DEBUGGING IN AVS 5.5

CHAPTER ELEVEN

Introduction

This chapter is a summary of suggestions and tips on developing and debugging AVS modules; it draws together a number of techniques and features that are available in different areas of the product. More detailed documentation is referenced when available elsewhere.

AVS 5.5 NOTE: This entire chapter is new for AVS 5.5. It covers features documented in prior releases as well as new features or documentation in such areas as internal debugging flags, include files, and test data generator modules.

The information is organized into the following sections:

- Coding and Porting - general advice on approaching portability
- Building for Debugging - suggestions on making modules easier to debug
- Examples - borrowing code and techniques from module source code examples
- Command Line Interpreter - setting up reproducible tests and using debug flags
- Debugging Modules - using avs_dbx
- Input and Output Data - using standard test data sets during shakeout
- Kernel Debugging - seeing what is going on in the overall system
- Working with AVS Customer Support - preparing questions, options

Coding and Porting

Module Generator

If you are writing a new module, use the Module Generator module to create and manage the module throughout its life span. Described in Chapter 2 of the Application Guide, the Module Generator will not only create a source code "template" for your new module but it will read your module back in later and allow you to make changes as it evolves. It provides support for the following operations:

- Creating source code - you enter the desired name, properties, inputs, parameters, outputs, etc. and it generates the matching source code for either a subroutine or coroutine, creating the source file with appropriate header file includes, description and compute functions with proper declarations in C or FORTRAN, and initialization calls

- Creating a makefile - a working Makefile using the recommended Makeinclude definitions file is generated, supporting general portability

- Creating man page template - creates a sample man page to fill in

- Compiling the module - compiles the module for debugging or not as desired

- Loading the module - loads the module into the current Network Editor, current module library

- Debugging the module - runs the module in *avs_dbx*, the module debugger

- Reading existing module source - reads the module back in, interpreting the header information (inputs, outputs, parameters) and remembering user code between special markers to include in the new output.

Common issues

There are a few areas that frequently cause problems that you should be aware of.

Porting between 32 and 64 bit platforms

On 32 bit platforms, the integer (int) and pointer (int*, void*, etc) data types are both 32 bits and can be (and often are) treated as the same; poor programming practice has tolerated code in which these types are cast back and forth to each other, or in which pointer values are accepted and stored as ints, etc.

Unfortunately, the SGI N64 and Dec Alpha platforms are 64 bit platforms that store ints as 32 bits, and pointers as 64 bits. This can result in bizarre values arising inside your 32-bit code when it is run on a 64-bit platform - pointer values can be sliced in half, sometimes resulting in extreme values or 0.

The bottom line is to carefully use ints and pointers correctly and look for problems like this if you are having porting problems.

FORTRAN and 64 bit platforms

When AVS 5 was originally developed, FORTRAN had no pointer variable capabilities. In order to pass pointers to fields and other data structures, FORTRAN functions would receive pointer values as INTEGERs, then pass these into C functions to do operations such as reallocating memory or computing offsets. In order to have FORTRAN code that runs on both 32 and 64 bit platforms, you need to change the declarations for values that receive pointer arguments from INTEGER to INTEGER*8. See "FORTRAN Modules on 64 bit Systems" in Chapter 3, "AVS 5.5 on UNIX Platforms", for more information on how best to do this.

XDR data format

Binary data file formats vary from one platform to the next and can cause serious problems between 32 and 64 bit platforms and also between big- and little-endian systems. Use "XDR" data format whenever possible to minimize these issues; see documentation on the Read Field and Write Field modules for more information.

Incorrect input and output declarations

It is easy to misdeclare the compute function arguments or not handle data allocation correctly, causing unexpected behavior downstream when the data isn’t what is expected.

- Use the Module Generator and the "include hints" operation as much as possible; this will help reduce initial coding errors.

