - creating, 1623
 - creating arrays, 1627
 - destroying, 1622
 - freeing, 854
 - testing existence, 1625
- Poisson random deviates
 - RANDOMN function, 1695
 - RANDOMU function, 1701
- polar plots, 1545
 - contours, 1562
 - coordinates
 - converting, 379
 - interpolation of surface, 1564
- POLAR_CONTOUR procedure, 1562
- POLAR_SURFACE function, 1564
- polishing of roots, 763
- political boundaries, 1302
- POLY function, 1566
- POLY_2D function, 1567
- POLY_AREA function, 1571
- POLY_FIT function, 1573
- POLYCONTOUR, *see* obsolete routines
- POLYFILL keyword, 4268

- POLYFILL procedure, [1577](#)
- POLYFILLV function, [1582](#)
- POLYFITW, *see* obsolete routines
- polygon filling
 - POLYFILL procedure, [1577](#)
 - returning array subscripts, [1582](#)
 - with HP plotters, [4268](#)
- polygon object
 - clipping meshes, [1388](#)
 - decimating meshes, [1395](#)
 - IDLgrPolygon, [3844](#)
 - merging meshes, [1401](#)
 - smoothing meshes, [1414](#)
- polyline object, [3875](#)
- polynomial warping, [1567](#)
- polynomials
 - digital smoothing, [1837](#)
 - Laguerre, [1217](#)
 - least-squares fit, [1837](#)
 - Legendre, [1221](#)
- POLYSHADE function, [1584](#)
- POLYWARP procedure, [1588](#)
- POPD procedure, [1591](#)
- pop-up menu *see* context-sensitive menu
- PORTRAIT keyword, [4269](#)
- portrait orientation
 - for IDL output (PORTRAIT keyword), [4269](#)
 - PostScript positioning, [4305](#)
- POSITION keyword, [4339](#)
- POSITION system variable field, [4382](#)
- positional parameters, [61](#)
 - returning number of, [1477](#)
- positioning
 - child widgets within a base, [2254](#)
 - commands, [4435](#)
 - cursor, [2169](#)
 - graphics cursor, [371](#)
 - PostScript output, [4305](#)
 - top level base widgets, [2343](#)
 - widget bases, [2254](#)
 - windows (XPOS and YPOS keywords), [2505](#)
- PostScript
 - CMYK, [4254](#)
 - color, [4303](#)
 - device, [4302](#)
 - encapsulated
 - about EPS, [4307](#)
 - device keyword, [4258](#)
 - EPSI (Encapsulated PostScript Interchange)
 - files, [4270](#)
 - files, [4289](#)
 - files with preview headers, [4270](#)
 - font index, [4260](#)
 - fonts
 - displaying, [1618](#)
 - using, [4303](#)
 - importing graphics into other programs, [4310](#)
 - importing into another document, [4259](#)
 - language level, [4265](#)
 - multiple plots on a single page, [4308](#)
 - pixel bit depth, [4252](#)
 - positioning output, [4305](#)
 - scaling entire plot (SCALE_FACTOR keyword), [4272](#)
 - TrueColor images, [4304](#)
 - writing 24-bit images, [4305](#)
- Powell minimization (POWELL procedure), [1592](#)
- PPM files
 - reading, [1741](#)
 - writing, [2526](#)
- PREVIEW keyword, [4270](#)
- PRIMES function, [1595](#)
- principal components analysis, [1538](#)
- PRINT procedure, [1596](#)
- PRINT_FILE keyword, [4270](#)
- PRINTD procedure, [1599](#)
- Printer Control Language, *see* PCL
- PRINTER device, [4301](#)
- printer object, [3902](#)
- PRINTF procedure, [1596](#)

- printing
 - closing files (CLOSE_FILE keyword), 4253
 - dialog, 559
 - filename for output (FILENAME keyword), 4260
 - graphics output files, 4289
 - landscape orientation, 4264
 - printer device, 4301
 - printer set up, 4290
 - properties, 557
 - setup dialog, 557
 - to file units, 1596
 - to standard output, 1596
- PRO statement, 1600
- probability
 - bivariate distributions, 871
 - density distribution, 874
 - Gaussian distribution, 782
 - Histogram function, 874
- probability functions
 - binomial distribution, 180
 - Chi-square distribution
 - cutoff value, 251
 - probability, 253
 - F distribution
 - cutoff value, 650
 - probability, 652
 - Gaussian distribution
 - cutoff value, 769
 - probability, 771
 - student's T distribution
 - cutoff value, 2085
 - probability, 2087
- procedure methods, calling sequence for, 2666
- procedures
 - call stack, returning, 860
 - calling, sequence for, 60
 - compiled, 1825
 - DEVICE, 4244
 - displaying compiled, 865
 - SET_PLOT, 4244
 - PRODUCT function, 1602
 - PROFILE function, 1606
 - PROFILER procedure, 1609
 - PROFILES procedure, 1612
 - program, listings, 78
 - programming
 - displaying traceback information, 866
 - stopping programs, 2012
 - suspending execution of programs, 2231
 - traceback information, 860
 - PROJECT_VOL function, 1614
 - projections
 - 2D from 3D datasets, 1614
 - 3D plots on walls, 2613
 - Aitoff, 1355
 - Albers, 1355
 - azimuthal equidistant, 1355
 - cylindrical equidistant, 1355
 - gnomonic (central, gnomonic), 1355
 - Hammer-Aitoff, 1355
 - Lambert's conformal conic, 1355
 - Lambert's equal area, 1356
 - Mercator, 1356
 - Miller, 1356
 - Mollweide, 1356
 - orthographic, 1356
 - satellite, 1356
 - sinusoidal, 1357
 - stereographic, 1357
 - Transverse Mercator (UTM), 1357
 - prompt
 - changing default, 4372
 - reading from, 1707
 - PROMPT, *see* obsolete routines
 - properties
 - displaying, 2424
 - IDLanROI, 2682
 - IDLanROIGroup, 2710
 - IDLcomIDispatch, 4219
 - IDLffDICOM, 2734
 - IDLffJPEG2000, 2795

properties (*continued*)

- IDLffLangCat, 2823
- IDLffMrSID, 2837
- IDLffShape, 2861
- IDLffXMLDOMAttr, 2890
- IDLffXMLDOMCDATASection, 2898
- IDLffXMLDOMCharacterData, 2903
- IDLffXMLDOMComment, 2915
- IDLffXMLDOMDocument, 2920
- IDLffXMLDOMDocumentFragment, 2942
- IDLffXMLDOMDocumentType, 2946
- IDLffXMLDOMEElement, 2954
- IDLffXMLDOMEntity, 2966
- IDLffXMLDOMEntityReference, 2973
- IDLffXMLDOMNamedNodeMap, 2978
- IDLffXMLDOMNode, 2988
- IDLffXMLDOMNodeList, 3015
- IDLffXMLDOMNotation, 3021
- IDLffXMLDOMProcessingInstruction, 3027
- IDLffXMLDOMText, 3034
- IDLffXMLSAX, 3042
- IDLgrAxis, 3542
- IDLgrBuffer, 3574
- IDLgrClipboard, 3602
- IDLgrColobar, 3630
- IDLgrContour, 3650
- IDLgrFont, 3688
- IDLgrImage, 3698
- IDLgrLegend, 3722
- IDLgrLight, 3742
- IDLgrModel, 3758
- IDLgrMPEG, 3781
- IDLgrPalette, 3797
- IDLgrPattern, 3811
- IDLgrPlot, 3821
- IDLgrPolygon, 3846
- IDLgrPolyline, 3877
- IDLgrPrinter, 3904
- IDLgrROI, 3929
- IDLgrROIGroup, 3948

properties (*continued*)

