CHAPTER 5

COMMAND LANGUAGE INTERPRETER

Introduction

The AVS Command Language Interpreter (CLI) is a text language that can be used to drive most of the AVS system. It is used to save networks and widget layouts, save parameter settings, and to record basic user interaction in the form of script files for later playback. It is one approach to providing an animation capability within AVS that allows precise control of AVS network operation for long operations (batch) or fixed sequences of parameter changes.

AVS modules can also send CLI commands to the kernel to build and modify networks, modify parameter values, change rendering transformations and properties, rearrange their user interfaces, and other operations ordinarily performed by direct user interaction with AVS. This provides the basis for building application modules which employ AVS module networks in end user applications without requiring the user to use the Network Editor directly.

This appendix provides detailed information on the CLI and the conventions it uses. For most users, the most important information about the CLI is how to use it for creating and running script files to drive an AVS network without directly manipulating widgets. The section on *Writing CLI Scripts* below is a good introduction to script writing basics.

Access to the CLI

The CLI is accessed every time a network file is read or a script is run. It can also be explicitly accessed for direct user interaction or module access in several different ways.

Command Line Option

The easiest way to get into the CLI is to start up AVS with the **-cli** command line option:

```
avs -cli
```

This will result in all standard input being directed to the CLI for interpretation and all results from those commands being displayed on standard output. Hitting an immediate carriage return will result in the CLI prompt (**avs>**) appearing to show that the CLI is ready for the next command. Typing **help** will provide basic summaries of the command sets and individual commands and their usage.

The **-cli** command line option optionally takes a CLI command string so that AVS can automatically be started running a script or other operation. For example,

```
avs -cli "script -play /usr/avs/test/scripts/script1 -q"
```

would run *avs*, playback the *script1* script and automatically quit when that script was complete.

Server Option

AVS can also establish a connection to an external process that is providing the CLI commands and displaying their output. This provides an avenue for other programs to drive AVS via CLI commands.

The *avs_client* program provides a simple example of an external CLI driver and demonstrates how additional local commands can be added to the *avs_client* side to extend the command set. First, if */usr/avs/examples/avs_client* is not already compiled, run *make* in that directory to compile it. Then start up *avs* with the server option:

```
avs -server
```

and as it starts up it will display the message:

```
AVS server process is <process>, port is: <port number>
```

In another *xterm* window, run */usr/avs/examples/avs_client* giving the **<process>** value as a command line option:

```
avs_client <process>
```

The *avs_client* program will read a temporary file called */tmp/avs_server.*<*process*> to obtain the port number to hook up to the *avs* process (the **<process>** identifier can also be obtained from the CLI variable "Pid"). Then the standard input to *avs_client* will be sent to the *avs* process as CLI input and the resulting output will appear in the *avs_client* terminal window.

Module Access

Individual modules can also send CLI commands to the AVS kernel process that is running them. This allows AVS application modules to manage AVS networks in response to changes in their own parameters.

C:
#include <*avs/avs.h*>
AVScommand(*destination, command_buffer, output_buffer, error_buffer***)**

char **destination, *command_buffer, **output_buffer, **error_buffer;*

FORTRAN:
#include <*avs/avs.inc***>**
AVSCOMMAND(*DESTINATION, COMMAND_BUFFER, OUTPUT_BUFFER, ERROR_BUFFER***)**
CHARACTER *(*) *DESTINATION, COMMAND_BUFFER*
CHARACTER*<maxsize> *OUTPUT_BUFFER_BUFFER, ERROR_BUFFER*

This routine can be used to send AVS CLI commands to the kernel or other CLI receivers. The *destination* argument is currently "kernel" only.

The *command_buffer* is a buffer containing one or more CLI commands. The *output_buffer* and *error_buffer* are used to return the output and error output from executing the commands, respectively. In C, each of these two arguments are provided as the address of a char pointer which will be changed to point to the actual buffers. (i.e. declare a char* variable ("buf") and pass its address ("&buf")). Memory management for the buffers is provided automatically by **AVScommand** and the caller should NOT attempt to free these buffers directly. In FORTRAN, the buffer contents are copied into local buffer strings provided by the caller. Choose a <maxsize> for the buffer arrays that is appropriate (the size you choose is communicated automatically from FORTRAN to the AVS C routines). Extra output beyond that amount will be lost.

Multiple commands can be included in the same command buffer and should be separated by newline characters. The accumulated output and errors will be in the buffers returned with a single result for the overall operation.

When a module wishes to reference itself in a CLI command, it should use the variable reference **$Module** instead of an explicit name like "read image.user.3". This is only recognized during an **AVScommand** call.

The **debug** command provides a switch, **AVScommand_debug**, that will tell the AVS kernel to display all CLI commands being received from modules that are using the **AVScommand** function. It will also show the results and error messages that these commands are generating. For example,

```
debug AVScommand_debug 1
```

will turn the switch on; a value of 0 will turn the switch back off again. The **debug** command is not currently supported, but help may be obtained by typing "help debug".

.avsrc File Option

AVS will recognize an *.avsrc* file option to read a CLI file in as part of system initialization. The line

```
CLIinit /home/me/my_avs_cli_file
```

will execute the command "source /home/me/my_avs_cli_file" as part of initialization so that personal variables can be set as part of start up using the **var_set** command. This is more useful for defining personal variables to customize the CLI environment than for performing active operations, such as starting up CLI scripts. For those operations it is preferable to use the **-cli** option.

Basic Concepts

There are some overall concepts that one needs to understand how to use the CLI. These include lexical and syntactic conventions, naming conventions and output redirection.

Commands and Tokens

A command is made up of a sequence of *tokens*, each of which may be one or more words. Each command is usually in the form of <verb> <subject> <options>. If an inadequate number of tokens are provided to a command, it will automatically print out the usage message. For example, in the following command there are three tokens.

```
parm_set mymodule.user.2:comment "This is a comment"
```

The first token is **parm_set** which is the command to set a module parameter value. The second token is **mymodule.user.2:comment** which specifies a module (mymodule.user.2) and one of its parameters (comment). The third token is "**This is a comment**". The use of quotation marks indicates that the four words are to be treated as a single string so it is handled as a single token (the new value for the parameter).

Tokens are separated by one or more spaces, and quotation marks must be used when the spaces are part of a token. In the case where a module name contains spaces, quotes must be used to enclose the module name. For example, in the parameter name

```
"read image.user.1":"File browser"
```

the quotes are used to override the embedded spaces in both the module name and parameter name. A single set of quotes enclosing the entire token would also be valid (the colon (:) is part of the module:parameter name format).

Case Sensitivity

CLI commands are case sensitive. Command names are nearly all lower case as are all option flags. Identifiers, such as module names and aliases and parameter names, must use the same combination of upper and lower case letters that was originally defined. Pathnames must match the case used in the target file system.

Interrupting CLI execution

When the CLI is executing a script, entering carriage returns on the standard input window will get the attention of the CLI; it will then offer the option of continuing or quitting from the script. Hitting control-C to try to abort excess CLI output will exit from *avs* itself and should not be used. Individual commands (including *net_read*) can not be interrupted. If the script is being run through the Script Controller Browser, the Pause or Abort buttons should be used to interrupt the CLI.

Multiple Line Commands

Most commands readily fit on a single line. In cases where they don't a backslash (\) can be used to indicate that the command continues on the next line. The command ends when a line ends without the backslash. For example,

```
parm_set mymodule.user.2:comment \
"This is a comment"
```

is identical to the previous *parm_set* example.

Some tokens may require embedded newlines, as in the case with a comment that is several lines of text. A newline is recognized as the two characters "\n" and in combination with a backslash can permit lengthy strings to be used as in the following:

```
script_check \
"This is a longer comment \n\
than I thought would be able \n\
to fit on a single line"
```

This would result in a three line comment to the **script_check** command.

Variable References

Since some CLI tokens can become long (pathnames for instance) or can change according to context (what platform you are running on, or where to find the main AVS directory), it can be useful to replace literal tokens with variable references. The CLI recognizes variables whose values are substituted in in place of the original token (or part of a token) when the CLI command is interpreted. Variables include both those predefined by the AVS kernel and those defined by the user for convenience.

Variables are defined using the **var_set** command and their values are retrieved using the **var_get** command. For example, to create a personal variable called "home", the user can type the following:

```
var_set home /home/terry
```

The **var_get** command can then print out the current value of the variable, "terry". The **var_set** command also has an option (**-env**) to obtain a value from an external environment variable instead of a literal value. For example to create a variable with the value of the "TERM" environment variable, the following command could be given:

```
var_set term -env TERM
```

There are some predefined variables that are set by AVS and cannot be changed by the user. These include the AVS path (**Path**), AVS kernel process id (**Pid**), data directory (**DataDirectory**), and network directory (**NetworkDirectory**) and their values represent the value provided by the user from command line options or the *.avsrc* file.

The **$** character indicates a variable reference within a CLI command and will cause the remainder of the alphanumeric string to be replaced with the variable value. For example,

```
module "read image.user.0" -ex $Path/avs_library/read_image
```

will cause the reference "Path" to be replaced with the internally maintained variable referencing the **-path** command line option or the **Path** *.avsrc* option. Variable references can be contained in any part of the token; however, they will be substituted even if they are contained in quoted tokens.

Output Redirection

CLI commands recognize the **>** character as redirecting output when used at the end of the commands. For example:

```
help * >/home/davids/tmp/help.txt
```

will cause the help description to be written out to *help.txt*. They also recognize a double **>** character (**>>**) to append to existing files.

Identifiers

There are a limited variety of objects to be referenced in AVS but it is important to understand how the naming conventions work. It is easier to see what names AVS has created after an interactive session than it is to directly predict names in advance, so it is worthwhile to work with the CLI interpreter interactively to get an understanding of how these conventions work in practice.

Module Names and Aliases

Module names take the form *module***.user.***N*, where "module" is the name set by the module itself and "N" is a unique module identifier number assigned when the module is created. The number may end up being different than what appears in the original network file since that number may already be used. This is what lets us merge network files readily. Unfortunately it makes it hard to write scripts from scratch since the numbers change. When reading a network, all references are reinterpreted to use the newly assigned numbers. Subsequent files read after that can find it hard to predict what numbers will be used. A **Clear Network** operation (or **net_clear** command) resets numbers and makes this easier.

The module command provides an option to explicitly provide a unique and permanent alias for a module that can be used in place of the ".user.N" convention. After building a network file and writing it out, edit the file using any text editor and add a **-alias** clause to the module commands to specify an alias. For example,

```
module "read image.user.1" -xy 100,100 -alias ReadImage
```

will recognize references to the module as "ReadImage" as well as "read image.user.1" (the "1" is likely to be changed during module creation and the assigned name is echoed back). If the alias ReadImage is already in use, an error will be reported and the alias will be ignored.

You can also directly tell the CLI to add an alias to an existing module as in

```
avs>  module "read image.user.1" -alias ReadImage
```

Once the module has an alias applied to it, it will persist in any networks written out. Setting the alias to "" (empty string) will eliminate any alias.

Parameter Names

Parameter names are qualified by what module they are associated with. For example, in the command

```
parm_set mymodule.user.1:parm value
```

the reference to "mymodule.user.1" indicates what module the parameter belongs to and "parm" indicates which parameter it is. The colon (:) must appear to distinguish where the module name ends and the parameter name starts.

Port Names

Port names are similar to parameter names in that they are qualified by the name of the module and a colon between the module name and port name. The port name is usually an integer indicating the number of the input or output port. Port numbers use zero based indexing, so the first input port would be 0, the second input port is 1, etc. Output ports use the same numbers. Parameter ports can also use integers, starting after the input port count (if there are 3 input ports (0, 1, and 2), the first active parameter port would be 3). Parameter ports may also use the parameter name itself just like references to the parameter value. Ports can be connected to regardless of whether or not they are "visible".