- Review the available AVS 5 example modules in *$AVS_PATH/examples* to find an example of data being passed or allocated similarly to what you need. The new *corout_f* FORTRAN coroutine example shows most data types being passed in and copied to outputs. The *widgets* examples are also good samples of how to pass parameter values.

- FORTRAN string output requires that you allocate a string that will persist after you exit the compute routine, something that is difficult to do in FORTRAN. See the newly documented *AVSoutput_string* function in Chapter 3, "AVS 5.5 on UNIX" under the "New documentation" section for help in doing this.

- Use output diagnostic modules downstream to check that you are getting what you expect (see section below).

- Visit the IAC module repository (*http://www.iavsc.org*) for related module source.

- Request AVS 5 module source code for modules handling similar operations. For more information see "Working with AVS Support" below.

Performance - Reducing data copying and processes

AVS 5 uses a variety of techniques to minimize the number of data copies it needs, including shared memory and direct module communications. These work reasonably well but there are still times where AVS 5 will make extra copies of data, for example to send the data from module A (original copy), through the kernel (2nd copy), to module B downstream (3rd copy). This particularly affects user data types which do not use shared memory.

The best solution, when possible, is to combine modules into a single "mongo" module and clean them up to be "reentrant" and "cooperative" (see next issue about Global Data). When the modules are in the same process, they can pass data just by passing pointers rather than having to serialize the data and transmit it between processes, which takes time and makes extra copies. Cutting down on the number of module processes also saves memory usage and disk space, and reduces the number of processes at run time. See "Multiple Modules in a Single Process", page 4-21 in the *AVS Developer's Guide*, for more information on making modules behave well in shared processes.

You can combine your modules with AVS 5 standard modules as needed as well, because the module binaries are included in a series of libraries. See the *$AVS_PATH/examples/multi_hog.c* for an example of how to do this.

Global Data and AVSstatic

In order to make your module code "cooperative" and "reentrant" (allowing multiple module instances to share the same process), you need to be especially careful about using global variables (variables declared outside a particular function). These are often used for remembering state information between compute function calls, but when more than one module instance is involved this "state" information may not be saved as originally expected.

The solution is to use the **AVSstatic** variable, documented in the section on Mutiple Modules (page 4-21 *AVS Developer's Guide*), mentioned above. You declare a local "state" data structure that can hold all the "global" state info for your module, make an instance of this data structure and store the pointer to it in AVSstatic. When your module is finished the value is stored away and then reset back when your module runs the next time. The AVSstatic variable is now declared in *$AVS_PATH/include/avs.h*; in releases prior to AVS 5.5 you will need to declare this variable locally as follows:

```
extern char *AVSstatic;
```

Port.h

An undocumented header file, *$AVS_PATH/include/port.h*, contains a number of platform specific compiler flags and declarations that permit AVS supported modules to work across machines. It provides flags to define characteristics such as endian-ness, BSD or not, data sizes of LONGS, etc. If you run into platform specific behavior, you may want to first review this file to see if there are appropriate declarations covering the areas of concern. Few changes have been made in this file over the last few releases.

Building for Debugging

When you write your module, it can be helpful to build in debugging aides that will produce diagnostic output when you want it. There are a few ways to do this.

AVSmessage

The AVSmessage function and its convenience forms (AVSinfo, AVSwarning, AVSerror, and AVSfatal) provide a means of sending multiple levels of information to dialog boxes, stdout, and a log file. Using AVSinfo may allow you to include debug output from your module. For more information on AVSmessage see "Handling Errors in Modules", page 3-13 in the *AVS Developer's Guide.*

The AVSmessage function has four levels of severity - (1) information, (2) warning, (3) error, and (4) fatal - corresponding to the four shortcut function calls. By default there is a "dialog cutoff level" of 2 - levels 2 through 4 will result in a dialog box being displayed with the message, requiring the user to acknowledge before the module continues. Below this cutoff level (2), lower level messages (in this case just "(1)information") are displayed to standard output (stdout). All messages are recorded to a log file, called */tmp/avs.log_<ProcessId>* for the duration of the session. You can tell AVS to preserve this log file after the session is completed by using the *SaveMessageLog* option in your *.avsrc* file.