- IDLgrScene, 3965
- IDLgrSurface, 3978
- IDLgrSymbol, 4012
- IDLgrText, 4034
- IDLgrView, 4058
- IDLgrViewgroup, 4075
- IDLgrVolume, 4087
- IDLgrVRML, 4117
- IDLgrWindow, 4138
- IDLitCommand, 3085
- IDLitCommandSet, 3097
- IDLitComponent, 3103
- IDLitContainer, 3129
- IDLitData, 3143
- IDLitDataContainer, 3161
- IDLitDataOperation, 3175
- IDLitIMessaging, 3189
- IDLitManipulator, 3208
- IDLitManipulatorContainer, 3237
- IDLitManipulatorManager, 3256
- IDLitManipulatorVisual, 3265
- IDLitOperation, 3275
- IDLitParameter, 3294
- IDLitParameterSet, 3317
- IDLitReader, 3332
- IDLitTool, 3347
- IDLitUI, 3408
- IDLitVisualization, 3429
- IDLitWindow, 3493
- IDLitWriter, 3527
- IDLjavaObject, 4225
- retrieving, 2670
- setting, 2670
- TrackBall, 4232
- widget, 2424
- properties of objects, 2669
- property sheet widgets, read-only, 2318, 2426
- PS_SHOW_FONTS procedure, 1618
- PSAFM procedure, 1619
- PSEUDO procedure, 1620

PSEUDO_COLOR keyword, [4271](#)
 pseudo-color images, converting from True-Color, [277](#)
 pseudo-color PostScript images, [4304](#)
 PSYM keyword, [4340](#)
 PSYM system variable field, [4382](#)
 PTR_FREE procedure, [1622](#)
 PTR_NEW function, [1623](#)
 PTR_VALID function, [1625](#)
 PTRARR function, [1627](#)
 pulldown menu
 activating from button, [2286](#)
 creating, [472](#)
 PUSH procedure, [1629](#)
 Put method, IDLgrMPEG, [3792](#)
 PWIDGET, *see* obsolete routines

Q

QGRID3 function, [1630](#)
 QHULL procedure, [1635](#)
 QL algorithm, [2146](#)
 QL method (computing eigenvalues), [596](#)
 QROMB function, [1640](#)
 QROMO function, [1645](#)
 QSIMP function, [1648](#)
 quantizing colors, [277](#)
 QUERY_* routines, [1651](#)
 QUERY_BMP routine, [1655](#)
 QUERY_DICOM function, [1656](#)
 QUERY_GIF routine, [1658](#)
 QUERY_IMAGE function, [1660](#)
 QUERY_JPEG routine, [1664](#)
 QUERY_MRSID function, [1667](#)
 QUERY_PICT routine, [1670](#)
 QUERY_PNG routine, [1671](#)
 QUERY_PPM routine, [1673](#)
 QUERY_SRF routine, [1675](#)
 QUERY_TIFF routine, [1676](#)
 QUERY_WAV function, [1678](#)
 question mark, starting online help, [4409](#)

quintic interpolation, [2139](#)
 quitting IDL, [629](#)
 quotation marks, octal numbers, [4407](#)

R

R_CORRELATE function, [1679](#)
 R_TEST function, [1682](#)
 radix, [1286](#)
 Radon backprojection, [1684](#)
 RADON function, [1684](#)
 Radon transform, [1684](#)
 random deviates
 binomial
 RANDOMN function, [1694](#)
 RANDOMU function, [1700](#)
 exponential
 RANDOMN function, [1695](#)
 RANDOMU function, [1700](#)
 gamma
 RANDOMN function, [1695](#)
 RANDOMU function, [1700](#)
 normal, [1700](#)
 Poisson
 RANDOMN function, [1695](#)
 RANDOMU function, [1701](#)
 random, [1701](#)
 random numbers
 normally-distributed, [1693](#)
 uniformly-distributed, [1698](#)
 RANDOMN function, [1693](#)
 RANDOMU function, [1698](#)
 RANGE system variable field, [4385](#)
 rank correlation coefficient, [1679](#)
 RANKS function, [1703](#)
 rank-sum test, [1828](#)
 RDPIX procedure, [1705](#)
 READ procedure, [1706](#)
 READ_ASCII function, [1709](#)
 READ_BINARY function, [1712](#)
 READ_BMP function, [1714](#)

- READ_DICOM function, 1717
- READ_GIF procedure, 1719
- READ_IMAGE function, 1721
- READ_INTERFILE procedure, 1723
- READ_JPEG procedure, 1725
- READ_JPEG2000 function, 1729
- READ_MRSID function, 1733
- READ_PICT procedure, 1736
- READ_PNG routine, 1738
- READ_PPM procedure, 1741
- READ_SPR function, 1743
- READ_SRF procedure, 1744
- READ_SYLK function, 1746
- READ_TIFF function, 1750
- READ_WAV function, 1759
- READ_WAVE procedure, 1760
- READ_X11_BITMAP procedure, 1762
- READ_XWD function, 1764
- READF procedure, 1706
- reading
 - ASCII files, 1709
 - BMP files, 1714
 - current color table, 2172
 - cursor position, 1705
 - data from a string, 1766
 - formatted data, 1706
 - from a prompt, 1707
 - GIF files, 1719
 - images from the display, 2174
 - Interfile files, 1723
 - JPEG files, 1725
 - mouse position, 371
 - PGM files, 1741
 - PICT files, 1736
 - pixel values, 1705
 - PPM files, 1741
 - SRF files, 1744
 - TIFF files, 1750
 - unformatted binary data, 1768
 - wave files, 1760
 - X11 bitmaps, 1762
 - XWD files, 1764
- read-only system variables, 521
- READS procedure, reference, 1766
- READU procedure, 1768
- real part of complex numbers, 735
- REAL_PART function, 1770
- realizing widgets, 2327
- REBIN function, 1771
- recall buffer, command, 1774
- RECALL_COMMANDS function, 1774
- RECON3 function, 1775
- reconstructions, 3D from 2D images, 1775
- recording an interactive IDL session, 1110
- rectangular coordinates
 - converting, 379
 - interpolation of, 1564
- recursive file searching, 701
- reduce operator, 618
- REDUCE_COLORS procedure, 1781
- REFORM function, 1783
- reformatting arrays, 1783
- region growing, properties dialog, 2631
- region labeling, 1211
- region of interest
 - defining, 519
 - IDLanROI, 2680
 - IDLanROIGroup, 2708
 - widgets, 416
 - XROI, 2621
- REGION system variable field
 - axes, 4386
 - plots, 4382
- REGION_GROW function, 1785
- Regis device, 4315
- REGISTER_CURSOR procedure, 1788
- registering iTools, 1079
- RegisterProperty method, 2673
- REGRESS function, 1790
- REGRESS1, *see* obsolete routines
- regression analysis, 1790
- REGRESSION, *see* obsolete routines
- regular expressions, 2020
- relational operators, 4399

- relaxed structure assignment
 - creating, 2053
 - restoring, 1805
- release, current version in use, 4372
- removing, breakpoints, 192
- rendering
 - 3D objects, 1406
 - 3D volumes as 2D images, 1614
 - voxel, 2225
- REPEAT...UNTIL statement, 1794
- REPLICATE function, 1795
- REPLICATE_INPLACE procedure, 1797
- reserved words, 4411
- Reset method
 - IDLgrModel, 3774
 - IDLgrTessellator, 4029
- RESET_STRING keyword, 4271
- resetting iTools, 1083
- resetting widgets, 2329
- resizing arrays
 - arbitrary amount, 302
 - by dimension multiples, 1771
 - two-dimensional, 633
- RESOLUTION keyword, 4271
- RESOLVE_ALL procedure, reference, 1799
- RESOLVE_ROUTINE procedure, 1802
- resolving iTool components, 1085
- resource names for IDL widgets, 2266
- RESTORE procedure, reference, 1804
- restoring IDL save files, 1804, 4193
- RETAIN keyword, 4271
- RETALL command, 1807
- retrieving
 - attributes of a Shapefile, 2872
 - image dimensions, 2839
- retrieving object properties, 2670
- RETURN command, 1808
- returning
 - subscripts of non-zero array elements, 2242
 - widget information, 2383
- REVERSE function, 1810
- reverse index list (for histograms), 874
- reversing
 - array indices, 1810
 - byte order, 2078
- REWIND, *see* obsolete routines
- RGB color system
 - converting, 275
 - displaying, 2171
 - widget slider, 481
- RGB_TO_HSV, *see* obsolete routines
- rhumb line, 1298
- RIEMANN, *see* obsolete routines
- rivers, 1302
- RK4 function, 1812
- RM, *see* obsolete routines
- RMF, *see* obsolete routines
- Roberts edge enhancement, 1815
- ROBERTS function, 1815
- ROI
 - deleting, 2634
 - geometric and statistical data, 2621
 - growing, 2630
 - histogram view, 2629
 - using XROI procedure, 2621
- Romberg integration
 - closed interval, 1640
 - open interval, 1645
- roots, 760, 763
- ROT function, 1817
- ROT_INT, *see* obsolete routines
- ROTATE function, 1820
- Rotate method, IDLgrModel, 3775
- rotating
 - arrays, 1820
 - by arbitrary amounts, 1817
 - by multiples of 90 degrees, 1820
 - the viewing matrix, 2089
 - using widgets, 396
- ROUND function, 1823
- rounding
 - ceiling function, 242