Combining Networks

This section describes some advanced features that allow the CLI greater control over how networks get merged together and managed as separate subnets, options not directly available through the current user interface. All of these features are used by the **Data Viewer** module as a means of creating and managing networks to form a larger network.

Module Tags

Module *tags* are a means of grouping together related modules for easier handling using the CLI. Modules that have the same tag can be moved together, hidden from view, disabled, or destroyed with a single command. This provides support for reading in a network as a single unit and deleting it as a single unit without disturbing other modules in use. It also allows for generic reference to pieces of a network that fulfill a given function (data input, data processing, etc) when grafting in a network (see "Module Maps" below).

The **module** command has a **-tag** option which permits a tag identifier to be added to the module directly using the CLI or while being read back from a network file. A tag identifier can be any valid single string token (if spaces are enclosed, quotes are necessary). The **net_read** command also has a **-tag** option that will automatically add tags to any modules instantiated while reading the network file.

A number of commands optionally take tags to reference all modules with the given tag. The **net_clear** command, for example, will delete only those modules with the given tag (the **-not_tag** option deletes all *but* those with the given tag). If there were six different tags used and we wanted to delete three of those tagged groups, three calls to **net_clear** using the **-tag** option would selectively remove them.

Module Maps

AVS networks are easily merged using the **Read Network** (**net_read)** operation, but this in itself does not allow the networks to be directly combined. The use of module maps allows for more direct control over how the networks are grafted together to share select resources and operations.

A module map is a list of existing module instances that should be used instead of creating new instances when reading in a network file. The map is created using the **net_map** command to add or remove specific modules or tagged groups to the map or to list the current map. When the network file is read in using a specific module map, each module creation request first checks to see if there is a module with the same base name in the map.

For example, if **read image.user.1** is in the module map when AVS reads in a new network which has an instance of **read image** in it, then the mapped **read image.user.1** is connected to rather than instantiating a new module and adding widgets to it. The existing **read image** module only gets the same connections that the network file specifies; any new widget layout and parameter settings are ignored. The existing module will retain its original tag value and will not pick up a new tag from the file being read in.

A module map can also use tags to reference all modules in a group. The first module in a tagged group that is found to match a map request while reading a network file is used. (Only one module is mapped into the net in place of the one in the network file).

In a few cases, it is useful to map modules to modules that have different names but which have the same types of inputs and outputs. For example, you might want to set up a generic base network with **ucd read field** in it but you want to be able to graft in a generic network with **read field** in it instead. **Type tags** allow similar modules to be identified. These are different from the module tags described above in that they are only used in mapping operations. The module **-type_tag** option allows an additional qualifier to be added that provides another map match key. If a module is not otherwise matched and there is a module in the map with the same **-type_tag** value, then a match is made even though they have different names. In the example described above, if both the **ucd read field** module and **read field** modules had the same type tag, such as "field input", then they would match.

Pend Operations

Pend operations request that a new module be inserted into an existing network. A module can be prepended or postpended to another module, inserting itself in between the specified module and the modules that are connected to it on a particular port (the **-prepend** and **-postpend** options on the **module** command). This can be useful to provide additional filtering before or after a module without specifically referring to the connections that are being disconnected and reconnected to include the new module. When the new module is deleted or the **-unpend** option is used, the connections that were broken in the initial operation are remade between the original modules. As an example, in a network consisting of "read field.user.0" connected to "print field.user.1", the following command would create a new instance of "clamp" inserted between the two:

```
module clamp –postpend "read field.user.0:0"
```

When the "clamp" module is deleted, the original connection between the "read field" and "print field" modules is recreated.

Writing CLI Scripts

CLI **scripts** are most easily generated by recording network operations and parameter settings using the *script* command. Scripts can also be written or edited by hand or generated by a program. AVS can then be driven under CLI control to reproduce a series of operations, in effect providing a basic mode of animation. They are useful for simple animations, test suites, and demonstrations.

A script is basically very similar to a network file except that each individual step is seen by the user rather than being done all at once. It includes parameter settings that are to be done sequentially (changes are seen after each setting) rather than all at once during initialization. It also allows you to change network connectivity and control what is running.

Writing Scripts

The easiest way to write a CLI script is to let AVS write the basic script for you and then use that ASCII file as a template. This minimizes the confusion in determining which commands to use and what names to give to things.

- Start up avs with the **-cli** option

  ```
  avs -cli other options
  ```

  and enter the network editor as you normally would
- In the standard input window (the *xterm* you started from) give the following command:

  ```
  script -open <your file> -echo yes
  ```

  This will start a new file that records the CLI equivalent of the actions you perform using the mouse. It will always begin with a **Clear Network** operation so that it can start off with a clean slate. It will prompt you for verification if there is an existing network. If you want to create a script using the current network, first do a **Write Network** to save the network and its current state and then once you have begun the script, use **Read Network** to read it back in again.
- Now you can do basic network editing operations and parameter settings and they should be recorded out to your file. It should capture the following CLI commands (and echo a line out to standard output):

  make/move/delete module **(module/mod_delete)**
  Disable (enable) module **(module)**
  connect/disconnect modules **(port_disconnect/port_connect)**
  parameter changes **(parm_set)** using the modules widgets
  all except the track widget are recorded.
  Clear Network **(net_clear)**
  Read/Write Network **(net_read/net_write)**
  Disable (enable) Flow Executive (to batch changes) **(net_flow)**
  Port Visible **(port_vis)**
  All directly typed in CLI commands
  Layout Editor operations

- Many common operations *are not* recorded out to scripts. These include all operations in the Geometry Viewer, Image Viewer, and Graph Viewers. You have to edit these commands into your script manually to achieve full animation capabilities. The **-echo** option on the script command allows you to see what is and what is not recorded out to the script.
- You can add comments to your script as you go by typing

  ```
  script_check "Comment that describes what you are doing"
  ```

  These comments are echoed when the script is played back and appear in the script controller widget (if it is being used). They tell the user what should be happening and provide breakpoints that can be used when playing the script back.

- When you are done, type the following:

```
script -close
```

Quitting without closing the script will loose the last output buffer.

Playing Back Scripts

You should now be able to play the script back by typing the following command:

```
script -play <your filename>
```

All scripts generated directly by AVS will begin with a **net_clear** operation to clear away any existing network and start a session with a blank slate. It will *not* prompt you for verification at this point so be sure to save your existing work if it is important. Manually edited scripts need not have a **net_clear** operation in them if that is not desired but you need to be careful that when they are played back the network context that they expect is available. The **script** command has some options detailing how quickly it will play back the script and what it will do as it runs. It can "break" at each command or just at the **script_check** comment points; and when it breaks it can pause for a fixed number of seconds or prompt for user input to continue.

You can also take advantage of the **sh** (shell) command to run an external system command such as sleep for a given number of seconds. There is also a CLI command, **script_sleep**, which will sleep for a given number of seconds while permitting other AVS operations (coroutine modules running, user intervention, etc) to proceed at the same time.

If you wish to interrupt a script while it is playing back, you can hit the **return** key repeatedly until you see a prompt appear:

```
Script - User Interrupt (continue - c, quit - q):
```

You must then type either **c** to continue with the script or **q** to quit and return immediately to the **avs>** prompt.

Errors or warnings that normally cause a dialog box to appear will instead cause a message to be output on standard terminal output. If a choice is required, the default choice is automatically selected and the script proceeds.

The Script Controller Browser

An easier alternative to using the **script** command for playback is to use the Script Controller Browser. Hit any **Help** button (on Network Editor, etc) and on the help browser that appears there is a button labeled **Help Demos**. Hitting that button will present the Script Controller Browser showing the scripts available in */usr/avs/demo/man_scripts*. You can type in an alternate directory and then select the desired script file. Once a script starts you will get some new buttons on the browser (**Pause, Continue,** and **Abort**) that let you pause momentarily during the script, continue or stop completely. When the script is finished, the network is left in place *unless* you added a **Clear Network** command to the script (which is not necessary). If it seems to be running very slowly, type **script -sleep 0** at standard input to get it to avoid pausing after each command that is executed.

The Script Controller Browser is a **topic** browser (like the main help browsers) which means it looks in the current directory for an optional *.topics* file that has the filename and a descriptive line about it. This allows you to provide a list of what the scripts do that is more informative than "script1"; topic browsers are described in more detail in the "On-Line Help Facility" appendix.

The Script Controller Browser can also be obtained using the **script** command (using the **-interface** option).

Script Suites

There are several additional features of script playback that allow for a series of scripts to be played in succession to allow for demo loops or extensive test suites to be run.

- When playing back a script you can give the **script** command the **-quit** option which tells AVS to quit when the script is finished.
- You can start AVS giving it a CLI command to run immediately:

  ```
  avs -cli "script -play <myfile> -sleep 0 -quit"
  ```

  starts AVS, runs the script, and then exits. Without the **-quit** option, AVS will continue to run, expecting subsequent commands from standard input or other direct interaction from the user.
- You can specify multiple scripts to play in a single script command as in

  ```
  script -play <file1> <file2> <file3>
  ```

  and when the first file is done, the next will be run, etc.
- You can chain from one script to another. If a script file hits a **script** command in it, it will close itself and start that other script file instead. This feature can be used to create a demo or test loop; an initial script file can contain a **script** command which starts a series of other scripts, the last of which is the initial script filename.

Commands

Commands are grouped into related command sets. This section describes the supported CLI command sets, providing additional background information on the concepts that they use. Online help information is available for an entire set as a whole or for individual commands. Typing "help" will describe how the help command works and list the major command sets:

```
Basic CLI commands
Network editor commands
Geometry viewer commands
Graph viewer commands
Image viewer commands
User interface layout commands
Application commands
```

There are some unsupported commands which are not described here and may be subject to change in subsequent releases. General help on these commands is available using the **-unsupported** flag in the help command. Also there have been some command name changes since the AVS 2 release for greater consistency between commands. These names are still recognized for compatibility, but will not appear in general help requests unless the **-alias** flag is used. Requests for help on any individual command will work regardless of the help flag settings.

Command Usage Notation

- All commands and options are in lower case as shown in the usage messages. CLI commands are case sensitive as noted above in "Basic Concepts".
- { } indicate optional clauses or tokens
- <name> indicate tokens to be replaced with user values as opposed to literal strings
- "..." indicates repeatable phrases
- | indicates "either OR"
- / indicates similar tokens with same phrase structure as in

```
-a/b <name>     means "-a foo" or "-b foo" are allowed
```

- # indicates a comment describing an option
- ( ) indicates where an abbreviation is allowed

```
(c)lose  means that "c" alone is adequate
```

- * indicates a wildcard match for a value. Only a few specific commands recognize this value.

Basic CLI Commands

These are basic commands used to control the flow of CLI commands from files (source, script, etc.), manage variables for easier reference, and provide help and other general services.