This "cutoff" level can be adjusted using the CLI "debug" command on the "AVSmessage_dialog" flag. Start up using "avs -cli" and then type "debug" to see or change the current debug flag settings. The level can be raised or lowered to suppress or enable dialog output above a certain level.

See the example *$AVS_PATH/examples/widgets* for an example of how AVSmessage can be used.

Invisible parameters

You can include extra debug control parameters in your module declaration that are not shown to users by default. Use the *AVSconnect_widget* function to select "none" (or clicking the option in the Module Generator) and no widget will be attached by default. Later when you wish to use this invisible parameter you can either use the CLI "parm_set" command for your module or bring up the Module Editor and from that the Parameter Editor to attach a widget to the parameter, making it accessible. One thing your parameter might control is additional debug output to stdout or AVSinformation().

Text browsers

The **print field** module shows an example of a module that writes text output to a file and then provides a window onto that file using a "text browser" widget. The parameter provides the name of a file that the module then writes to; when the module signals for a widget update, the widget reads in the text file to refresh its output. Though narrow by default, the module can select to make this window wider and parent it to the main shell window if desired.

The end user can do the same thing using the Layout Editor mode in the Network Editor, reparenting the module panel to the shell window, stretching the text browser, and reconfiguring the module panel. For an example of how to use text browser widgets this way see the new example module, *$AVS_PATH/examples/geom_write.c*

Debugging in AVS 5

Memory leaks

A number of good commercial memory management packages, such as Purify from Rational Software, are available to help you track down memory leaks and mismanagement. AVS 5 also contains a less powerful memory management diagnostic package, invoked by including *$AVS_PATH/include/mem_defs.h* and defining MEM_DEFS_ENABLE during compilation.

This package redefines the front end to many string management and memory management functions and can provide verbose output. These macros insert a layer of error checking routines between the application and the system memory allocation routines. For instance if you call 'malloc', this substitutes a macro for the function call so it calls 'MEMmalloc' instead. It collects statistics, does error checking, and optionally prints out debugging information. These substitutions can cause some strange syntax errors under certain conditions. See "Memory Allocation Debugging", page 4-1 in the *AVS Developer's Guide* for more details.

AVS Examples

When you need module source code examples to see how more complex operations are done there are a number of sources.

- *$AVS_PATH/examples* - the first place to start for examples of C, FORTRAN, and C++ example modules. Several new module examples have been added in AVS 5.5; see Chapter 3, "AVS 5.5 on UNIX Platforms" under the "New Example modules" section.

- *AVS_DEMOS* - the extended demo package includes a number of customer module contributions which are compiled from source when the package is installed. See the chapter on "Extended Features" under the "Installing the Demos" and "Running the Demos" sections for more information.

- International AVS Center: The IAC has a large number of customer and AVS contributed source code modules. Visit their web site at www.iavsc.org. An older snapshot of the IAC web site is available on the "Cool CD" originally shipped with AVS 5.3 and all new shipments.

- AVS supported module source code: You can request the source code for most supported modules from AVS Customer Support after signing a source code agreement. Nearly all non-builtin modules are available. For more information see "Working with AVS Support" below.

Command Line Interpreter

The AVS Command Line Interpreter (CLI) can be useful for testing in the following ways:

- providing access to a number of debug switches for producing more verbose output
- timing how long it takes a particular command to execute
- recording test and demo scripts to test networks or reproduce problems more easily
- playing back test scripts providing either stepping or time delayed execution or early termination.