rounding (*continued*)
 determining, 1286
 floor function, 737
 to nearest integer, 1823
 ROUTINE_INFO function, 1825
 routines
 converting array subscripts, 123
 files
 base name, 661
 directory name, 676
 iTools (interactive)
 contours, 898
 creating tools, 933
 current (active) tool, 1073
 deleting tools, 1075
 image, 942, 965
 plot, 1021
 property sheet, 2424
 registering tools, 1079
 resetting tools, 1083
 retrieving current tool, 1077
 surface, 1052
 volume, 1087
 logical
 AND, 1254
 OR, 1256
 TRUE, 1258
 obsolete, 4456
 path caching, 1530
 saving as binary files, 1833
 validating variable names, 926
 row bases, 2268
 RS_TEST function, 1828
 RSI_GAMMAI, *see* obsolete routines
 RSTRPOS, *see* obsolete routines
 Runge-Kutta method, 1812
 run-length encoding, 1583
 runs test for randomness, 1682
 RUNS_TEST, *see* obsolete routines

S

S system variable field, 4386
 S_TEST function, 1831
 satellite map projection, 1356
 SAVE file, querying, 4193
 Save method, IDLgrMPEG, 3793
 SAVE procedure, reference, 1833
 save/restore
 binary files, 1834
 files, 1804
 object, 4193
 saved commands, displaying, 865
 savefile object, 4193
 SAVGOL function, 1837
 saving
 IDL routines as binary files, 1833
 IDL variables, 1833
 system variables, 1835
 variables, 1835
 Savitzky-Golay smoothing filter, 1837
 scalable pixels, 4293
 Scale method, IDLgrModel, 3776
 SCALE_FACTOR keyword, 4272
 SCALE3 procedure, 1842
 SCALE3D procedure, 1844
 scaling
 3D transformation, 2089
 factors, 4386
 values into range of bytes, 206
 scene object, 3963
 SCHOOLBOOK keyword, 4273
 SCOPE_LEVEL function, 1845
 SCOPE_VARFETCH function, 1846
 SCOPE_VARNAME function, 1852
 scroll bars
 for draw widgets, 2353
 APP_SCROLL keyword, 2354
 for text widgets, 2480
 scroll offset, 4377
 SEARCH2D function, 1858
 SEARCH3D function, 1861

- searching subdirectories, [701](#)
- searching, within strings, [2041](#)
- segmentation, [1211](#)
- Select method
 - IDLgrBuffer, [3596](#)
 - IDLgrWindow, [4170](#)
- semicolon character, [4407](#)
- semi-logarithmic plots
 - AXIS, [142](#)
 - CONTOUR, [325](#)
 - PLOT, [1545](#)
 - SHADE_SURF, [1879](#)
 - SURFACE, [2062](#)
- sensitizing widgets, WIDGET_CONTROL, [2331](#)
- SET_CHARACTER_SIZE keyword, [4273](#)
- SET_COLORMAP keyword, [4273](#)
- SET_FONT keyword, [4275](#)
- SET_GRAPHICS_FUNCTION keyword, [4277](#)
- SET_NATIVE_PLOT, *see* obsolete routines
- SET_PLOT procedure
 - device settings, [4244](#)
 - reference, [1865](#)
- SET_RESOLUTION keyword, [4278](#)
- SET_SCREEN, *see* obsolete routines
- SET_SHADING procedure, [1867](#)
- SET_STRING keyword, [4278](#)
- SET_SYMBOL, *see* obsolete routines
- SET_TRANSLATION keyword, [4279](#)
- SET_VIEWPORT, *see* obsolete routines
- SET_WRITE_MASK keyword, [4279](#)
- SET_XY, *see* obsolete routines
- SetCurrentCursor method, IDLgrWindow, [4173](#)
- SetCurrentZoom method, IDLgrWindow, [4175](#)
- SETENV procedure, reference, [1869](#)
- setjmp, C language, [236](#)
- SETLOG, *see* obsolete routines
- SetRGB method, IDLgrPalette, [3807](#)
- setting
 - breakpoints, [193](#)
 - keywords, [2667](#)
 - the current window, [2550](#)
 - widget values, [2338](#)
- setting properties
 - existing objects, [2670](#)
 - initialization, [2670](#)
 - objects, [2669](#)
- SETUP_KEYS procedure
 - reference, [1870](#)
 - using, [503](#)
- SFIT function, [1873](#)
- SHADE_SURF procedure, [1876](#)
- SHADE_SURF_IRR procedure, [1881](#)
- SHADE_VOLUME procedure, [1884](#)
- shaded surfaces
 - changing position of light source, [1867](#)
 - creating, [1876](#)
 - from polygons, [1584](#)
- shading
 - changing position of light source, [1867](#)
 - volumes, [1584](#)
- Shapefile
 - adding attributes, [2865](#)
 - attribute structure, [2856](#)
 - attributes, [2855](#)
 - closing, [2869](#)
 - entity, [2851](#)
 - entity structure, [2852](#)
 - included files, [2851](#)
 - inserting entities, [2882](#)
 - naming conventions, [2851](#)
 - object properties, [2876](#)
 - opening, [2880](#)
 - retrieving attributes, [2872](#)
 - retrieving entities, [2874](#)
 - setting attributes, [2884](#)
- sharable library, building, [1292](#)
- shared colormap
 - device keyword, [4279](#)

- shared colormap (*continued*)
 - translation vector, [4281](#)
- shared memory
 - debugging, [1889](#)
 - mapping, [1891](#)
 - unmapping, [1906](#)
- sheet feeder, [4258](#)
- shells, spawning, [1972](#)
- SHIFT function, [1887](#)
- shifting
 - array elements, [1887](#)
 - bit, [1041](#)
- SHMDEBUG function, [1889](#)
- SHMMAP procedure, [1891](#)
- SHMUNMAP procedure, [1906](#)
- SHMVAR function, [1908](#)
- short word swap, [203](#)
- shortcut menu *see* context-sensitive menu
- Show method, IDLgrWindow, [4177](#)
- SHOW3 procedure, [1912](#)
- SHOWFONT procedure, [1914](#)
- showing
 - images, [2165](#)
 - windows, [2552](#)
- shrink operator, [618](#)
- shrinking
 - arrays, [1771](#)
 - windows, [2552](#)
- .sid image files, [2835](#)
- SIGMA, *see* obsolete routines
- sign test, [1831](#)
- SIGN_TEST, *see* obsolete routines
- signal
 - filtering, [186](#)
 - processing, [331](#)
- SIMPLEX function, [1916](#)
- simplex method, [1916](#)
- SIMPSON, *see* obsolete routines
- Simpson's rule, [1648](#)
- SIN function, [1921](#)
- SINDGEN function, [1923](#)
- sine
 - hyperbolic, [1924](#)
 - inverse, [132](#)
 - SIN function, [1921](#)
- single-precision
 - arrays
 - FINDGEN function, [725](#)
 - FLTARR function, [741](#)
 - converting values to, [735](#)
- singular value decomposition
 - computing, [2065](#)
 - LA_SVD procedure, [1187](#)
 - using, [2075](#)
- SINH function, [1924](#)
- sinusoidal map projection, [1357](#)
- size, of arrays, [1926](#)
- SIZE function, [1926](#)
- skeletons of bi-level images, [2106](#)
- skewness
 - computing with MOMENT function, [1443](#)
 - computing with SKEWNESS function, [1931](#)
- SKEWNESS function, [1931](#)
- SKIP_LUN procedure, [1933](#)
- SKIPF, *see* obsolete routines
- slash character, [2668](#)
- SLICER, *see* obsolete routines
- SLICER3 procedure, [1936](#)
- SLIDE_IMAGE procedure, [1956](#)
- slider widgets
 - changing maximum value, [2333](#)
 - changing minimum value, [2333](#)
 - creating, [2441](#)
 - drag events, [2449](#)
 - draggable, [2442](#)
 - events returned by, [2449](#)
 - floating-point, [442](#)
 - maximum value, [2443](#)
 - minimum value, [2443](#)
 - returning minimum and maximum values, [2395](#)
- SMOOTH function, [1960](#)