General Commands

help — List available commands or command sets

With no arguments it will list a summary of the main command sets. A capital letter will match a command set name and list that command set. A specific command name will show a full description for that command. An asterisk (*) will show a brief summary of all commands. A name ending in an asterisk (*) will show commands starting with that name. Some commands are not supported or intended for continued use and these are only shown when special options are used (**-alias, -unsupported**) and are flagged accordingly. Except for specific command requests, a shortened description is given two flags (-full and -usage) can be used to expand help given.

```
help {-alias }        # Show commands which are aliases for other commands
   {-full}            # Show full description and usage for command sets
   {-unsupported}     # Show commands which are not yet supported
   {-usage}           # Show usage information for command sets
   { <capital letter> # Show summary of commands in a command set
   { <command> | * }  # Show full help on specific command or all commands.
```

quit — Quit from AVS with optional confirmation dialog

Ordinarily, this command will immediately quit out of AVS. The **-dialog** option will present the standard confirmation dialog box before exiting. The **-status** option will cause AVS to return an exit status number consisting of the number of Error and Fatal level messages generated during the session to support automated testing.

```
quit { -dialog} { -status}
```

sh — Send command to external sh

This executes commands in an external Unix shell. All output goes to standard output so this may not work properly in some CLI modes (module access for instance). All arguments are sent to the shell together and do not need to be quoted into a single string. However some shell conventions such as macros and output redirection are overridden by CLI command processing.

```
sh <any>
```

version — Identify avs version

This command is useful to check the current AVS version number and is also used in network files to record the version number of the AVS kernel that wrote the network file. In interactive situations, providing a version string has no effect.

```
version { <version string> }
```

Script Commands

For more information, see the section above on "Writing CLI Scripts".

script_check — Checkpoint comment for script playback

In record mode, saves script_check command to script output; in playback or normal mode, writes out <comment> to output with optional pause or break depending on script command options.

```
script_check { <comment> }
```

script — Manage script output or playback

This command is used both to create scripts (**-open** and **-close**) and to play them back (**-play**). It can also be used to present the Script Controller Browser in a specific directory (**-interface**). During playback, the default mode is to read each command, wait until the network has finished executing, and then proceed with the next command. The **-break** and **-action** options allow for AVS to pause after each command and either wait for a given number of seconds or wait until the user hits return to continue.

```
script { <operation_clause> } | { <action_option> } | { <echo_clause> }
  # If no arguments, prints current status of script operations
  # Operation clauses - first letter is sufficient
    -o(pen) <filename>      # open a script file for output
    -c(lose)                # close script output
    -p(layback) <filename> {<filename>...} # read back script file
    -i(nterface) {<dirname} / close} # open or close script controller panel
  # Action Options - first letter is sufficient
    -b(reak) { s(tep), c(heck) }  # When to break
                            # on each command (step) or script_check
    -a(ction) { p(ause), u(ser_prompt) } # What to do on break
    -s(leep) <n>    # number of seconds to delay for pause
  # Other clauses
    -e(cho) {y | n} # Echo script lines as they are read in or recorded
    -q(uit)         # quit when script finishes
```

script_sleep — Pause for number of seconds during script

This command will permit modules to run while the CLI is waiting for the given number of seconds to elapse and will allow X and other input events be processed.

```
script_sleep <seconds>  # Number of seconds to sleep (float)
```

source — Read commands from file (without pausing after each)

This command is very similar to the script command but does not wait for modules to finish executing before advancing. It should only be used when all commands need to be executed immediately, without the Flow Executive running until the file has been read in completely.

```
source { <filename> ...}
```

Variables Commands

The CLI recognizes local variables whose values are substituted in place of the original token (or part of a token) when the CLI command is interpreted. For more information on variables see the section in "Basic Concepts" above on "Variable References".

var_get — Get current value of specific CLI variables

If only one variable is given, just its value is returned. If multiple values are requested, then the name and value of the requested variables are displayed. If no variables are requested, all known variables are displayed.

```
var_get {<varname> ...}
```

var_set — Set the current value of a CLI variable

Any unknown variable is created automatically. This command cannot be used to modify system defined variables, such as DataDirectory, NetworkDirectory, Module, Path, and Pid. The value can be given directly or derived from an environment variable. If the environment variable is not found, a default value can be provided.

```
var_set <varname> <value> | {-env <env_varname> {<default>}}}
```

Network Editor Commands

These commands are used to create and delete modules, connect ports, save and set parameter values, and list out current network state information. They are presented in subgroups of related commands which handle network wide operations, manipulate modules, make connections, and modify parameters.

When typing directly at the CLI it can be difficult to know which commands are of the greatest use. It is useful to highlight certain commands that provide good information about the current state of the network and control it.

- Available Modules: The **mod_show** command is useful to find out the available modules (module palette) that can be instanced OR to find out general information about existing module instances. It will optionally list port or parameter descriptions as well but doesn't describe connections or parameter values.
- Network State: The **net_show** command describes the current network in terms of the modules it is made up of and the connections between those modules. It displays its output in terms of **module** and **port_connection** commands.
- Flow Control: The **net_flow** command is very useful to disable module computation when setting a series of parameter values when they should be treated as a batch of changes instead of incremental changes.
- Parameter Settings: The **parm_save** command will list the current values of select parameters or all parameters in the form of **parm_set** commands. It provides the best template for changing parameter values. The **mod_show** command provides some additional range information on parameters.

One of the best ways to see what the CLI commands do is to use the **script** command (**script -open <myfile> -echo yes**) which will echo out the equivalent CLI commands for most interactive network editing operations. The CLI output may include options that are not required in direct input. For example, the following session would work quite well to produce a simple network:

```
avs> module "read image" -alias ReadImage -xy 10,10
"read image.user.0"
avs> module "display image" -alias DisplayImage -xy 10,60
"display image.user.1"
avs> mod_show -all
MODULE "read image.user.0" (ReadImage) TYPE: data FLAGS: C subroutine
  OUTPUT [0] "Field Output" TYPE: "field 2D 4-vector byte"
  PARM [0] "Read Image Browser" TYPE: string  RANGE: $NULL ""
MODULE "display image.user.1" (DisplayImage) TYPE: render FLAGS: C subroutine
  INPUT [0] "Image Input" TYPE: "field 2D 4-vector byte" FLAGS:  required
  PARM [1] Magnification TYPE: choice  RANGE: "x1 x2 x4 x8 x16" " "
  PARM [2] Automag_Size TYPE: integer  RANGE: 50 1024
  PARM [3] "Maximum Image Dimension" TYPE: integer  RANGE: 100 4096
avs> port_connect ReadImage:0 DisplayImage:0
avs> sh ls /usr/avs/data/image/*.x
/usr/avs/data/image/mandrill.x  /usr/avs/data/image/avs.x
/usr/avs/data/image/marble.x
avs> parm_set ReadImage:"Read Image Browser" /usr/avs/data/image/mandrill.x
```

Network Commands

net_clear — Clear the entire network or a tagged group of modules

Without any argument it will delete all modules and their user interface and reset network editor; with an optional tag, it will only delete the modules that match that tag (the **-not_tag** option deletes those that *don't* match).

```
net_clear {-tag/not_tag <tag>}
```

net_flow — Enable or disable the flow executive, or wait for it to complete

This command controls module computation and is very useful to batch related parameter changes together so that they occur in one step rather than a number of incremental changes. **Wait** is only supported through the server communications port and not interactively; it is used to wait on the Flow Executive to finish executing the active modules. **Restart** and **restart_default** are operations that restart dead modules with the current parameters and default parameters respectively.

```
net_flow { on | off | restart | restart_default | wait }
```

net_group — Manipulate a group of modules together based on tag references

This command is used to move or hide groups of module icons in the Network Editor, and to enable or disable groups of modules, based on module tags. It is used by application modules combining several networks into one.

```
net_group {-tag <tag>} # identify a group of modules or all by default
 {-xy X,Y}     # move the modules by the given amount
 {-on/-off}    # enable or disable the modules
 {-hide/-show} # hide or show the modules in the network editor workspace
```

net_map — List or modify maps of shareable resources

A map is a list of existing modules that is used in place of new modules during a **net_read** operation and is used to build aggregate networks with shared data source modules and renderers or other shared modules. With no arguments, the existing map names are listed. With only a map name, that map's contents are listed. Other options add or delete individual modules or tagged groups to a map or clear it to empty. See the section in "Basic Concepts" above on "Module Maps".

```
net_map {<mapname> { {-add/delete <module> } | { -add_tag/delete_tag <tag> }
        | {-clear}}
```

net_read — Read network description from file

A network file consists of optional comments (#), a **version** command, and then a series of commands that define modules, make connections, set parameters, and then define the user interface layout of the network. The network is read in its entirety before any modules begin execution. It may apply a tag identifier to all resulting modules and/or selectively substitute existing modules for new ones based on a map.

```
net_read <filename> { -map <map_name>} { -xy <x,y>} {-tag <tag>} {-substack}
  -map <map_name> option substitutes shared modules from map
  -xy <x,y> option offsets module boxes positions by a given amount delta
  -tag <tag> option stamps new modules or widgets with tag
  -no_pends  option to suppress checking for pended modules during mapping
  -substack option makes file's Top Level Stack a substack named tag
```

net_show — Print out existing network connectivity

This is displayed in form of **module** and **port_connect** commands. An optional tag will show those modules in a network with that tag and connections which are output from that set of modules.

```
net_show {-tag <tag>}
```

net_write — Write network description to file

This command stores out the network connectivity, parameter settings and user interface layout as CLI commands. It is essentially equivalent to the output of the following CLI commands: **version, net_show, parm_-save -range,** and **layout -root**.

```
net_write <filename>
```

Module Commands

module — Create or modify a module instance

The module is created if it does not already exist and its name is printed out. If the module exists it is changed to match the given arguments. If the name is given without a ".user.N" prefix it is assumed to be a request to create a new module; the new name will be printed out.

The pend operations connect the module in between other modules or remove it from a previous pend operation. A tag is an added identifier that allows a module to be referenced as part of a group of related modules.

```
module <module.user.N>
  {-xy X,Y}         # location of module in network editor work space
  {-host <host_name>} # remote host name
  {-ex <module_path>} # location of module executable
  {-on/-off}        # enable or disable the module
  {-alias <module alias>} # optional unique permanent name for module
  {-parent <macro name>}    # this module is contained in the macro module
  {-macro }         # this module is a macro module
  {-unshared}       # prevents the module being mapped out during readnet
```

```
{-prepend <module:port>}  # insert module before module port
{-prepend_tag <tag>}      # insert module before tagged group
{-postpend <module:port>} # insert module after module port
{-postpend_tag <tag>}     # insert module after tagged group
{-unpend }                # remove module from pended connections

{-tag <tag>}              # set or change the module tag ("" to clear)
{-type_tag <type_tag>}    # set or change the type tag ("" to clear)
{-unshared}       # prevents the module being mapped out during readnet
```

mod_delete — Delete individual modules

This command "hammers" one or more module instances in a network; it does not remove modules from module libraries.

```
mod_delete <module> { ... }
```

mod_exec — Execute module regardless of changed inputs or parameters

If the flow executive is disabled it will just add the module to the queue for later execution when the flow executive is re-enabled

```
mod_exec <module>
```

mod_group — Select a group of modules within an area

This command performs the lasso operation within the Network Editor and is used only during script recording. With no arguments it clears the selection.

```
mod_group {<x1 y1 x2 y2>}
```

mod_lib — Module library operations

Read or write a module library, select the current library, or list the libraries that are currently loaded.

```
mod_lib { -read <pathname> } | { -select <libname> } | { -list } |
        {-write <libname> <pathname>}
```

mod_read — Read the module(s) that are in the given executable file

The modules are added to the currently selected module library palette and can subsequently be instanced using the module command.

```
mod_read <module filename> { <host name> }
```

mod_show — Display information about one or more user or system modules

This command may be used to see the ports and parameters of a module or some of its internal flags that are not displayed by the module command. If there is no module specified it will list all modules; otherwise it will list one or more modules. The **-system** option lists the modules in the module libraries that can be instanced.