In order to use the CLI directly, just start "avs -cli" and the "avs>" prompt indicates AVS is listening for CLI commands. For more general information see Chapter 5 in the *AVS Developer's Guide.*

Debug switches

The "debug" command will either list or change a series of debug switches. The values shown are the default settings. A number of these are unsupported and incomplete but work well enough to be very useful.

```
debug  [UNSUP] Set or view internal debug switches
Without arguments will list and describe the known switches.
With just the name of the switch it will show the current value.
All switches take an integer value, with off being 0 in most cases.
     Usage: debug {<switch> <value>}
```

- AVScommand_debug 0 # Echo commands received through AVScommand and their results
- AVScomm_debug 0 # Enable module communications debug output (0 or non-zero)
- AVSfield_debug -1 # Enable field debug output levels -1, 0-4 increasing output
- AVSmessage_dialog 2 # AVSmessage level that presents dialog (0-4 = info-fatal)
- CLIrecord_LUI 0 # Enable recording of LUI button hits during script output (INCOMPLETE AND SUBJECT TO SEVERE LIMITATIONS)

- CLIscript_dialog 1 # AVSmessage (during scripts) present dialog (1)
- CLItime 0 # Display the time each CLI command takes to execute (1)
- EDITORmacro_mode 1 # System mode for implementing macros. 0 means old macros, 1 is new
- FlowConcurrent 0 # Turn on or off concurrent starting of networks with parallel forks
- FlowVerbose 0 # Make internal kernel operations report status information
- MEM_verbose 0 # Print a line for every allocate & free. 1 = within module compute function only; 2 = outside; 3 = Everywhere.
- MEM_check 0 # Fill memory on allocate or free. Also report leaks on second and subsequent module compute function calls.
- MEM_history 0 # If non-zero: Fences and history of allocations. More checking and better messages on free. 2 (bit 1) = check fence posts on every allocate and free. 4 (bit 2) = when exiting, list items allocated and never freed.
- XSynchronize 0 # Control X calls being flushed immediately (1) or not (0)

Some of the more interesting flags are AVScommand_debug, which shows what commands are coming in from modules using AVScommand; CLItime, which activates timing and reporting on individual CLI command executions; and FlowVerbose for gaining insight into what module is currently executing. The "MEM" flags will affect new module instances after they are set if the MEM package is compiled into those modules.

Writing scripts

CLI scripts are helpful for automated testing as well as reproducing specific problems. In order to write a script use the "script" command:

```
% avs -cli
avs> script -open <filename>.scr -echo yes
avs_script> ....  <do what you want>
avs_script> script -close
avs>
```

By saying "echo -yes" you will see the CLI commands being recorded going to *stdout* as well. This helps the user learn basic CLI commands and confirms what is and isn't being recorded. Most interactive operations will be recorded - Network Editor operations and module widget interactions in particular. Some additional operations will be recorded when you enable the debug flag, CLIrecord_LUI, by setting it to 1. Then when you record a script, more button hits and geometry viewer operations will be recorded. **Note:** The CLIrecord_LUI option is not supported as it is incomplete in some areas - direct interaction with the Geometry Viewer views is not recorded for example. However it can sometimes provide the "missing glue" that script writers need to perform a specific operation.

For more information on the CLI, see Chapter 5 of the *AVS Developer's Guide.*

Playing back scripts

Scripts can be played back in several ways - using the "avs -cli" command line option, the Network Editor/Help/Demos script browser, the AVS Demos pull down menus, or the "script -play" command as follows:

```
% avs -cli
avs> script -play <filename>.scr -echo yes -action user -break step
```

This last is the most useful and allows you to break at every command (*step*) or only at special points (*check*), either waiting for a user prompt or pausing for a specified amount of time. You can break at any time by hitting return in the CLI and it will ask if you wish to continue or abort the script.

In combination with other debug switches, you can slow down a prerecorded script and analyse the sequence of events taking place for each operation by using the "script -sleep" option. For more information type "help script" to the CLI to get the online help for this command.

Debugging Modules

The best way to debug modules is using the *avs_dbx* script that effectively wraps around a module and mediates with the debugger of your choice.