- smoothing
 - CONVOL function, [331](#)
 - median, [1380](#)
 - meshes, [1414](#)
 - MIN_CURVE_SURF function, [315](#)
 - SMOOTH function, [1960](#)
- SOBEL function, [1963](#)
- SOCKET procedure, [1965](#)
- SORT function, [1970](#)
- sorting, arrays, [1970](#)
- sparse arrays
 - FULSTR, [753](#)
 - LINBCG, [1224](#)
 - READ_SPR, [1743](#)
 - SPRSAB, [1997](#)
 - SPRSAX, [2000](#)
 - WRITE_SPR, [2528](#)
- spawn, shell process, [1972](#)
- SPAWN procedure, [1972](#)
- SPEARMAN, *see* obsolete routines
- Spearman's rho rank correlation, [1679](#)
- special characters, displaying in plots, [4417](#)
- special functions
 - BETA, [166](#)
 - IBETA, [894](#)
- SPH_4PNT procedure, [1980](#)
- SPH_SCAT function, [1982](#)
- SPHER_HARM function, [1985](#)
- spherical coordinates, [379](#)
- spherical gridding
 - SPH_SCAT function, [1982](#)
 - TRIANGULATE procedure, [2132](#)
 - TRIGRID function, [2136](#)
- spherical harmonic, relation to Legendre polynomial, [1985](#)
- spherical interpolation, [1982](#)
- spherical triangulation, [2132](#)
- SPL_INIT function, [1988](#)
- SPL_INTERP function, [1990](#)
- spline
 - spline (*continued*)
 - cubic interpolation
 - establishing type, [1988](#)
 - parametric cubic, [1994](#)
 - performing, [1992](#)
 - thin-plate surface, [1433](#)
 - SPLINE function, [1992](#)
 - SPLINE_P procedure, [1994](#)
 - spreadsheet data files
 - reading, [1746](#)
 - writing, [2532](#)
 - SPRSAB function, [1997](#)
 - SPRSAX function, [2000](#)
 - SPRSIN function, [2002](#)
 - SPRSTP function, [2005](#)
 - SQRT function, [2006](#)
 - square root, [2006](#)
 - SRF files
 - reading, [1744](#)
 - writing, [2530](#)
 - stacked histogram plots (LEGO keyword), [2060](#)
 - standard, input, [786](#)
 - standard deviation
 - MOMENT function, [1443](#)
 - STDDEV function, [2010](#)
 - STANDARDIZE function, [2008](#)
 - standardized variables, [2008](#)
 - STATIC_COLOR keyword, [4279](#)
 - STATIC_GRAY keyword, [4280](#)
 - statistics
 - approximating models, [283](#)
 - fitting data
 - growth trends, [283](#)
 - least absolute deviation regression, [1214](#)
 - moving averages, [1960](#)
 - multiple linear regression, [1790](#)
 - nonlinear least-squares regression, [374](#)
 - outlying data regression, [1214](#)
 - kurtosis, [1122](#)

statistics (*continued*)

- tools
 - absolute deviation, [1443](#)
 - chi-square error, minimizing, [1229](#)
 - combinations, [654](#)
 - contingency table, [368](#)
 - cumulative sum, [2118](#)
 - factorial, [654](#)
 - frequency tables, [874](#)
 - histogram, [874](#)
 - kurtosis
 - KURTOSIS function, [1122](#)
 - MOMENT function, [1443](#)
 - Lomb normalized periodogram, [1248](#)
 - magnitude-based ranking, [1703](#)
 - maximum, [1370](#)
 - mean
 - MEAN function, [1376](#)
 - MOMENT function, [1443](#)
 - mean absolute deviation, [1378](#)
 - median, [1443](#)
 - minimum, [1430](#)
 - number generators
 - normally-distributed random, [1693](#)
 - primes, [1595](#)
 - uniformly-distributed, [1698](#)
 - permutations, [654](#)
 - skewness
 - MOMENT function, [1443](#)
 - SKEWNESS function, [1931](#)
 - sort, [1970](#)
 - standard deviation
 - MOMENT function, [1443](#)
 - STDDEV function, [2010](#)
 - T-statistic, Student's, [2116](#)
 - variance
 - MOMENT function, [1443](#)
 - VARIANCE function, [2209](#)
- STDDEV function, [2010](#)
- STDEV, *see* obsolete routines
- STEPWISE, *see* obsolete routines
- stereographic map projection, [1357](#)
- STOP procedure, [2012](#)
- stopping program execution
 - STOP procedure, [2012](#)
 - using breakpoints, [191](#)
- STR_SEP, *see* obsolete routines
- STRARR function, [2013](#)
- STRCMP function, [2014](#)
- STRCOMPRESS function, [2016](#)
- STREAMLINE procedure, [2018](#)
- streamlines, [2213](#)
- STREGEX function, [2020](#)
- STRETCH procedure, [2024](#)
- STRING function, reference, [2026](#)
- strings
 - calling
 - IDL functions from, [228](#)
 - IDL methods from, [230](#)
 - IDL procedures from, [232](#)
 - converting to lowercase, [2032](#)
 - converting to uppercase, [2057](#)
 - creating arrays, [1923](#)
 - creating string arrays, [2013](#)
 - data type, converting to, [2026](#)
 - executing contents of, [626](#)
 - extracting substrings from, [2039](#)
 - finding substrings within, [2041](#)
 - inserting strings into, [2044](#)
 - length of, [2031](#)
 - reading data from, [1766](#)
 - removing whitespace (all), [2016](#)
 - removing whitespace from (leading/trailing), [2051](#)
- STRJOIN function, [2029](#)
- STRLEN function, [2031](#)
- STRLOWCASE function, [2032](#)
- STRMATCH function, [2034](#)
- STRMESSAGE function, [2037](#)
- STRMID function, [2039](#)
- STRPOS function, [2041](#)
- STRPUT procedure, [2044](#)

- STRSPLIT function, 2046
- STRTRIM function, 2051
- STRUCT_ASSIGN procedure, reference, 2053
- STRUCT_HIDE procedure, 2055
- structures
 - concatenating, 356
 - creating and defining, 356
 - creating arrays of, 1796
 - defining, 2053
 - displaying information on currently-defined, 865
 - FSTAT, 750
 - relaxed definition
 - performing, 2053
 - restoring, 1805
 - returned by widgets, 2380
 - returning length of, 1479
 - returning number of tags, 1478
 - tag names
 - creating structure from, 356
 - returning, 2092
- structuring element, 569
- STRUPCASE function, 2057
- STUDENT_T, *see* obsolete routines
- Student's t distribution
 - cutoff value, 2085
 - probability, 2087
- Student's T-statistic, 2116
- STUDENT1_T, *see* obsolete routines
- STUDRANGE, *see* obsolete routines
- STYLE system variable field, 4386
- subscripts, converting to multi-dimensional, 123
- SUBTITLE keyword, 4341
- SUBTITLE system variable field, 4382
- subtraction operator, 4392
- summation, array elements, 2118
- Sun raster files
 - reading, 1744
 - writing, 2530
- suppressing information messages, 4372
- surf_track.pro (example file), 4238
- surface fitting, SFIT, 1873
- surface object, 3976
- surface plots
 - interactive (iTool) routine, 1052
 - interface for, 2640
 - with images and contours, 1912
- SURFACE procedure
 - duplicating transformations, 2064
 - reference, 2058
- SURFACE_FIT, *see* obsolete routines
- surfaces, shaded
 - creating, 1876
 - creating for elevation data, 1881
 - MESH_OBJ procedure, 1406
- SURFR procedure, 2064
- SVBKSB, *see* obsolete routines
- SVD, *see* obsolete routines
- SVDC procedure, 2065
- SVDFIT function, 2068
- SVSOL function, 2074
- SWAP_ENDIAN function, 2076
- SWAP_ENDIAN_INPLACE procedure, 2078
- swapping the order of bytes, 201
- SWITCH statement, 2080
- SYLK files
 - reading, 1746
 - writing, 2532
- SYMBOL keyword, 4280
- symbol object, 4010
- symbolic link files
 - reading, 1746
 - writing, 2532
- symbolic links
 - creating, 688
 - following, 695
- symbols, IDLgrSymbol, 4010
- symbols, plotting
 - graphic keyword, 4340
 - system variable, 4382
 - user-defined, 2205