The basic display for a module is to indicate its basic description:

```
MODULE <name> (<alias>) TYPE: <type> FLAGS: <various flags>
```

Using the **-inputs, -outputs, -parms,** or **-all** flags will show

```
INPUT/OUTPUT/PARM [<n>] <name> TYPE: <type> FLAGS: <flags>
```

The PARM output also lists range information where appropriate for the data type.

```
mod_show { <module> ...} {-all/parms/inputs/outputs }
  {-type data/filter/mapper/render} {-system { -host <host> }}
```

present — Present a module control panel or a viewer panel

This can pop up the module panels just like hitting the module dimple or the viewer panels like hitting the viewer pop up list. The viewer names are those that appear on the main AVS menu buttons and must be given in their entirety, i.e. "Network Editor", "Geometry Viewer", etc. Names are case sensitive. The Network Editor accepts two optional values: -closed enters the editor with the main editor window initially closed; -nobutton permanently inhibits the Display Network Editor button to prevent the user from examining the network. The **-closed** and **-nobutton** options are only valid for the "Network Editor" and control the initial appearance of the Network Editor work space.

```
present {<module>} | { -tag <tag>} | {<viewer name>} {-closed {-nobutton}}
```

Parameter Commands

parm_save — Save parameter values - all(default) or individually

This command displays current parameter settings in the form of **parm_set** commands. If no module name or parameter name is given, all current parameters are displayed. If a tag is given, only modules in the tagged group are checked.

The **-since** flag allows selective display of parameters that have changed since particular reference points; since the module began execution **(exec)**, since the module was created **(init)**, or since an arbitrary checkpoint **(check)**. Each **check** request clears the flags on the parameters requested and so updates the "checkpoint" automatically.

The **-range** option displays the parameter's minimum and maximum values when they have been changed by a module because of input data it has received. The **-oneshots** option is used to request that oneshot parameters be shown as well.

```
parm_save {-tag <tag>} {-since exec | init | check } { -range }
    { -oneshots} { <module> | <module>:<parm> ... }
```

parm_set — Set parameter value

If the value is not accepted by the parameter it is ignored and an error message is displayed. The **-range** option is only permitted when reading a network file, to restore the minimum and maximum values that the parameter originally had when the network was saved.

```
parm_set <module>:<parm> <value> { -range <minvalue> <maxvalue> }
```

Port Commands

port_connect — Connect two module ports together

```
port_connect <module_from>:<output_port> <module_to>:<input_port>
```

port_disconnect — Disconnect two modules from each other

```
port_disconnect <module_from>:<output_port> <module_to>:<input_port>
```

port_vis — Control visibility of a port

```
port_vis <module>:<portname> {-on/-off}
```

port_add — Add port to a macro module

```
port_add <module>:<portname> <type> {-in/-out}
```

port_delete — Delete port from a macro module

```
port_delete <module>:<portnumber> {-in/-out}
```

Creating Macro Modules From CLI

This section contains information on how macro modules are stored using CLI commands. The information might be useful to users who want to hand edit networks, dynamically modify networks, etc.

Macro modules are created with the **module** command using the *-macro* option. For example:

```
module foobar.user.0 -xy 100,100 -ex /tmp/foo -macro -type 1
```

The command creates a macro module that has no input or output ports. Input and output ports are created for the macro module with the **port_add** command. For example:

```
port_add -in foobar.user.0:"port name" "field 2D 4-vector byte" \
    -flags 0x2
```

This example adds a single input port to the "foobar" module. The values for the *flags* argument (in this case 0x2) should be obtained from the include file *avs.h*. In this case we are specifying that the port be **OPTIONAL** and **MULTIPLE**. This is a good default for macro ports.

Children modules are added to the macro module by adding a **-parent** option onto the **module** command that creates them. The **-parent** option shouldn't be used with existing modules as it is ignored.

Connections are made from modules outside the macro to the macro in the ordinary way. Connections are made from modules inside the macro to the ports of the macro using the **IN->...** and **OUT->...** modules. The output ports of the module called "IN-> foobar.user.0" are used to represent the input ports of the macro module. Similarly the input ports of "OUT-> foobar.user.0" are used to represent the outputs of the macro module. Here is an example command that connects the **crop** module to the input port of the macro module:

```
port_connect "IN-> foobar.user.0":0 crop.user.1:0
```

Macro Module Description File

The macro module description file (the file created when you save a macro module) is just a special case of a network file. The first macro module created in the network is assumed to be the macro that is defined in that file. When you use the **Read Module** operation on this file, this is the module whose description is generated. When you instance a macro module, the network file that describes it is simply merged into the currently active network.

Another Way to Create Macro Modules From CLI

The above commands describe how the macro module is saved either in the macro module description file or in a network file that contains the macro module. One other way to create macro modules from within a network is to reference the macro module description file. This has the advantage that the description file is used every time that the network is read, but the disadvantages that you must transport the macro file with the network, and that the network is sensitive to incompatibile changes made in the module description file. For example, if you delete a widget or a port in the module description file, the network file will still contain references to the removed widget or port and will generate error messages.

```
module foobar.user.0 -ex /tmp/foo
```

When this command is encountered in a network file, the macro module description file, */tmp/foo* in this case, is used to start the module, just like any other module. Also like any other module, this module is loaded into the current module library.

Geometry Viewer Commands

These commands control the state of the Geometry Viewer and manage changes to transformation matrices and properties of objects, cameras, lights, and texture maps. There are several different situations in which a user might want to use the Geometry Viewer CLI:

- As a command line interface to obtain Geometry Viewer functionality either for more precision, to script or batch operations, or to access functionality not available through the Geometry Viewer interface. In particular, there is functionality for saving Geometry Viewer scenes in PostScript that is not available through the user interface.
- To edit saved networks, scenes or object script files.
- To control the Geometry Viewer application from a module using the **AVScommand** function. The Geometry Viewer CLI provides a slightly larger set of operations than using the *geom* data type and also provides 2-way communication. Using the CLI, the programmer can query as well as set properties of objects. The **Animator** module, for example, uses the Geometry Viewer CLI to control animations in the Geometry Viewer.

An important note before we begin discussing the details of the geometry viewer CLI: unlike many other operations in AVS, the Geometry Viewer CLI commands are meant to be batched. Almost no commands directly cause the scene to be refreshed themselves. Instead, you must explicitly refresh the scene when you want to see the results of your changes with the **geom_refresh** call (see below).

Geometry CLI State

It is often the case that the user will want to use Geometry Viewer commands without affecting the user visible state. Each stream of Geometry Viewer CLI commands, therefore, has its own state including a current object stack and a current scene. Commands that the user types at the CLI prompt will not affect the state of a script being played back or the state of command received from a module.

By default, the CLI state corresponds to the state that the user sees. Unless the user sets it explicitly, the current CLI object will always be the same as the user's current object. If you are typing commands at the prompt, this may very well be the behavior that you desire. If, however, you are generating a CLI script or executing commands from a module, you may not want to change the properties of a scene or object in a more predictable way. The current user's state should not affect the objects that you modify in this case.

If you want to control the state yourself, you should first set the current CLI scene. If you don't set the scene, it will default to choosing whatever scene the user has currently selected. You can set the current scene either by creating a new scene with the command **geom_create_scene** or by setting an existing scene with **geom_set_scene**.

geom_create_scene— Creates a new scene

The new scene becomes the current CLI scene for further CLI commands. At the time of creating the scene, you can specify the name of the scene and/or the name of the default camera. These names are useful to refer to the view or scene in later CLI commands.

```
geom_create_scene [<X Y W H>] [-view <name>] [-scene <scene name>]
```

geom_set_scene— Sets the current CLI scene to an existing scene

To set the current CLI scene to an existing scene requires knowing the name of a **geometry viewer** or **render geometry** module, the name of the scene, or you can set the scene to the user's current scene with:

```
geom_set_scene [-module <geom module name>] [-scene <scene name>]
   or with no arguments for the current scene
```

Once you have selected the scene you are operating on, many commands require you to specify an object to operate on. Particularly when generating scripts it is convenient to deal with the notion of a current object rather than having to specify the object for each command. Since objects in the geometry viewer are hierarchical, it is most convenient to have the notion of a current object stack. Unless you are dealing with hierarchies (or editing a script that does), you don't need to concern yourself with the object stack. You simply change the current object by setting the object on the top of the stack. The command to control the current CLI object is:

geom_set_cur_cli_obj— sets the current CLI object

When issued with no arguments this command sets the current CLI object to the same object as the current UI object. If the **-push** argument is used, it pushes the current object onto the top of the stack (moving the previous top of the stack to the second and so on). The **-push** argument does not change the current CLI object. If the **-pop** argument is used, the object on the top of the stack becomes the current CLI object. If an object name is specified, the current CLI object is replaced and the stack is unaffected.

```
geom_set_cur_cli_obj [-push] [-pop] [object name]
```

Beware that this command is different than the command **geom_set_cur_obj** which sets the user's notion of the current object.

Saving/Restoring Scenes and Objects

When a user saves a scene with the **Save Scene** button or saves an object with the **Save Object** button, the state of the scene or object is saved as Geometry Viewer CLI commands (usually referencing one or more *geom* files to obtain the raw geometry information). These files will contain all of the necessary commands to reproduce the state of the saved object or scene. The only difference in the file formats used by **Save Object** and **Save Scene** is the commands that are saved. For example, the first line of a file created by **Save Scene** is **geom_create_scene**.

The files saved by the Geometry Viewer commands are just CLI command scripts. The Geometry Viewer treats them just like other normal CLI command scripts that might contain network editor commands, etc. In particular, the **Read Object** and **Read Scene** commands do exactly the same thing when given a file with the *.scr* suffix except for the fact that the **Read Object** command will create a new scene if there is not an existing current scene, whereas the **Read Scene** command assumes that the script will create a scene.

Because Geometry Viewer CLI scripts are just lists of commands, users can create scripts that perform other operations that do not necessarily create objects or scenes. Scripts can set up different lighting environments, control object rotations and viewing environments, or execute geometry animations.

Geometry CLI Versus *.obj* and *.scene* Files

AVS supports two additional native file formats *.obj* and *.scene*. Any file with these suffixes are interpreted to be of these formats which were used to store object and scene files in previous versions of AVS. These file formats can still be generated by AVS by setting the environment variable: **AVS_GEOM_WRITE_V30**. The CLI format for storing geometry commands is more general than these other formats because it allows for editing of objects, cameras and lights rather than simply creating objects cameras and lights.

Saving Network Geometry State

When a network containing a **geometry viewer** or **render geometry** module is saved, the internal state of those modules are saved as Geometry Viewer CLI commands. These geometry CLI commands are very similar to those created when the **Save Scene** button is used. There is one significant difference. When the user saves a file with **Save Scene**, if the geometry description for a particular object is not know to be currently saved in a file, the user is prompted for a file to save this geometry. When saving geometry state as part of a network though, it is assumed that the modules in the network will regenerate all of the geometry necessary to reproduce the scene. This may not always be the case. If the current settings of the module's parameters will not regenerate all of the geometry, the user will have to save each of these objects' geometry by hand using the **Save Object** button in the Geometry Viewer.

Naming Objects, Cameras and Lights

Many commands operate on an object. In such cases, the object can either be specified with the **-object <name>** option. If left unspecified, the object will choose the current CLI object as mentioned in the above section **Geometry CLI State**. Object names, by default, are not given any suffix. For example, if the object was generated by a module, it will have the appendix ".<n>" where the number <n> depended on the module instance. To specify the particular instance of the object, the user would have to type the ".<n>" suffix in addition to the root of the object name.

If you are dealing with objects generated by a module, it is sometimes convenient to have this suffix automatically appended. You can do this by setting the "name context" to the name of the module that is generating the geometry. This is done with the command:

geom_set_name_context— Sets the name context

If given a <module name> argument, it will set the name context to the name specified. Otherwise it unsets the name context meaning that, in the future, names will no longer be modified.

```
geom_set_name_context [<module name>]
```

Some commands allow the specification of a particular camera. There are two ways that cameras are named, either sequentially as: "-camera 1", "-camera 2", etc. where "-camera 1" is the first camera in the current list of cameras or absolutely with the specific camera name as "-view <name>". Note that the numbered mechanism may cause problems if you delete cameras. If you delete "camera 1", then "camera 2" becomes "camera 1". The second method requires knowing the name of the camera which may not be appropriate for all situations.