This tool is just a shell script that can be printed out or changed if necessary; by default it is set to use the most common debugger on a given platform. It can be run either with command line arguments to select a module or it will interactively query for the minimum information it requires. The steps for running it are as follows:

- Compile your module for debugging using either the Module Generator or by setting the "G" environment variable to the needed local debug flag and calling "make".
- Use the Module Generator "debug" option or invoke *avs_dbx* <*binary-name*>, selecting a specific module if needed with "-mod". If desired you can select a different debugger using the "-debug" option. Set any break points you need immediately.
- Instance the module in the Network Editor
- Type "run" AFTER the message "<module> instance waiting, fire when ready...".

More information on *avs_dbx* can be found in page 3-20, *AVS Developer's Guide.*

Input and Output Data

During development it can help to use test data patterns in addition to your own data sets. Test patterns can help confirm basic operations are working over a wide range of expected data. They can also help provide sample data sets showing problems with supported modules when working with AVS Customer Support.

Test input

Sample data sets can be either static - as found in *$AVS_PATH/data*, the AVS_DEMOS package, or the IAC - or dynamic, as output by test data generators like the new **test field** module (*$AVS_PATH/examples/test_field.c*) or the older FORTRAN examples, *test_field_f.f* or *test_fld2_f.f*. The new test field module provides a much more extensive set of test data patterns, dimensions, and sizes. It is described in the UNIX chapter and documented in its own man page.

Binary data file formats vary from one platform to the next and can cause serious problems between 32 and 64 bit platforms and also between big- and little-endian systems. Use "XDR" data format whenever possible to minimize these issues; see documentation on the Read Field and Write Field modules for more information.

Diagnostic output modules

Once you have run data through your module without crashing, the question is "how do you know it produced the RIGHT answer ?" There are several ways to see what came out. Note: for more information on these and any other modules, see the online man pages.

The most basic way is to see how the modules downstream respond to the output. If they crash or produce unexpected results, something is likely wrong with the data your module is producing. If you can debug the modules downstream you may see which aspect of your data sets are causing problems.

- **Print Field** - Some supported modules are provided that help you actually look at your data more directly. **Print field** in particular is a very useful tool for examining AVS fields. If attached to a field output, it will display information about the header and some or all of the data in a text browser widget (note: Use the Network Editor/Layout Editor to expand the default user interface layout for print field or use a separate text editor onto the same file name that print field is writing to. Print field allows you to take various text slices through the data by selecting different index ranges.

  What you are usually looking for using Print Field and similar modules is unusual data values, such as "NaN"s (not-a-number values) and extreme values with high exponents that are unexpected. Of course, any serious deviation from your expectations (all zeros, clipped values, etc) can give you a powerful clue to the problem.

- **Print UCD** - a similar module which outputs UCD data instead of fields.

- **Write Geom** - this new example module outputs binary data, calls *geom_to_text* to convert the data, and displays the result in a text window like print field. The source for Write Geom shows how to use the text browser widget to view a file.

- **Geometry Viewer** - The builtin Geometry Viewer module and the Geometry Viewer itself recognize a special environment variable, **AVS_GEOM_VERIFY**, as a request to print a verbose description of what they are seeing as they process a geom data type. This output will display NaN's and other oddities that may give some clue to the problem.

- **AVSGraph** - The AVSGraph module recognizes an environment variable, called **AGX_DEBUG**, to create a log file called */tmp/AVSGraph.log* of internal Toolmaster diagnostic messages. This may help you discern problems with input data or other problems.

- **Compare Field** - The compare field module serves a useful purpose when the data can be directly compared to an expected result or a similar baseline file. For example, in order to certify that similar C and FORTRAN modules produce the same output field, you would use compare field.

Debugging in AVS 5

Kernel debugging

Debug output

Another source of problems relates more to how the system is transmitting the data or running the module. AVS 5 is capable of producing many different types of diagnostic output, though some of this information may be more useful for AVS Customer Support to help diagnose your problems than for you to make sense out of it without understanding the internal architecture.