symmetric array or matrix
 determining eigenvalues and eigenvectors,
 2146
 reducing, 2149
 SYMSIZE keyword, 4341
 system clock, 2082
 system variable fields
 BACKGROUND, 4379
 BLOCK, 4359
 CHANNEL, 4379
 CHARSIZE, 4379, 4383
 CHARTHICK, 4379
 CLIP, 4380
 CODE, 4359
 COLOR, 4380
 CRANGE, 4384
 FILL_DIST, 4375
 FLAGS, 4376
 FONT, 4380
 GRIDSTYLE, 4384
 LINESTYLE, 4380
 MARGIN, 4385
 MINOR, 4385
 MSG, 4360
 MSG_PREFIX, 4360
 MULTI, 4380
 N_COLORS, 4377
 NAME, 4359, 4377
 NOCLIP, 4381
 NOERASE, 4381
 NSUM, 4382
 OMARGIN, 4385
 ORIGIN, 4377
 POSITION, 4382
 PSYM, 4382
 RANGE, 4385
 REGION, 4382, 4386
 S, 4386
 STYLE, 4386
 SUBTITLE, 4382
 SYS_CODE, 4359

system variable fields (*continued*)

SYS_CODE_TYPE, 4360
 SYS_MSG, 4360
 T, 4382
 T3D, 4383
 TABLE_SIZE, 4377
 THICK, 4383, 4387
 TICKFORMAT, 4387
 TICKINTERVAL, 4388
 TICKLAYOUT, 4388
 TICKLEN, 4383, 4388
 TICKNAME, 4388
 TICKS, 4389
 TICKUNITS, 4389
 TICKV, 4389
 TITLE, 4383, 4389
 TYPE, 4390
 UNIT, 4377
 WINDOW, 4378, 4390
 X_CH_SIZE, 4378
 X_PX_CM, 4378
 X_SIZE, 4378
 X_VSIZE, 4378
 Y_CH_SIZE, 4378
 Y_PX_CM, 4378
 Y_SIZE, 4378
 Y_VSIZE, 4378
 ZOOM, 4378
 system variables
 !C, 4375
 !CPU, 4363
 !D, 4375
 !D.TABLE_SIZE, 2178
 !D.WINDOW
 creating window, 2503
 deleting specified, 2238
 setting, 2550
 !DIR, 4365
 !DLM_PATH, 4365
 !DPI, 4357
 !DTOR, 4357

system variables (*continued*)

!EDIT_INPUT, 4367
 !ERR, 2243, 4359
 !ERROR_STATE, 4359
 message text, 2037
 !EXCEPT, 4361
 !HELP_PATH, 4367
 !JOURNAL, 1110, 4367
 !MAKE_DLL, 4368
 !MAP, 1353, 4357
 !MORE, 4370
 !MOUSE, 371, 4361
 !ORDER, 4379
 !P, 4379
 !P.COLOR, 2611
 !P.MULTI, 4308
 !P.T, 4342
 !PATH, 4371
 !PI, 4357
 !PROMPT, 4372
 !QUIET, 1426, 4372
 !RADEG, 4357
 !VALUES, 4357
 !VERSION, 4372
 !WARN, 4362
 !X, 4383
 !Y, 4383
 !Z, 4383
 creating, 521
 displaying variable information, 866
 for axes, 4383
 for graphics, 4375
 obsolete, 4456
 overview, 4356
 read-only, 521
 saving, 1835
 SYSTIME function, 2082

T

T system variable field, 4382

T_CVF function, 2085
 T_PDF function, 2087
 T3D keyword, 4342
 T3D procedure, 2089
 T3D system variable field, 4383
 tab widgets
 about, 2451
 events returned by, 2459
 table widgets
 keyboard focus events, 2467
 WIDGET_TABLE, 2461
 TABLE_SIZE system variable field, 4377
 TAG_NAMES function, 2092
 tags, number in a structure, 1478
 TAN function, 2094
 tangent
 hyperbolic, 2096
 inverse, 137
 TAN function, 2094
 TANH function, 2096
 TAPRD, *see* obsolete routines
 TAPWRT, *see* obsolete routines
 TCP/IP client side socket support, 1965
 TEK_COLOR procedure, 2098
 TEK4014 keyword, 4280
 TEK4100 keyword, 4280
 Tektronix device, 4316
 TEMPORARY function, 2099
 temporary variables, 2099
 ternary operator, ?:, 4401
 tessellation, 2132
 Tessellate method, IDLgrTessellator, 4030
 tessellator object, 4021
 test functions, 1248
 CTI_TEST, 368
 FV_TEST, 758
 KW_TEST, 1124
 LNP_TEST, 1248
 MD_TEST, 1374
 R_TEST, 1682
 RS_TEST, 1828

test functions (*continued*)

S_TEST, 1831

TM_TEST, 2116

XSQ_TEST, 2637

TESTCONTRAST, *see* obsolete routines

TETRA_CLIP function, 2101

TETRA_SURFACE function, 2103

TETRA_VOLUME function, 2104

text

aligning (XYOUTS), 2656

character

height, 4378

size, 4379

thickness, 2656, 4379

width, 4378

displaying, 2561

font index, 4337

font selection, 4380

plane of, 2656

plotting in graphics windows, 2655

positioning, 4435

size, 4343

size of characters, 2656

widgets, *see* text widgets

width of, 2656

text files, counting lines, 685

text object, 4032

text widgets, 2480

appending text to, 2310

changing selected text, 2348

converting

character offsets to column/line form, 2398

line/column positions to character offsets,
2399

determining

if all events are being returned, 2398

if text widget is editable, 2398

editable, 2481

making editable after creation, 2318

events returned by, 2480

specifying, 2309

text widgets (*continued*)

events returned by, 2480

WIDGET_TEXT, 2489

keyboard focus events, 2483

returning

line number of top line in viewport, 2399

number of characters, 2398

offsets of text selection, 2398

selected text, 2348

setting

text selection, 2335

top line, 2336

setting keyboard focus to, 2323

suppressing newline characters, 2326

THICK keyword, 4342

THICK system variable field

axes, 4387

plotting, 4383

thickness of characters, 2656

THIN function, 2106

thinning images, 2106

thin-plate-spline interpolation

GRID_TPS function, 801

MIN_CURVE_SURF function, 1433

THREED procedure, 2108

three-dimensional

transformations

array transforms, 2218

coordinate conversion, 336

coordinates, 458

duplicating SURFACE transforms, 2064

plotting

adjusting, 458

coordinate conversion, 336

scaling

setup, 1842

unit cube, 1844

T3D keyword, 4383

THRESHOLD keyword, 4281

throw, C++ language, 236

- tick marks
 - annotation
 - graphic keyword, 4349
 - system variable, 4388
 - data values for
 - graphic keyword, 4350
 - system variable, 4389
 - getting values of, 4345
 - intervals
 - graphic keyword, 4349
 - system variable, 4389
 - layout in individual axes, 4348
 - length
 - graphic keyword, 4342
 - system variable, 4383
 - length on individual axes
 - graphic keyword, 4348
 - system variable, 4388
 - linestyles, 4343
 - minor
 - graphic keyword, 4343
 - system variable, 4385
 - string labels for, 4387
 - styles, 4384
 - suppressing
 - graphic keyword, 4349
 - system variable, 4389
 - units for labeling, 4349
- TICKFORMAT system variable field, 4387
- TICKINTERVAL system variable field, 4388
- TICKLAYOUT system variable field, 4388
- TICKLEN keyword, 4342
- TICKLEN system variable field
 - axes, 4388
 - plotting, 4383
- TICKNAME system variable field, 4388
- TICKS system variable field, 4389
- TICKUNITS system variable field, 4389
- TICKV system variable field, 4389
- TIFF files
 - reading, 1750
 - TIFF files (*continued*)
 - writing, 2534
 - TIFF_DUMP, *see* obsolete routines
 - TIFF_READ, *see* obsolete routines
 - TIFF_WRITE, *see* obsolete routines
 - time
 - converting from string to binary, 171
 - returning current, 2082
 - TIME_TEST2 procedure, 2110
 - TIMEGEN function, 2111
 - TIMES keyword, 4281
 - time-series analysis
 - autocorrelation, 86
 - autocovariance, 86
 - autoregressive modeling
 - TS_COEF function, 2156
 - TS_FCAST function, 2160
 - cross correlation, 209
 - cross covariance, 209
 - forward differencing, 2158
 - TITLE keyword, 4343
 - TITLE system variable field
 - axes, 4389
 - plotting, 4383
 - TM_TEST function, 2116
- t-means test, 2116
- toggle buttons, WIDGET_BUTTON, 2280
- top margin, setting, 4385
- top-level base, 2257
- TOTAL function, 2118
- TQLI, *see* obsolete routines
- TRACE function, 2123
- traceback information
 - displaying, 866
 - returning, 860
- TrackBall
 - class, 4231
 - methods, 4231
 - Init, 4234
 - Reset, 4235
 - Update, 4237