Lights are referred to by number, e.g. "-light 1".

Some commands allow the specification of an object or a camera or a light. In these cases, if you specify no argument, the default is to choose the current object. You must specify an option to choose a light or a camera.

Matrix Operations

geom_get_matrix — Returns the transformation for an object, camera, or light

The transformation is a 4x4 matrix.

```
geom_get_matrix {-object <name> } { -camera {1-n}/-view <name> } { -light {1-n} }
 { -absolute } (gives redirected object's matrix if applicable)
```

geom_set_matrix — Sets a transformation for an object, camera or light

```
geom_set_matrix {-object <name> } { -camera {1-n}/-view <name> } { -light {1-n} }
   { -mat <4x4 mat> -rx,-ry,-rz <angle> -tx,-ty,-tz <val>
     -sx,-sy,-sz,-sxyz <val> }
```

geom_concat_matrix — Appends a transformation to an object or camera

```
geom_concat_matrix {-object <name> } { -camera {1-n}/-view <name> } { -light {1-n} }
    { -mat <4x4 mat> -rx,-ry,-rz <angle> -tx,-ty,-tz <val>
      -sx,-sy,-sz,-sxyz <val> }
```

geom_set_transformable — Sets the transform to object, light, camera or map

```
geom_set_transformable { object | light | camera | map
```

Global Object Commands

geom_set_bounding_box — Turns the object bounding box feature on

```
geom_set_bounding_box { 1 (on) | 0 (off) }
```

geom_normalize — Normalizes the specified object

```
geom_normalize {-object <name> }  <camera name>
```

geom_reset — Resets the specified object

```
geom_reset {-object <name> } { -camera {1-n}/-view <name> } { -light {1-n} }
```

geom_refresh — Refreshes the current CLI scene

```
geom_refresh
```

geom_get_center — Returns the center of the object

```
geom_get_center {-object <name> }  <object name>
```

geom_set_center — Sets the center for rotation and scaling of an object

```
geom_set_center {-object <name> } { -camera {1-n}/-view <name> } { -light {1-n} }
    <X Y Z>
```

geom_set_position — Sets the position of a camera, light or object

```
geom_set_position {-object <name> } { -camera {1-n}/-view <name> } { -light {1-n} }
         <X Y Z>
```

geom_get_extents — Returns extent information for an object

The extent consists of the xmin, xmax, ymin, ymax, zmin and zmax of the object.

```
geom_get_extents {-object <name> } <object name>
```

Browser Commands

geom_show_prop_editor — Raises the property editor

```
geom_show_prop_editor
```

geom_show_texture_editor — Raises the texture editor

```
geom_show_texture_editor
```

geom_show_object_info — Raises the object info window

```
geom_show_object_info
```

geom_show_object_list — Display the browser of object names

```
geom_show_object_list
```

geom_show_camera_ice— Display the popup camera ICE

Popup the camera color editor widget.

```
geom_show_camera_ice
```

geom_show_lights_ice— Display the popup lights ICE

Popup the lights color editor widget.

```
geom_show_lights_ice
```

geom_show_labels_ice— Display the popup labels ICE

Popup the labels color editor widget.

```
geom_show_labels_ice
```

Object Commands

geom_get_cur_obj_name — Returns the name of the user's current object

```
geom_get_cur_obj_name {-object <name> }
```

geom_set_cur_obj — Changes the user's idea of the current object to the object named

```
geom_set_cur_obj {-object <name> }
```

geom_set_cur_cli_obj — Changes the cli's current object to the object specified

or pushes this name onto the stack or pops the name off of the stack.

```
geom_set_cur_cli_obj [name] [-push] [-pop]
```

geom_get_all_obj_names — Returns the names of all the objects in a scene

This function returns a list of the object names in the current scene. A newline character is the delimiter between two object names.

```
geom_get_all_obj_names {-object <name> }
```

geom_lookup_obj_names — Returns object names associated with a module

Only returns the names of objects in the current scene.

```
geom_lookup_obj_names <module name>
```

geom_create_obj — Creates an object

The created object will initially have no geometry. If the **-unique** option is used, the name will append a suffix to ensure that the object does not exist. This option is useful if you are trying to create a script that can be read in multiple times to create different instances of the same object description. The [-mod <module>] will generate the name of the object as though it had been created by the specified module. A ".<n>" suffix will be appended to the name.

```
geom_create_obj {-object <name> } <object name> [-mod <module>] [-unique]
```

geom_read_geometry— Add geometry to an existing object from a file

This commands reads the geometry from the *.geom* file specified and adds this geometry to the current CLI object (or the object specified with the **-object** option). If the **-replace** option is used, any original geometry that the object had will be discarded. Otherwise, the geometry is added to the object. If the **-subset** option is used, only the geometry objects with the specified group name will be added to the object. See the documentation on the routine **GEOMset_object_group()** for information for attaching a group name to a **GEOMobj**.

```
geom_read_geometry {-object <name> } [ <filename> ] [-subset <name>] [-replace]
```

geom_read_obj — Reads an object from a geometry file

```
geom_read_obj <filename.geom> -name <object name>
```

geom_save_obj — Saves all the current object's geometries

```
geom_save_obj <filename>
```

geom_delete_obj — Deletes the named object

```
geom_delete_obj {-object <name> }
```

geom_set_trans_mode — Sets the object's transform mode

See the documentation for the GEOM libraries for more information on the transform modes.

```
geom_set_trans_mode {-object <name> }
   { redirect | normal | notify | parent | ignore }
```

geom_set_select_mode — Sets the object's selection mode

See the documentation for the GEOM libraries for more information on the selection modes.

```
geom_set_select_mode {-object <name> } { normal | notify | parent | ignore }
```

geom_get_visibility — Returns the visibility of the object

```
geom_get_visibility {-object <name> }
```

geom_set_visibility — Sets the visibility/state of an object

```
geom_set_visibility {-object <name> }
    { 0 (invisible) | 1 (visible) | -1 (delete)
```

geom_get_color — Returns the color of the object
or the background color of the camera or the light color.

```
geom_get_color {-object <name> } { -camera {1-n}/-view <name } { -light {1-n} }
```

geom_set_color — Sets the color of an object, light or camera
Sets the object or light color or background color of a camera.

```
geom_set_color {-object <name> } { -camera {1-n}/-view <name } { -light {1-n} }
    <R G B> (all numbers between 0 and 1)
```

geom_get_properties — Returns the properties of the object

```
geom_get_properties {-object <name> }
```

geom_set_properties — Sets the material properties of an object

```
geom_set_properties {-object <name> }
    [-amb <val> -diff <val> -spec <val> -exp <val> -trans <val>
     -spec_col <R G B>]
```

geom_set_parent — Sets the parent for a given object

```
geom_set_parent {-object <name> } [-mod <module> ] parent_name
```

geom_set_texture — Sets the texture for a given object

```
geom_set_texture {-object <name> } file name | dynamic
```

geom_delete_texture — Deletes the current object's texture map

```
geom_delete_texture
```

geom_set_UV_map — Sets the current object's UV texture map

```
geom_set_UV_map
```

geom_set_texture_map — Sets the texture map for a given object

```
geom_set_texture_map {-object <name> } {sphere | plane}
```

geom_show_map — Shows the current object's UV texture map

```
geom_show_map < 1 (on) | 0 (off)>
```

geom_get_render_mode — Returns the render mode of the object given

```
geom_get_render_mode {-object <name> }
```

geom_set_render_mode — Sets the render mode for the object

```
geom_set_render_mode {-object <name> }
    {lines | gouraud | flat | smooth_lines | points | phong}
```

geom_set_backface_cull — Turns the backface cull feature on

```
geom_set_backface_cull {-obj <name>} [ normal | back | front | flip | inherit ]
```

geom_set_subdiv — Sets the subdivision level for spheres

The subdivision value is the same as the value that is displayed on the menu.

```
geom_set_subdiv { -object <name> } <subdiv val: number from 1 -> 8>
```

geom_set_clip— Sets the clip plane state for the specified object

The Geometry Viewer provides the capability to associate a "clip plane" with a specific object. The orientation of the clip plane is defined by the transformation of the object. Such a "clip object" can be used to clip any object in the scene (including itself). This clipping relationship is set up with the **geom_set_clip** command.

This command takes two objects as arguments. The current CLI object (or the object specified by the -object option) is the object to be clipped. The object specified as the argument "**<**clip object name**>**" in the usage below is the object to clip the other object (called the "clip object"). The clip object can either clip the other object to the inside or outside. If the inherit state is selected, the other object will inherit the clip attribute for this clip object. If the **ignore** option is selected, this other object specifically won't clip against this object (this would be used to override a inside or outside state set to the parent of the other object).

```
geom_set_clip {-object <name> } <clip object name>
          -state <inside | outside | inherit | ignore>
```

geom_set_name_context — Takes a string (usually a module name) to use to create unique names

```
geom_set_name_context <module name> or no arguments to reset the context
```

Light Commands

geom_set_light — Sets the color, type and state of a light

```
geom_set_light   { -light {1-n} }
     -color <R G B>  -type <type name> -state <boolean>
```

geom_get_light — Returns the properties of the light, color, state and type

```
geom_get_light   { -light {1-n} }
```

geom_show_lights — Shows the position of the current light

```
geom_show_lights < 1 (on) | 0 (off) >
```

Camera Commands

geom_set_scene — Sets the scene to operate on

```
geom_set_scene [-module <geom module name>] [-scene <scene name>]
    or nothing for the current scene
```

geom_set_obj_window — Sets the window range for an object for normalization

```
geom_set_obj_window {-object <name> } <xmin> <xmax> <ymin> <ymax> <zmin> <zmax>
```

geom_create_scene — Creates a new scene

```
geom_create_scene  <X Y W H> [-view <name>] [-scene <name>]
```

geom_create_camera — Creates a new camera

Creates a new camera. If a position and size are given, they are used to determine the position and size of the window otherwise a default size and position are chosen. If the **-view** option is given, it determines a camera name that can be used in subsequent camera operations.

```
geom_create_camera <X Y W H> [-view <name>]
```

geom_delete_camera — Deletes the specified camera

```
geom_delete_camera {-camera {1-n}/-view <name>}
```

geom_read_scene — Reads a scene from a scene file

```
geom_read_scene <filename.scene>
```

geom_save_scene — Saves all the current scene

```
geom_save_scene <filename>
```

geom_set_renderer — Changes the renderer used to render a camera

The renderer name should be one of the renderers that is supported on your system. The renderer name is the name that appears in the menu, e.g. "Software Renderer".

```
geom_set_renderer { -camera {1-n}/-view <name> }  <renderer name>
```

geom_set_freeze_camera — Turns the freeze camera feature on

```
geom_set_freeze_camera {-camera {1-n}/-view <name>} < 1 (on) | 0 (off) >
```

geom_get_view_modes — Returns the viewing modes of the current view

```
geom_get_view_modes
```

geom_set_view_modes — Sets the viewing modes for the current scene

```
geom_set_view_modes  {<-depth_cue 1|0> | <-z_buffer 1|0> | <-shadows 1|0> |
    <-global_anti_alias 1|0> | <-perspective 1|0> | <-accelerate 1|0> |
    <-axes_for_scene 1|0> | <-front/back_clipping 1|0> | <-double_buffer 1|0>}
```

geom_show_camera— Sets the visibility of the camera specified

```
geom_show_camera { -camera {1-n}/-view <name> } < 1 (show) | 0 (hide) >
```

geom_resize_camera— Changes the size of the window of the camera

```
geom_resize_camera { -camera {1-n}/-view <name> } <width> <height>
```

geom_set_background— Sets the background color for the current scene

```
geom_set_background <R G B>
```

geom_set_camera_params— Sets camera orientation and position

This command sets the parameters that control the camera orientation and projection. These parameters are: "from" (the from point), "up" (the up direction), "at" (a point you are looking at), "wsize" (the window size), "fov" (field of view angle), "front" (the distance from the ’from’ point to the front clipping plane), "back" (the distance from the ’from’ point to the back clipping plane).

```
geom_set_camera_params  { -camera {1-n}/-view <name> }
     -from <X> <Y> <Z> -up <VX> <VY> <VZ> -at <X> <Y> <Z>
     -wsize <S> -fov <A> -front <D> -back <D>
```

geom_save_postscript— Saves the specified camera in a PostScript file

This command implements a feature that is not available elsewhere in the Geometry Viewer user interface. It has some limitations but provides a significant piece of functionality. With this command, the user can save a particular view as a PostScript file. Unlike the **image to PostScript** module, this command outputs lines and text objects as PostScript objects. This increases the resolution of the primitives from the resolution of the screen image to the resolution of the printer which is generally much higher.