The Network Editor is the best way to see the network and which module is currently executing. It provides Verbose mode under the "Module Tools" menu, which will cause diagnostic output about which module is executing and what connections are being made.

A related AVS command line option, "-mod_time", will time how long a given module takes to run when it is being used. Another way to see how long operations are taking is to use the CLI debug flag "CLItimer", which will time individual CLI commands and send the output to *stdout*.

A number of environment variables exist that produce some tracing information to describe communications or module execution. These are set using "setenv" in the C-shell.

- AVS_COMM_DEBUG - provides trace information about communications traffic between the AVS kernel and all other modules, including coroutines. This can highlight data transmission problems. Can also be set using the CLI debug command to set the AVScomm_debug flag.

- AVS_GEOM_VERIFY - mentioned already, as a flag to enable diagnostic output from the Geometry Viewer as it processes field data input.

- AVS_FIELD_DEBUG - provides AVS field data information as fields are allocated and deallocated in local or shared memory. This can provide some idea of the size of fields moving around as they are being created. Can also be set using the CLI debug command to set the AVSfield_debug flag.

- AGX_DEBUG - mentioned already, provides a diagnostic output log from the AVSGraph module.

Runtime conditions

AVS 5 provides a number of means of handling data such as shared memory (shm) and direct module communications (dmc) that expedite communications and reduce data copies, but that may obscure your problem. These communications options can be disabled using the "-noshm" and "-nodmc" options respectively.

The **AVS_OGL_INFO** environment variable is useful for determining information about your graphics adapter and OpenGL implementations. Modules may seem to produce different output when in fact the Geometry Viewer is rendering their output differently on different machines. Set this environment variable to 1 before starting AVS then enter the Geometry Viewer to get diagnostic output about OpenGL initialization.

Another option to consider when isolating problems is disabling modules selectively to pin point which module is the more likely "culprit". Use the Network Editor/Module Tools/Disable Module option or the CLI command, "module -off/-on" to disable individual modules.

Working with AVS support

Finally if you continue to have troubles with your module or application, and are currently on maintenance for AVS 5, you should contact AVS Customer Support for additional help or information.

The biggest problem we often face is being able to reproduce your problem. It helps immensely if you can demonstrate the problem using some of our supported modules and data sets. If not, the ideal is to provide us a copy of your module or application to review the situation you are facing.

Our FTP site (ftp.avs.com) provides the best means of sending large data sets or executables to us for help in investigating your problem and is much more reliable than email for large files. Please contact AVS Customer Support BEFORE you place files on our ftp server to ensure we are expecting your files and can pull them in quickly to attach to your case records. Follow the instructions to place them on the ftp site, in the *incoming* directory.

Less effective but still useful is seeing a Postscript snapshot of your network, obtained using the Network Editor/Network Tools/Print Network option or seeing output views from the system using *xv* or the *Write Image* module.

Requesting Module Source Code

In some situations, you may feel it would be most effective to obtain the source code to some of our supported modules, in order to make minor changes or investigate issues. In most cases it is possible for AVS Support to release the source code to individual AVS5 modules. This will be considered only for customers under a valid support contract.

Before the source can be sent we must receive or have on file a signed copy of our Confidential Disclosure Agreement. In addition a signed Source Code Addendum must be sent for each module or set of modules requested. Both of these documents can be downloaded from our web site at *http://help.avs.com/AVS5/faq/modsrc.asp*. The requests should be faxed to the AVS main office in the U.S. at (781) 890-8287 or through your local distributor or support office.

Once the request has been received, it will be reviewed for approval. Certain modules will not be made available. The following lists some of the modules not available as source code, though there may be more. Most of the following are "builtin" modules that only work as part of the kernel process anyway; these might be made available to Developer AVS customers that require them.

- geometry viewer
- image viewer
- graph viewer
- display image
- isosurface
- colormap manager
- display pixmap
- render geometry
- transform pixmap
- image manager
- render manager
- ucd legend

Debugging in AVS 5