- TrackBall (*continued*)
 - properties, [4232](#)
- transformation matrices, [4382](#)
- transforming, map coordinates, [1322](#), [1351](#)
- transforms
 - Fourier, [656](#)
 - Hough, [882](#)
 - Radon, [1684](#)
- Translate method, IDLgrModel, [3778](#)
- translation, [2089](#)
- TRANSLATION keyword, [4281](#)
- translation tables, bypassing, [4253](#)
- transparency
 - image objects, [3716](#)
 - polygon objects, [3862](#)
 - surface objects, [3997](#)
- TRANSDRIFT function, [2126](#)
- transposing arrays, [2126](#)
- Transverse Mercator map (UTM) projection, [1357](#)
- TRED2, *see* obsolete routines
- tree widgets
 - about, [2492](#)
 - events returned by, [2501](#)
- TRI_SURF function, [2128](#)
- TRIANGULATE function, [1388](#)
- TRIANGULATE procedure, [2132](#)
- triangulation
 - Delaunay, [1635](#)
 - scattered data points, [1630](#)
 - spherical, [2132](#)
 - TRIANGULATE procedure, [2132](#)
 - TRIGRID function, [2136](#)
- TRIDAG, *see* obsolete routines
- tridiagonal array or matrix, [1204](#)
 - determining eigenvalues and eigenvectors, [2146](#)
 - Householder's method, [2149](#)
 - solving, [2151](#)
- TRIGRID function, [2136](#)
- trilinear interpolation, [1007](#)
- trimming strings, [2051](#)
- TRIQL procedure, [2146](#)
- TRIRED procedure, [2149](#)
- TRISOL function, [2151](#)
- TRNLOG, *see* obsolete routines
- TRUE_COLOR keyword, [4281](#)
- TrueColor
 - images
 - and the PostScript device, [4304](#)
 - converting to pseudo-color, [277](#)
 - displaying, [2167](#)
 - reading, [2175](#)
- true-color visuals, [4257](#)
- TrueType fonts
 - overview, [4414](#)
 - samples, [4442](#)
 - specifying with DEVICE, [4277](#)
- TRUNCATE_LUN procedure, [2154](#)
- truncating file contents, [2154](#)
- TS_COEF function, [2156](#)
- TS_DIFF function, [2158](#)
- TS_FCAST function, [2160](#)
- TS_SMOOTH function, [2162](#)
- TT_FONT keyword, [4282](#)
- TTY keyword, [4282](#)
- TV procedure, [2165](#)
- TVCRS procedure, [2169](#)
- TVDELETE, *see* obsolete routines
- TVLCT procedure, [2171](#)
- TVRD function, [2174](#)
- TVRDC, *see* obsolete routines
- TVSCL procedure, [2178](#)
- TVSET, *see* obsolete routines
- TVSHOW, *see* obsolete routines
- TVWINDOW, *see* obsolete routines
- two-dimensional Gaussian fit, [773](#)
- type conversion
 - to 64-bit integer, [1266](#)
 - to byte, [199](#)
 - to complex
 - COMPLEX function, [292](#)

type conversion (*continued*)
 to complex
 DCOMPLEX function, 497
 to double-precision, 590
 to integer, 731
 to longword, 1264
 to single-precision, floating-point, 735
 to string, 2026
 to unsigned 64-bit integer, 2197
 to unsigned integer, 2183
 to unsigned longword, 2195
 TYPE system variable field, 4390
 type-ahead buffer, 786

U

UINDGEN function, 2181
 UINT function, 2183
 UINTARR function, 2185
 UL64INDGEN function, 2187
 ULINDGEN function, 2189
 ULON64ARR function, 2191
 ULONARR function, 2193
 ULONG function, 2195
 ULONG64 function, 2197
 undocumented object classes, 1488, 2677
 unformatted binary data, 1768, 2548
 uniform random deviates, 1701
 uniformly-distributed random numbers, 1698
 UNIQ function, 2199
 unit number, logical, 1513
 UNIT system variable field, 4377
 UNIX, changing file permissions, 664
 unmapping widgets, 2263
 unsigned 64-bit integer
 arrays, 2187
 data type, converting to, 2197
 unsigned arrays, longword, 2189
 unsigned integer
 arrays, 2181
 data type, converting to, 2183

unsigned longword
 arrays, 2193
 data type, converting to, 2195
 upper margin, setting, 4385
 uppercase, converting strings to, 2057
 user directory, 105
 USER_FONT keyword, 4282
 user-defined plotting symbols, 2205
 USERSYM procedure, 2205
 using external modules, 217
 UTM (Transverse Mercator) map projection,
 1357

V

validating variable names, 926
 VALUE_LOCATE function, 2207
 variables
 associated, 134
 data type, determining, using SIZE function,
 1926
 deleting, 523
 interactive editing tool, 2642
 named, 61, 61, 2667
 reading display images into (TVRD func-
 tion), 2174
 returning information on, 859
 saving, 1835
 temporary, 2099
 valid name, 926
 variance
 FV_TEST function, 758
 MOMENT function, 1443
 VARIANCE function, 2209
 VAX_FLOAT, *see* obsolete routines
 VECTOR_FIELD procedure, 2211
 vector-drawn fonts
 ! character, 4435
 displaying, 1914
 editing (EFONT procedure), 594
 overview, 4414

- vector-drawn fonts (*continued*)
 - samples, [4445](#)
 - special characters, [4417](#)
 - vectors, drawing arrowheads, [126](#)
 - VEL procedure, [2213](#)
 - velocity field, plotting
 - FLOW3 procedure, [739](#)
 - VEL procedure, [2213](#)
 - VELOVECT procedure, [2215](#)
 - VELOVECT procedure, [2215](#)
 - VERT_T3D function, [2218](#)
 - vertices
 - merged mesh example, [1401](#)
 - mesh smooth example, [1414](#)
 - view object, [4056](#)
 - viewgroup object, [4073](#)
 - viewing, HDF5 files, [840](#)
 - VMSCODE, *see* obsolete routines
 - VOIGT function, [2220](#)
 - volume object, [4085](#)
 - volume slices, [1936](#)
 - volumes
 - extracting slices, [646](#)
 - interactive (iTool) routine, [1087](#)
 - rendering, [1614](#)
 - searching for objects, [1861](#)
 - visualizing
 - POLYSHADE function, [1584](#)
 - PROJECT_VOL function, [1614](#)
 - SHADE_VOLUME procedure, [1884](#)
 - VOXEL_PROJ function, [2225](#)
 - volumetric reconstruction, [1775](#)
 - Voronoi diagrams, [1635](#)
 - VORONOI procedure, [2223](#)
 - voxel rendering, [2225](#)
 - VOXEL_PROJ function, [2225](#)
 - VRML object, [4114](#)
 - VT240 keyword, [4282](#)
 - VT240 terminal, [4315](#)
 - VT330 terminal, [4315](#)
 - VT340 keyword, [4282](#)
 - VT340 terminal, [4315](#)
- ## W
- WAIT procedure, [2231](#)
 - WARP_TRI function, [2232](#)
 - warping
 - images, [1567](#)
 - images to maps
 - MAP_IMAGE function, [1313](#)
 - MAP_PATCH function, [1318](#)
 - polynomial, [1567](#)
 - using the Z-buffer, [1580](#)
 - WATERSHED function, [2234](#)
 - Wavefront Advanced Data Visualizer
 - reading, [1760](#)
 - writing, [2546](#)
 - Wavefront files
 - reading, [1760](#)
 - writing, [2546](#)
 - wavelet transform
 - discrete, WTN function, [2554](#)
 - WDELETE procedure
 - reference, [2238](#)
 - window systems, [4286](#)
 - weather fronts, plotting, [2239](#)
 - WEOF, *see* obsolete routines
 - WEXMASTER (widget examples), [2587](#)
 - WF_DRAW procedure, [2239](#)
 - WHERE function, [2242](#)
 - WHILE...DO statement, [2246](#)
 - whitespace
 - removing all, [2016](#)
 - removing leading/trailing, [2051](#)
 - WIDED, *see* obsolete routines
 - widget events, [2379](#)
 - WIDGET_ACTIVEX function, [2247](#)
 - WIDGET_BASE function, [2254](#)
 - WIDGET_BUTTON function, [2280](#)
 - WIDGET_COMBOBOX function, [2295](#)
 - WIDGET_CONTROL procedure, reference, [2305](#)