The PostScript language does not provide primitives for doing shading or hidden surface removal. It is best, therefore, to use the image operation when significant surface information is present. This command attempts to combine both techniques—using an image to represent surface objects and using native PostScript primitives to represent lines and labels.

If this command does not perform the function adequately, the **Software Renderer** functionality allows you to generate images that are larger than screen resolution. This is prohibitively slow in many cases however.

NOTE: To use this command, you must enable the **Software Renderer** on the specified view or an error will be generated.

The default behavior of this command is to output only the line primitives encountered in the scene. Colors and depth cueing affects are ignored as is the background color of the scene. This will generally produce a picture with black lines on a white background.

Additional options are:

- -lw **<**val**>** — set the line width of the lines to use. The value is in device coordinate space (1/300th of an inch) for an ordinary laser printer.
- -grey — use the greyscale color model. Take the luminence of the objects and use this as a color. The background color of the window will be used in this case as well. If lines are vertex colored, the average of the two colors will be used to draw a solid colored line.

- -color — use the color of the objects to produce a color PostScript file. The background color of the window is used. If lines are vertex colored, the average of the two colors will be used to draw a solid colored line.
- -image — include an image that represents the surfaces. Line primitives will be overlayed on top of this image.
- -zbuffer — attempt to perform hidden line removal with the line primitives. Note that the line removal is generated at the resolution of the image window. Increasing the size of the window will produce a more accurate result even if this is used without the **-image** option.
- -land — output the image in landscape mode where the horizontal dimension of the image maps to the vertical dimension of the page.
- -eps — output encapsulated PostScript.
- -pw <page width> — set the width of the page in inches (default is 7.5).
- -ph <page height — set the height of the page in inches (default is 10.5).

```
geom_save_postscript  { -camera {1-n}/-view <name> }
   -lw <line width> -zbuffer (zbuffer lines)
   -image (save non-line/labels)
   -color -grey (as opposed to b+w)
   -eps (encapsulated) -land (landscape)
   -pw <page width> -ph <page height>
```

Action Commands

geom_cycle_store — Sets the state of the Store Frames flag for cycles

```
geom_cycle_store  < 0 (off) | 1 (on) >
```

geom_append_frame — Appends the current object to the current cycle

```
geom_append_frame
```

geom_delete_frame — Deletes the current object from the current cycle

```
geom_delete_frame
```

geom_cycle_direction — Sets the direction of cycles

```
geom_cycle_direction  0 (backwards) | 1 (forwards)
```

geom_cycle_motion — Sets the motion of cycles

```
geom_cycle_motion 0 (off) | 1 (continuous) | 2 (bounce)
```

geom_set_cycle — Makes the specified object a cycle

This command takes an existing object that is a parent object and turns it in a cycle (or flipbook) of geometries. A flipbook is like a group of objects except that only one object in the group is visible at a given time. To create a flipbook from the CLI, first create a parent object, then create your individual frames as children of this parent. Before displaying the result, you set the "cycle" property of the parent object and it becomes a flipbook.

```
geom_set_cycle {-object <name>} <0 (no cycle) 1 (cycle)>
```

Image Viewer Commands

These commands control the state of the Image Viewer and manage changes to transformation matrices and properties of images, subimages, views, and other information. The easiest way to get started writing Image Viewer commands is to use AVS interactively to generate a complex scene in the Image Viewer and to then save it out with **Save Scene**. The resulting *.ims* scene file is written in Image Viewer CLI commands. Also see the directory */usr/avs/demo/image_viewer* for some Image Viewer scripts provided with the AVS distribution. For more general information, see the *AVS User's Guide* chapter on "The Image Viewer Subsystem".

Scene Commands

image_read_scene — Reads a scene from a scene file

```
image_read_scene <filename>
```

image_save_scene — Save a scene to a scene file

```
image_save_scene  <filename>
```

image_create_scene — Create a new scene with scene location and size

```
image_create_scene  <xlocation ylocation width height>
```

```
image_show_image_list — Display the scrolling list of image names
```

```
image_show_image_list
```

image_show_view_ice — Display the Views popup ICE

This command pops up the Viewport Background Color control widget.

```
image_show_view_ice
```

image_show_label_ice — Display the Labels popup ICE

This command pops up the label Color Editor widget.

```
image_show_view_ice
```

View Commands

image_create_view — Create a new view with view location and size

```
image_create_view  <xlocation ylocation width height>
```

image_delete_view — Delete a view

```
image_delete_view
```

image_set_view_size — Sets the position and size of a view

```
image_set_view_size  <xlocation ylocation width height>
```

image_get_view_size — Returns the size of a view

```
image_get_view_size
```

image_set_view_transformation — Sets a transformation for a view

```
image_set_view_transformation  { -tx,-ty <val> -sx,-sy,-sxy <val> }
```

image_get_view_transformation — Returns the transformation of the image or view

```
image_get_view_transformation
```

image_set_color — Sets the background color of a view

```
image_set_color  <R G B> (all numbers between 0 and 1)
```

image_get_color — Returns the background color of the view

```
image_get_color
```

Image Commands

image_create_image — Creates a new image without any data associated with the image

```
image_create_image { -image <name> }
```

image_read_image — Reads an image from an image file

```
image_read_image { -image <name> } <filename>
```

image_write_image — Writes an image to an image file

```
image_write_image { -image <name> }  <filename>
```

image_duplicate_image — Duplicate an image

```
image_duplicate_image { -image <name> }
```

image_delete_image — Delete an image

```
image_delete_image { -image <name> }
```

image_reset — Resets an image position to the initial setting

```
image_reset { -image <name> }
```

image_normalize — Normalizes an image position to the current view

```
image_normalize { -image <name> }
```

image_set_image_transformation — Sets a transformation for an image

```
image_set_image_transformation { -image <name> }
    { -tx,-ty <val> -sx,-sy,-sxy <val> }
```

image_get_image_transformation — Returns the transformation of the image or view

```
image_get_image_transformation { -image <name> }
```

image_set_visibility — Sets the visibility/state of an image

```
image_set_visibility { -image <name> }
    0 (invisible) | 1 (visible) | -1 (delete)
```

image_get_visibility — Returns the visibility of the image

```
image_get_visibility { -image <name> }
```

image_raise_image — Raise the image

```
image_raise_image { -image <name> }
```

image_lower_image — Lower the image

```
image_lower_image { -image <name> }
```

image_zoom_in — Zoom in the image

```
image_zoom_in { -image <name> }
```

image_zoom_out — Zoom out the image

```
image_zoom_out { -image <name> }
```

image_set_scale_control — Sets the scale control buttons (1 = on, 0 = off)

```
image_set_scale_control  <scalex scaley>
```

image_set_bounding_box — Turn the bounding box on or off (1 = on, 0 = off)

```
image_set_bounding_box
```

Image Processing Technique Commands

image_read_technique — Reads in an image processing technique

```
image_read_technique  <name of technique>
```

image_set_technique_position — Sets the position and size of the image processing window

```
image_set_technique_position  { -tx <val> ty <val> -sx <val> sy <val> }
```

image_zoom_to_image — Applies the current image processing technique to the entire image

```
image_zoom_to_image
```

image_set_technique_window — Sets whether the current image processing technique will be In Place or New Window

```
image_set_technique_window  0 (New Window) | 1 (In Place)
```

image_set_current_image — Store the current viewed image into the image data area

```
image_set_current_image
```

image_restore_current_image — Restores the current viewed image to the original image

```
image_restore_current_image
```

Label Commands

image_label_name — Create a new label for the image

```
image_label_name { -image <name> } { -label <name> }
```

image_label_transformation — Sets a transformation for a label

```
image_label_transformation { -image <name> } { -label <name> }
    { -tx,-ty <val> }
```

image_get_label_transformation — Returns the transformation for a label

```
image_get_label_transformation { -image <name> } { -label <name> }
    <xtran ytran>
```

image_label_color — Sets the color of a label

```
image_label_color { -image <name> } { -label <name> }
    <R G B> (all numbers between 0 and 1)
```

image_get_label_color — Returns the color of the label

```
image_get_label_color { -image <name> } { -label <name> }  <R G B>
```

image_label_height — Sets the height of a label

```
image_label_height { -image <name> } { -label <name> } (a number between 0 and
1)
```

image_get_label_height — Returns the height of the label

```
image_get_label_height { -image <name> } { -label <name> }
```

image_label_attributes — Sets the display properties of a label

```
image_label_attributes { -image <name> } { -label <name> }
    [-justify <val> -bold <val> -italic <val> -font_num <val> ]
```

image_get_label_attributes — Returns the attributes of the label

```
image_get_label_attributes { -image <name> } { -label <name> }
    <justify bold italic font_num>
```

Cycle Commands

image_read_cycle — Reads a cycle from a cycle file

```
image_read_cycle  <filename>
```

image_save_cycle — Save a cycle to a cycle file

```
image_save_cycle  <filename>
```

image_cycle_read_data — Reads data from an image file (.x) and puts it in the cycle

```
image_cycle_read_data { -image <name> } <filename>
```

image_append_frame — Appends the current image to the current cycle

```
image_append_frame { -image <name> }
```

image_delete_frame — Deletes the current image from the current cycle

```
image_delete_frame { -image <name> }
```

image_cycle_store — Sets the state of the Store Frames flag for cycles

```
image_cycle_store { -image <name> } 0 (off) | 1 (on)
```

image_cycle_direction — Sets the direction of cycles and moves one image in the new direction

```
image_cycle_direction { -image <name> }  0 (backwards) | 1 (forwards)
```

image_cycle_motion — Sets the motion of cycles

```
image_cycle_motion { -image <name> } 0 (off) | 1 (continuous) | 2 (bounce)
```

image_cycle_speed — Sets the replay speed of cycles

```
image_cycle_speed { -image <name> }  <0 -> 30>
```

Graph Viewer Commands

These commands control the state of the Graph Viewer. For more general information, see the *AVS User's Guide* chapter on "The Graph Viewer Subsystem".

Reading Plot Data

graph_read_ascii_data — Reads in ASCII data file

```
graph_read_ascii_data <filename.dat>
```

graph_read_plot_file — Reads in AVS Plot data file

```
graph_read_plot_file <filename.plt>
```

graph_read_field_data — Reads in AVS Field data file

```
graph_read_field_data <filename.fld>
```

graph_read_ximage — Reads in X Image file

```
graph_read_ximage <filename.x>
```

Modes for Reading Data

graph_set_data_format — Sets the current data format mode

```
graph_set_data_format 0 (One Column) | 1 (Two Column) | 2 (Multi Column)
```

graph_set_one_column — Sets the state of one column input

```
graph_set_one_column X Axis Intervals, Y Axis Column, Color Column
```

graph_set_two_column — Sets the state of two column input

```
graph_set_two_column X Axis Column, Y Axis Column, Color Column
```

graph_set_multi_column — Sets the state of multi column input

```
graph_set_multi_column Color Selection, Level Selection
       (0 (Auto) | 1 (Value) | 2 (User)), Levels, Start Range, End Range
```

graph_set_contour_levels — Sets individual contour levels

```
graph_set_contour_levels <level1 level2 ... leveln>
```

Writing Plot Data

graph_write_ascii_data — Writes out ASCII data file

```
graph_write_ascii_data <filename.dat>
```

graph_write_plot_file — Writes out AVS Plot data file

```
graph_write_plot_file <filename.plt>
```

graph_write_postscript — Writes out PostScript

```
graph_write_postscript <filename.ps>
```

graph_write_geometry — Writes out AVS Geometry file

```
graph_write_geometry <filename.geom>
```

graph_output_image — Send an image to the Graph Viewer module's output port

```
graph_output_image
```

General Plotting

graph_set_plot_mode — Sets the current plot mode

```
graph_set_plot_mode  0 (Replace) | 1 (Add to) | 2 (Create New)
```

graph_normalize_plot_data — Sets the state of plot data normalization

```
graph_normalize_plot_data 1 (normalize on)  0 (normalize off)
```

graph_delete_plot_window — Deletes the current plot window

```
graph_delete_plot_window
```

graph_delete_plot_dataset — Deletes the current plot dataset

```
graph_delete_plot_dataset { -graph <number> }
```

graph_set_plot_size — Sets the size of the plot window

```
graph_set_plot_size x y
```

Titles and Labels

graph_set_plot_title — Sets the title for the current plot

```
graph_set_plot_title <title>
```

graph_set_title_info — Sets the state of the plot title

```
graph_set_title_info Font(0-5), Bold(0|1), Italic(0|1), Drop Shdw(0|1),
      Height(0.0-1.0), Position(0|1|2), r(0.0-1.0), g(0.0-1.0), b(0.0-1.0)
```

graph_set_plot_xlabel — Sets the x axis label for the current plot

```
graph_set_plot_xlabel <x axis label>
```

graph_set_xlabel_info — Sets the state of the plot x axis label

```
graph_set_xlabel_info
      Font(0-5), Bold(0|1), Italic(0|1), Drop Shdw(0|1), Height(0.0-1.0),
      Position(0|1|2), r(0.0-1.0), g(0.0-1.0), b(0.0-1.0)
```

graph_set_plot_ylabel — Sets the y axis label for the current plot

```
graph_set_plot_ylabel <y axis label>
```

graph_set_ylabel_info — Sets the state of the plot y axis label

```
graph_set_ylabel_info
      Font(0-5), Bold(0|1), Italic(0|1), Drop Shdw(0|1), Height(0.0-1.0),
      Position(0|1|2), r(0.0-1.0), g(0.0-1.0), b(0.0-1.0)
```

graph_set_ticlabel_info — Sets the state of the plot tic label

```
graph_set_ticlabel_info
      Font(0-5), Bold(0|1), Italic(0|1), Drop Shdw(0|1), Height(0.0-1.0),
      Position(0|1|2), r(0.0-1.0), g(0.0-1.0), b(0.0-1.0)
```

Plot Legend

graph_set_legend_pos — Sets position of the legend within the plot

```
graph_set_legend_pos <x y> (both numbers between 0.0 and 1.0)
```

graph_set_legend_label — Sets the legend label for a dataset

```
graph_set_legend_label { -graph <number> } <label>
```

graph_set_legend_label_info — Sets the state of a legend label

```
graph_set_legend_label_info { -graph <number> }
      Font(0-5), Bold(0|1), Italic(0|1), Drop Shdw(0|1), Height(0.0-1.0),
      Position(0|1|2), r(0.0-1.0), g(0.0-1.0), b(0.0-1.0)
```

Miscellaneous Dataset Information

graph_set_plot_style — Sets the style of the current plot

```
graph_set_plot_style { -graph <number> }
      0 (Line) | 1 (Scatter) | 2 (Area) | 3 (Bar)
```

graph_set_plot_color — Sets the color of the current plot

```
graph_set_plot_color { -graph <number> }  r, g, b (0.0 - 1.0)
```

graph_set_line_thickness — Sets the style of the current line thickness

```
graph_set_line_thickness { -graph <number> }
    a number between 0 and 50
```

graph_set_line_style — Sets the style of the current line

```
graph_set_line_style { -graph <number> }
    0 (Solid) 1 (Dash) 2 (Dot) 3 (Dot-Dash)
```

graph_set_scatter_symbol — Sets the symbol for the current scatter plot

```
graph_set_scatter_symbol { -graph <number> } <symbol>
```

graph_set_scatter_info — Sets the state of the scatter plot symbol

```
graph_set_scatter_info { -graph <number> } Font, Height
```

graph_set_xaxis — Sets the state of the x axis

```
graph_set_xaxis
    Axis Scale(0|1), From, To, Tic Marks(0|1|2|3), Num Tics, Precision
```

graph_set_yaxis — Sets the state of the y axis

```
graph_set_yaxis
      Axis Scale(0|1), From, To, Tic Marks(0|1|2|3), Num Tics, Precision
```

graph_show_line_ice — Presents the Line Color Editor popup

```
graph_show_line_ice
```

graph_show_scatter_ice — Presents the Scatter Color Editor popup

```
graph_show_scatter_ice
```

graph_show_bar_ice — Presents the Bar Color Editor popup

```
graph_show_bar_ice
```

graph_show_area_ice — Presents the Area Color Editor popup

```
graph_show_area_ice
```

graph_show_label_ice — Presents the Label Color Editor popup

```
graph_show_label_ice
```

User Interface Layout Commands

Introduction

This section describes the CLI commands used to create or modify the user interface layout of an AVS network. These commands offer additional control over the user interface to more advanced users or module developers, providing for complete control over default user interface layouts and supporting dynamic user interfaces that can be reconfigured to respond to changing conditions during use.

A layout is a arrangement of *manipulator* widgets, that are used to input data to module parameters, and *panel* widgets, which organize other manipulator or panel widgets into a hierarchy. The choice of which module parameters are accessible to the user and how they are presented is very important in making a usable application out of a collection of modules or in making a complex module with a large number of parameters more manageable. The layout hierarchy may closely parallel the network's organization as modules; alternatively, it may combine parameters from different modules into shared panels as appropriate.

A layout may be created and modified in several different ways: the AVS kernel creates default layouts automatically; the user can interactively change the layout in the Network Editor; or the user or a module can use CLI layout commands.

The most common approach is to allow the AVS kernel to build a default layout automatically, based on the modules that are used in a network and the order and data type of the parameters that they define. Through the use of parameter properties, the module writer can influence how the default layout will appear. This approach is described more fully in the "AVS Routines" appendix.

When the default layout is unsatisfactory, the user can interactively alter the default layout using two different components of the Network Editor: the Layout Editor and the Parameter Editor. The Layout Editor permits the user to grab individual widgets and change their position or size, reparent them to reorganize the hierarchy, change what kind of widget is being used, or delete them entirely. The Parameter Editor provides a complementary way to change the type of widget being used without directly controlling placement or parentage. These facilities are described in the "Network Editor" chapter in the *AVS User's Guide*.

The third approach is to use the CLI layout commands directly to select what kind of widget is used; its placement, size, and place in the hierarchy; and other properties, such as visibility. The commands can be used in several ways. Since networks and their layouts are stored out in the form of CLI commands, a developer can directly edit the network file to fine tune geometry or widget properties. A more powerful application of the layout commands is to use them from a module to directly create or reconfigure the module's own user interface. A module with many parameters may be easier to use if its default layout is reorganized into related groups of parameters on different panels, in a *stack* panel widget, which shows one child panel at a time, depending on which button is selected. A more complex module, such as the Module Generator, can use the layout commands to initially hide portions of its user interface and only present them when they are appropriate.

Basic Layout Concepts

The layout commands consist of edit commands (**manipulator**, **panel**, **shell**), which can incrementally create, modify, or reorganize widgets; a general query command (**layout**) which will report the current layout of part or all of the widget hierarchy in terms of edit commands; a general destroy command (**delete_widget**); and an environment command (**window_mgr**) which tells AVS which window manager environment is in use, to allow it to accurately determine top level window locations.

The layout widgets are object oriented and organized into a class hierarchy. All widgets are *widgets* and can be operated upon by the **layout** and **delete_-widget** commands. There are two major subclasses, *manipulators* and *panels*, each of which has their own editing command (**manipulator** and **panel**), since they respond to slightly different options. Manipulators are widgets that control the value of a module parameter, and panels organize manipulators and other panels into a widget hierarchy. Manipulators have subclasses based on the parameter data type and can be changed to different subclasses of the same data type (i.e. a real parameter can change from a dial to a slider). A simplified widget class hierarchy looks like this:

```
widget
   manipulators
      integer-type widgets
         idial
         islider
      real-type widgets
         dial
         slider
   panels
      panel
      stack
```

The edit commands take a widget name, and a series of optional clauses specifying the widget's class, immediate parent, position, size, properties and some options such as visibility. An edit command will either create a new widget or modify an existing one; unspecified options will be set to default values when creating a new widget. The widget name is the basis for associating the widget with a module or module parameter and is described at some length below. For example, if an instance of the "clamp" module had no user interface, you could create a dial widget and attach it to the "clamp_min" parameter with the following command:

```
manipulator clamp.user.0:clamp_min -w dial
```

Missing information such as the widget's parent, location, and size will default to appropriate values.

The **layout** command can be used to report on part or all of the widget hierarchy, down to a specified level. If we asked about the manipulator widget that was just created it might look like this:

```
layout clamp.user.0:clamp_min
```

and produce:

```
manipulator clamp.user.0:clamp_min -w dial -p clamp.user.0 \\
    -xy 10,10 -wh 90,130
```

The **delete_widget** command destroys one or more widgets by name, leaving the underlying modules or parameters without any current user interface. To remove the widget that was just created we would enter:

```
delete_widget clamp.user.0:clamp_min
```

The widget can now only be recreated by using a new **manipulator** or **panel** command, as appropriate.

Widget Naming

Panel widgets are used to organize other widgets and may be directly associated with individual AVS modules. Manipulator widgets are always associated with parameters. The widget name is the basis for making the association between the widget and the module or parameter.

A panel with the same name as a module is recognized as that module's main control panel and will be the default parent for manipulator widgets attached to its parameters. The name can either be the standard instance name such as ("clamp.user.0") or an alias name (see "Module Names and Aliases" above); when referenced from within a module by the **AVScommand** function, the name should be the **$Module** generic name. One or more secondary panels can also be associated with a module by a name prefixed by the module name and an exclamation point (!), as in **$Module!subpanel**. A major advantage of panels that are associated with a particular module is that they will be automatically deleted when the module is destroyed. For example if the "clamp" module needed to organize its widgets into subpanels, it might create a new panel with the following command:

```
panel $Module!my_subpanel -p $Module
```

and the new panel would end up being called "clamp.user.0:my_subpanel" if clamp was the first module in the network or "clamp.user.1:my_subpanel" if it was the second module. For most of the examples given here, we will use the instance name as if the commands were being typed into the CLI by the user.