- WIDGET_DISPLAYCONTEXTMENU function, [2351](#)
- WIDGET_DRAW function, [2353](#)
- WIDGET_DROPLIST function, [2370](#)
- WIDGET_EVENT function, reference, [2379](#)
- WIDGET_INFO function, reference, [2383](#)
- WIDGET_KILL_REQUEST event, [2272](#)
- WIDGET_LABEL function, [2407](#)
- WIDGET_LIST function, [2414](#)
- WIDGET_MESSAGE, *see* obsolete routines
- WIDGET_PROPERTY SHEET function, reference, [2424](#)
- WIDGET_SLIDER function, [2441](#)
- WIDGET_TAB function, [2451](#)
- WIDGET_TABLE function, [2461](#)
- WIDGET_TEXT function, [2480](#)
- WIDGET_TREE function, [2492](#)
- widgets
 - aligning (ALIGN_XXX keywords), [2257](#), [2425](#)
 - aligning keywords, [2248](#)
 - animation, [384](#)
 - annotation, [103](#)
 - base, WIDGET_BASE, [2254](#)
 - buttons
 - bitmap labels, [1762](#)
 - groups, [402](#)
 - release events, [2287](#)
 - WIDGET_BUTTON, [2280](#)
 - callbacks
 - WIDGET_ACTIVEX, [2250](#)
 - WIDGET_BASE, [2262](#), [2265](#)
 - changing appearance of, [2266](#)
 - clearing events (CLEAR_EVENTS keyword), [2312](#)
 - color
 - index
 - CW_CLR_INDEX, [409](#)
 - CW_RGBSLIDER, [481](#)
 - resources, [2267](#)
- widgets (*continued*)
 - color
 - selection, [413](#)
 - combobox, [2295](#)
 - compound
 - 3D orientation widget, [458](#)
 - animation widget, [384](#)
 - button group widget, [402](#)
 - color index selection widget, [409](#)
 - color selection widget, [413](#)
 - data entry field widget, [421](#)
 - display zoom widget, [487](#)
 - manipulating 3D orientation, [396](#)
 - pulldown menu widget, [472](#)
 - RGB widget sliders, [481](#)
 - ROI definition widget, [416](#)
 - slider widget, [442](#)
 - template for creating, [485](#)
 - default font for, [2314](#)
 - destroying, using WIDGET_CONTROL, [2316](#)
 - determining if widgets are realized
 - (ACTIVE keyword), [2385](#)
 - (REALIZED keyword), [2394](#)
 - disabling and enabling screen updates (UPDATE keyword), [2346](#)
 - draw, WIDGET_DRAW, [2353](#)
 - droplist, WIDGET_DROPLIST, [2370](#)
 - events, WIDGET_EVENT, [2379](#)
 - examples, [2587](#)
 - exclusive buttons, [2260](#)
 - field, [421](#)
 - form, [433](#)
 - getting user values, [2320](#)
 - help buttons, [2286](#)
 - hiding and showing, [2339](#)
 - horizontal size, changing
 - SCR_XSIZE, [2330](#)
 - XSIZE, [2349](#)
 - iconifying, [2323](#)
 - invalid IDs, [2310](#), [2381](#)

widgets (*continued*)

- label, creating, [2407](#)
- list, creating, [2414](#)
- main event loop for, [2581](#)
- mapping
 - mapping and unmapping, [2325](#)
 - using WIDGET_BASE, [2263](#)
- menu bars, [2263](#)
- message dialog box, [547](#)
- modal, [547](#)
- non-exclusive buttons, [2265](#)
- positioning, children in a base, [2254](#)
- property sheets, [2424](#)
- pulldown menu
 - creating, [472](#)
 - separators, [2289](#)
- realizing, [2327](#)
- region of interest, [416](#)
- registered, [2619](#)
- registering with XMANAGER, [2581](#)
- resetting all widgets, [2329](#)
- resizing (DYNAMIC_RESIZE keyword)
 - WIDGET_BUTTON, [2284](#)
 - WIDGET_COMBOBOX, [2296](#)
 - WIDGET_DROPLIST, [2371](#)
 - WIDGET_LABEL, [2408](#)
- returning
 - children of, [2385](#)
 - information about, [2383](#)
 - name of event handler procedure, [2388](#)
 - parent of, [2392](#)
 - siblings of, [2395](#)
 - size of (GEOMETRY keyword), [2389](#)
 - tracking event status, [2400](#)
 - type of
 - setting NAME keyword, [2392](#)
 - TYPE code definitions, [2401](#)
 - validity of, [2402](#)
- sending event to (SEND_EVENT keyword), [2330](#)

widgets (*continued*)

- sensitizing and de-sensitizing
 - WIDGET_ACTIVEX, [2251](#)
 - WIDGET_BASE, [2269](#)
 - WIDGET_BUTTON, [2288](#)
 - WIDGET_COMBOBOX, [2300](#)
 - WIDGET_CONTROL, [2331](#)
 - WIDGET_DRAW, [2361](#)
 - WIDGET_DROPLIST, [2374](#)
 - WIDGET_LABEL, [2410](#)
 - WIDGET_LIST, [2418](#)
 - WIDGET_SLIDER, [2445](#)
 - WIDGET_TAB, [2456](#)
 - WIDGET_TABLE, [2470](#)
 - WIDGET_TEXT, [2485](#)
 - WIDGET_TREE, [2497](#)
- setting buttons, [2331](#)
- showing and hiding, [2339](#)
- size
 - changing horizontal
 - SCR_XSIZE, [2330](#)
 - XSIZE, [2349](#)
 - changing vertical
 - SRC_YSIZE, [2330](#)
 - YSIZE, [2350](#)
- slider
 - CW_FSLIDER, [442](#)
 - WIDGET_SLIDER, [2441](#)
- space between children, [2269](#)
- tab, WIDGET_TAB, [2451](#)
- table, WIDGET_TABLE, [2461](#)
- template for creating, [2589](#)
- text, WIDGET_TEXT, [2480](#)
- tracking events, [2273](#)
- tree, WIDGET_TREE, [2492](#)
- unmapping
 - WIDGET_BASE, [2263](#)
 - WIDGET_CONTROL, [2325](#)
- values, [2321](#)
- version of implementation, [2402](#)