Additional panels are used to organize the module panels and parameter manipulators. There are several standard system widgets which provide a common framework for network layouts. The root window is referred to as "ui", a *shell* panel widget; all of its immediate children will be root level windows, given window decorations by the window manager. The standard Network Control Panel is an an *app_panel* panel widget named "Application"; it has the AVS logo, help/viewers menu, and status bar built into it. The stack panel widget called "Top Level Stack" is placed within "Application", and serves as the default parent for all new module panels and "pages" and stacks created by the Layout Editor. These relationships can be seen in a simplified layout of a network with two modules, "boolean.user.0", which is contained within "Top Level Stack" and "clamp.user.1", whose control panel was made a root window.

```
shell "ui" shell
 panel Application -w app_panel -p ui
  panel "Top Level Stack" -w master_stack -p Application
   panel boolean.user.0 -w panel -p "Top Level Stack"
    manipulator boolean.user.0:boolean -w toggle -p boolean.user.0
 panel clamp.user.1 -w panel -p ui
  manipulator clamp.user.1:clamp_min -w dial -p clamp.user.1
  manipulator clamp.user.1:clamp_max -w dial -p clamp.user.1
```

The developer can create panels that are not associated with a particular module, either using the Layout Editor or the CLI. The Create Page button in the Layout Editor creates panel widgets named "page.N"; these are sometimes referred to as "pages" even though they are standard panel widgets. The Create Stack button creates new stack panel widgets named "stack.N".

A manipulator widget with the same name as a module parameter will be attached to that parameter (see the "Parameter Names" section above). If the parameter is modified by the module, the manipulator is automatically updated to show the current value. When the parameter is destroyed, the widget is automatically destroyed as well.

In some cases, manipulators are also used to control the internal parameters of widgets themselves. These are not seen very often but are used to refer to panel title widgets. The names consist of the name of the widget itself, an exclamation point and the name of the internal parameter.

Geometry

Widget geometry consists of the widget’s position (**-xy**) within it’s parent window and its size (**-wh**). Both sets of coordinates are provided in canonical screen size coordinates (a standard 1280x1024 screen). On different size screens, these coordinates are scaled up or down automatically as the command is interpreted by AVS or saved out to a network.

The widget position does not need to be provided; it’s parent will provide a default location if necessary, based on its current contents. Some panels, such as stacks, ignore a requested location, placing all children in the same location.

The widget size information is also optional; it will default to the standard size established by the widget class. In some cases, widgets will ignore size information because they are a fixed size.

Module Based Layout Control

Using the layout commands from a module offers the opportunity to predetermine the module's default location or to dynamically change the module's interface during operation as changing conditions warrant that certain parameters appear or disappear. Standard AVS modules such as the Module Generator or particle advector use these options to improve their user interfaces. Specifying Default Layouts

The **layout** property is a tool for creating a preferred default user interface. It is a parameter property containing a block of one or more commands that will be executed when the module is first instanced. This block can contain a single command that specifies the manipulator widget that should be attached to that individual parameter or an entire module user interface, complete with new panels to regroup the various modules.

The basic procedure to create such a description is the following:

- Make an instance of the module in the Network Editor.
- Using the Layout Editor, rearrange the interface the way you want it.
- Using either the Write Network operation or the **layout** command directly from the CLI ("layout -none 0 mod_name **>** output_file"), write out the module's layout.
- Using a text editor make the following changes to make the layout description into a string block that the C compiler will like:
  - Change all instances of the module name to **$Module** so it is generic.
  - Change all quotation marks (") to escape-quote (\\") so that they are handled as characters by the C compiler.
  - At the end of each line, add a newline-continue string (\en\e) so that the strings will be handled as one big string in C.
  - Edit out layout command options that will default reasonably (such as module panel location, etc.)
- Insert the resulting test as a string block property using an **AVSadd_parameter_prop** command on the first parameter in the module.

For example, after saving a layout of an instance of the clamp module, and editing the string into a block in the module description function, the result might look like the following:

```
iparm = AVSadd_float_parameter("clamp_min", 0.0, FLOAT_UNBOUND, FLOAT_UNBOUND);

AVSadd_parameter_prop(iparam, "layout", "string_block",
       "panel $Module -w panel -p "Top Level Stack" -wh 200,150\n\
     manipulator $Module:clamp_min -w dial -xy 10,10 \n\
     manipulator $Module:clamp_max -w dial -xy 100,10 ");
```

Dynamic Layouts

The **AVScommand** function can be used from a module's compute function to talk to the CLI to query the module's current layout or to actively rearrange it through a combination of the **manipulator** or **panel** commands. As with the **layout** property, some special conventions need to be followed to produce an acceptable string; the module name should be the generic **$Module** name, quotes need to be escaped and newlines need to be handled properly (at least for multiple line command blocks). The module can issue commands to change the size of its panel, rearrange widgets into different configurations to conserve screen space or to change the visibility of individual widgets.

Layout commands

delete_widget — Delete individual widgets

Widgets are either panels (panel or stack class widgets) or manipulators (widgets that change parameter values). Deleting a panel widget will also delete all of its children unless they are locked against deletion (some image or pixmap windows) in which case they will pop out of the panel and become independent widgets. More than one widget may be deleted at a time.

```
delete_widget <manipulator | panel> ...
```

layout — Write description of current layout

This command will list the current layout of selected widgets or all widgets by default. If a panel widget is specified, the layout of all of its children are displayed as well as the initial widget itself, depending on the number of levels desired. Layouts are reported using **manipulator** or **panel** commands. Each reports the widget name, the widget class (-w), the widget's parent (-p), the location (-xy), and the size (-wh). Widgets may also report property values that have been modified from the initial defaults in the form of -P <property name> <property type> <property value>. (See also the discussion of the **AVSadd_parameter_prop** call in the "AVS Routines" appendix of the *AVS Developer's Guide*.

```
layout { options } { name,... }
     # Options to control how much layout information is presented
     {-l levels}   # Number of levels in the hierarchy to report (default all)
     {-root}       # Adjust reported window positions for window manager
     {-none 0/1}   # Do not report widgets with -none as their widget class
                   # (displayed by default)
```

manipulator — Create or modify a manipulator or list available classes

Manipulators are things like dials and sliders that are used to change module parameter values. Their names take the form module:parameter. The widget class must be compatible with the parameter type. Most manipulator widgets will ignore the size (**-wh**) option, although all browsers will recognize it (in pixel coordinates). The parent widget must already exist. Property values will alter the appearance or behavior of various widgets. Manipulator options that are not specified will receive reasonable default values - parameters will be parented to their module's panel widget, get a new location, default class, etc.

The **-reconfig** (reconfigure) option will request a manipulator to request its parents to resize themselves. The **-no_create** option inhibits creating the widget if it does not already exist. Manipulators may be hidden temporarily from view using the **-hide** option and then be shown again using the **-show** option.

The **manipulator** command with no arguments will list the current classes of widgets for the known parameter types. See the *AVS Developer's Guide* for more information on parameters, widgets, and widget properties.

For example, to attach a dial to a module parameter called "clamp.user.1:clamp_min" the following **layout** command might be used:

```
manipulator clamp.user.1:clamp_min -w dial -p clamp.user.1 -xy 10,10
```

This would attach a dial, parent the widget to an existing panel named clamp.user.1 (directly related to the module), and place it at 10,10 in that panel. Use the Layout Editor to create a basic layout that you want and look at the resulting network file layout commands for more examples.

```
manipulator { name {-w class} {-p parent} {-xy x,y} { -wh w,h}}
        { -show/hide } {-no_create} {-reconfig}}
        { {-P <prop name> <prop type> <prop value> } { ... } }
```

panel — Create or modify a panel widget or list available classes

Panels are widgets that can contain other widgets, such as manipulators or other panels. Their names are either the names of existing modules (or module aliases), or generated names such as "page.0", or user supplied names. The size information may be ignored by certain classes of panels: stacks will usually determine their own size. While there are a variety of panel classes, the most useful are panels and stacks; most other classes are special purpose classes for device managers or AVS system support.

The **-reconfig** option will request a panel to redetermine its ideal size from its current collection of children and inform its parents. The **-no_create** option inhibits creating the widget if it does not already exist. Panels may be hidden temporarily from view using the **-hide** option and then be shown again using the **-show** option. The **-xwindow** option will report back the main X window in the panel to allow a module to parent its own X window into the panel directly.

The **panel** command with no arguments will list the current classes of panel widgets available. For more information on panel widgets (panels and stacks), see the "Layout Editor" section in the "Network Editor" chapter of the *AVS User's Guide*.

```
panel  {name {-w class} {-p parent} {-xy x,y} {-wh w,h}
        { -show/hide } {-no_create} {-reconfig}} {-xwindow}
        { {-P <prop name> <prop type> <prop value> } { ... } }
```

shell — Describe top level widget

This command is a place holder for information about the root widget (ui) and is ignored by the CLI even though it is written out in layout information.

```
shell (ignored)
```

window_mgr — Identify current window manager

The Layout Editor needs to determine the exact location of top level windows. The report locations can be affected by the window manager being used and the Layout Editor needs to make adjustments for various window managers to get reasonable values. The argument to this command specifies the current window manager; with no arguments it will report the window managers that it currently knows about. If your window manager is not on the list, try the listed window manager values to see if one of them is close enough.

```
window_mgr { <wm> }
```

Application Commands

These commands were developed to support application modules such as the Data Viewer module. They provide several basic services such as asking for a new file, controlling the help browser to present a particular help file, and creating and modifying the pull down menu bar.

choose_file — Choose file from a browser

This command is useful for applications to popup a browser for a filename to be provided by the user. If the user closes the browser instead of selecting a file, it will return $NULL; otherwise it will return the value selected, constrained by the list of allowed extensions.

```
choose_file <choice name> { <directory> { <extensions>} }
```

doc_browser — Present the documentation browser

If no topic is given, the help browser is presented showing the most recent topic. If a topic is given, the related topic documentation is presented if possible; if the topic is not found, the default help file is presented instead. The **-check** option just verifies that the topic can be looked up and read in. Topic browsers are described in more detail in the "On-Line Help Facility" appendix.

```
doc_browser { <topic> {-check} }
```

menu — Manage a list of menu items and associated callbacks

The **menu** command defines or modifies a pulldown menu on the menu bar. Pressing a menu button executes an associated CLI command. A menu is a named list of name-value pairs. Each **<name>** is shown as a button and used when disabling or reenabling the menu item or deleting it from the menu. A new name-value pair can be added in an initially enabled (**-add**) or disabled state (**-add_disabled**). The **-clear** option will clear out the entire list. A series of **-add**, **-delete**, and **-clear** transactions can be combined in a single **menu** command.

The value is either an entire CLI command (by default) or a value to be sent to the module parameter that is associated with the particular menu. The **-modparm** option specifies a particular module:parameter that will receive the value when the button is hit. The **-modparm_lock** option requests a variation in which a button hit will automatically lock the menu bar until the module unlocks the menu bar using the **-unlock** option.

Before a menu can be seen it must be made a **-pulldown**, though it may be built up and then made a **-pulldown** in a subsequent **menu** command. If a **<menuname>** is given with no new options or value pairs, its current contents is shown. If no menu names or options are given, the currently known menu names are listed. A menu may be deleted from the list using the **-destroy** option. You can only change the order by destroying the menu and recreating it at the end of the list. Menus associated with modules are deleted automatically when the module is destroy. The menubar is automatically cleared on a Net Clear operation.

Some operations control the overall menu bar itself: **-show** and **-hide** control its visibility, **-lock** and **-unlock** disable or re-enable any menu picking operations, and **-status** enables or disables the display of the status bar. For example, to make a simple menu that will send values to a module parameter called clamp.user.1:clamp_min, the following commands would create a menu:

```
menu ClampMenu -pulldown -modparm clamp.user.1:clamp_min \e
     -add "Min Value" 0.0 -add "Mid Value" 112.5
menu -show
```

Then later if the Min Value was temporarily an invalid operation, the following would grey out and disable the button until re-enabled:

```
menu ClampMenu -disable "Min Value"
```

```
menu { -show | -lock/unlock | -hide | -status on/off }
    {<menuname> {-modparm/modparm_lock <mod>:<parm>}
    { -pulldown }
    {-clear | -destroy | -add/add_disabled <name> <value> |
     -delete/disable/enable <name>} }
```