- widgets (*continued*)
 - vertical size, changing
 - SRC_YSIZE, 2330
 - YSIZE, 2350
 - viewing widgets managed by XMANAGER, 2591
 - zoom, 487
- width of text, 2656
- Wilcoxon Rank-Sum Test, 1828
- WILCOXON, *see* obsolete routines
- window object, 4135
- window objects, maximum size, 4135
- WINDOW procedure
 - reference, 2503
 - using, 4286
- WINDOW system variable field
 - index of current, 4378
 - plotting, 4390
- WINDOW_STATE keyword, 4282
- windows
 - backing store
 - about, 4286
 - device keyword, 4271
 - WINDOW procedure, 2504
 - copying areas, 4254
 - copying pixels from, 4254
 - creating, 2503
 - deleting, 2238
 - display size, 4378
 - draw widgets, 2353
 - erasing, 610
 - exposing, 2552
 - height, 2505
 - hiding, 2552
 - iconifying, 2552
 - ID for draw widgets, 2363
 - index of currently open, 4378
 - number of colors, 4377
 - pixmap, 2504
 - position of, 4263, 4382
 - positioning, 2505
 - windows (*continued*)
 - selecting current, 2550
 - systems, 4286
 - visible area of display, 4378
 - width, 2505
 - Windows display device (WIN), 4244
 - Windows Metafile Format, 4244
 - Windows platform, changing file permissions, 664
 - wire-mesh surface plots, 2058
 - WMENU, *see* obsolete routines
 - WMF, 4244
 - World Wide Web, 1438
 - write mask
 - GET_WRITE_MASK keyword, 4263
 - SET_WRITE_MASK keyword, 4279
 - WRITE_BMP procedure, 2507
 - WRITE_GIF procedure, 2510
 - WRITE_IMAGE procedure, 2512
 - WRITE_JPEG procedure, 2514
 - WRITE_NRIF procedure, 2519
 - WRITE_PICT procedure, 2521
 - WRITE_PNG procedure, 2523
 - WRITE_PPM procedure, 2526
 - WRITE_SPR procedure, 2528
 - WRITE_SRF procedure, 2530
 - WRITE_SYLK function, 2532
 - WRITE_TIFF procedure, 2534
 - WRITE_WAV procedure, 2545
 - WRITE_WAVE procedure, 2546
 - WRITEU procedure, 2548
 - writing
 - BMP files, 2507
 - GIF files, 2510
 - JPEG files, 2514
 - NRIF files, 2519
 - PGM files, 2526
 - PICT files, 2521
 - PPM files, 2526
 - SRF files, 2530
 - TIFF files, 2534

writing (*continued*)
 wave files, [2546](#)

WSET procedure
 reference, [2550](#)
 using, [4286](#)

WSHOW procedure
 reference, [2552](#)
 using, [4286](#)

WTN function, [2554](#)

X

X resources, widget colors, [2267](#)

X Windows

bitmap files, reading, [1762](#)
 Dump files, reading, [1764](#)
 fonts, [2568](#)
 resource names, [2266](#)

X Windows device

DirectColor visual, [4257](#)
 PseudoColor visual, [4271](#)
 reference, [4319](#)
 StaticColor visual, [4279](#)
 StaticGray visual, [4280](#)
 TrueColor visual, [4281](#)
 visuals, [4319](#)

X Windows resource names, [2299](#)

X_CH_SIZE system variable field, [4378](#)

X_PX_CM system variable field, [4378](#)

X_SIZE system variable field, [4378](#)

X_VSIZE system variable field, [4378](#)

XANIMATE, *see* obsolete routines

XBACKREGISTER, *see* obsolete routines

XBM_EDIT procedure, reference, [2559](#)

XCHARSIZE keyword, [4343](#)

XDISPLAYFILE procedure, [2561](#)

XDL, *see* obsolete routines

XDR format (floating point values), [201](#)

XDXF procedure, [2564](#)

XFONT function, [2568](#)

XGRIDSTYLE keyword, [4343](#)

XINTERANIMATE procedure, [2570](#)

XLOADCT procedure, [2578](#)

XMANAGER procedure, reference, [2581](#)

XMANAGERTOOL, *see* obsolete routines

XMARGIN keyword, [4343](#)

XMAX machine-specific parameter, [1287](#)

XMENU, *see* obsolete routines

XMIN machine-specific parameter, [1287](#)

XMINOR keyword, [4343](#)

XML

parsers

IDLffXMLDOM, [2887](#)

IDLffXMLSAX, [3039](#)

XMNG_TMPL procedure, [2589](#)

XMTOOL procedure, [2591](#)

XOBJVIEW procedure, [2593](#)

XOBJVIEW_ROTATE procedure, [2603](#)

XOBJVIEW_WRITE_IMAGE procedure,
[2605](#)

XOFFSET keyword

graphics positioning, [4283](#)

PostScript positioning, [4305](#)

XON_XOFF keyword, [4283](#)

XOR operator, [4398](#)

XPALETTE procedure, [2607](#)

XPCOLOR procedure, [2611](#)

XPDMENU, *see* obsolete routines

XPLOT3D procedure, [2612](#)

XRANGE keyword, [4344](#)

XREGISTERED function, reference, [2619](#)

XROI

growing a region, [2630](#)

importing images, [2629](#)

procedure, [2621](#)

using, [2627](#)

XSIZE keyword, [4283](#)

XSQ_TEST function, [2637](#)

XSTYLE keyword, [4344](#)

XSURFACE procedure, [2640](#)

XTHICK keyword, [4344](#)

XTICK_GET keyword, [4345](#)

XTICKFORMAT keyword, [4345](#)
 XTICKINTERVAL keyword, [4348](#)
 XTICKLAYOUT keyword, [4348](#)
 XTICKLEN keyword, [4348](#)
 XTICKNAME keyword, [4349](#)
 XTICKS keyword, [4349](#)
 XTICKUNITS keyword, [4349](#)
 XTICKV keyword, [4350](#)
 XTITLE keyword, [4350](#)
 XVAREDIT procedure, [2642](#)
 XVOLUME procedure, [2644](#)
 XVOLUME_ROTATE procedure, [2650](#)
 XVOLUME_WRITE_IMAGE procedure,
[2653](#)
 xwd files, reading, [1764](#)
 XYOUTS procedure, [2655](#)
See also positioning

Y

Y_CH_SIZE system variable field, [4378](#)
 Y_PX_CM system variable field, [4378](#)
 Y_SIZE system variable field, [4378](#)
 Y_VSIZE system variable field, [4378](#)
 YCHARSIZE keyword, [4343](#)
 YGRIDSTYLE keyword, [4343](#)
 YMARGIN keyword, [4343](#)
 YMINOR keyword, [4343](#)
 YOFFSET keyword
 graphics positioning, [4283](#)
 PostScript positioning, [4305](#)
 YRANGE keyword, [4344](#)
 YSIZE keyword, [4284](#)
 YSTYLE keyword, [4344](#)
 YTHICK keyword, [4344](#)
 YTICK_GET keyword, [4345](#)
 YTICKFORMAT keyword, [4345](#)
 YTICKINTERVAL keyword, [4348](#)
 YTICKLAYOUT keyword, [4348](#)
 YTICKLEN keyword, [4348](#)
 YTICKNAME keyword, [4349](#)

YTICKS keyword, [4349](#)
 YTICKUNITS keyword, [4349](#)
 YTICKV keyword, [4350](#)
 YTITLE keyword, [4350](#)

Z

Z keyword, [4351](#)
 ZAPFCHANCERY keyword, [4284](#)
 ZAPFDINGBATS keyword, [4284](#)
 Z-buffer
 closing, [4253](#)
 reference, [4328](#)
 using with POLYFILL, [1578](#)
 using with POLYSHADE, [1584](#)
 warping images to polygons, [1580](#)
 ZCHARSIZE keyword, [4343](#)
 zeroing byte arrays, [197](#)
 ZGRIDSTYLE keyword, [4343](#)
 ZMARGIN keyword, [4343](#)
 ZMINOR keyword, [4343](#)
 ZOOM procedure, [2659](#)
 ZOOM system variable field, [4378](#)
 zoom widget, [487](#)
 ZOOM_24 procedure, [2661](#)
 ZoomIn method, IDLgrWindow, [4178](#)
 ZoomOut method, IDLgrWindow, [4179](#)
 ZRANGE keyword, [4344](#)
 ZROOTS, *see* obsolete routines
 ZSTYLE keyword, [4344](#)
 ZTHICK keyword, [4344](#)
 ZTICK_GET keyword, [4345](#)
 ZTICKFORMAT keyword, [4345](#)
 ZTICKINTERVAL keyword, [4348](#)
 ZTICKLAYOUT keyword, [4348](#)
 ZTICKLEN keyword, [4348](#)
 ZTICKNAME keyword, [4349](#)
 ZTICKS keyword, [4349](#)
 ZTICKUNITS keyword, [4349](#)
 ZTICKV keyword, [4350](#)
 ZTITLE keyword, [4350](#)
 ZVALUE keyword, [4351](#)

