AVS DEVELOPER'S GUIDE

Release 4
May, 1992

Advanced Visual Systems Inc.

Part Number: 320-0013-02, Rev B

NOTICE

Advanced Visual Systems Inc.
300 Fifth Ave.
Waltham, MA 02154

TABLE OF CONTENTS

1 AVS Overview

2 AVS Data Types

4 Advanced Topics

5 Command Language Interpreter

B AVS C Language Field Macros

C Examples of AVS Modules

D On-Line Help Facility

E Unstructured Cell Data Library

F Field Arguments in FORTRAN

G Geometry Library

H The F77_Binding Utility Program

List of Tables

List of Figures

CHAPTER 1

AVS OVERVIEW

Introduction

The AVS system allows users to dynamically connect software *modules* to create data flow networks for scientific computation. These modules pass data of mutually agreed upon types between each other. Programmers can extend AVS by developing new modules. There are a variety of ways in which modules can be integrated into AVS. These allow the user a spectrum between dynamic configuration and maximum efficiency.

This manual describes what a programmer needs to know to write AVS modules. The manual assumes an elementary understanding of the concept of a data flow network and a working knowledge of either the C or the FORTRAN programming languages. It also assumes familiarity with AVS on the user level. For AVS user documentation, see the *AVS User's Guide*.

AVS Overview

AVS consists of two major parts: the main application (which includes the AVS Kernel) and AVS modules, which are computational units that can be linked together into flow networks.

The AVS Kernel includes the Network Editor, the control panel Layout Editor, the user interface code, functions that control execution of AVS flow networks, and communications functions. The functions that control the execution of flow networks created by the Network Editor are collectively referred to as the flow executive. When a flow network is active, its modules are invoked in turn (and only when necessary) by the flow executive.

User-written modules are implemented as separate UNIX programs. They communicate with the AVS Kernel using the Berkeley UNIX socket mechanism and, in some cases, use shared memory. UNIX domain (local machine) sockets are used where possible for efficiency, otherwise, TCP domain sockets are used (for example, when modules are running on a remote machine, or there are no more UNIX domain sockets available).

Modules can also be "builtin," i.e., linked directly into the AVS Kernel. Several of the modules supplied by Advanced Visual Systems are currently im-

plemented as "builtin" modules. While you can develop your own modules, you cannot create "builtin" modules.

As the number of modules in use increases, AVS may use quite a large number of process slots in the UNIX kernel. As an efficiency enhancement, users can place multiple modules into a single executable. Placing multiple modules in a single executable can cut down on system memory used as well as reduce the number of process slots needed. The programmer declares multiple modules in a single program by including multiple *description functions* in the source code, each of which describes the relevant entry points and data structures for a single module. Next, these modules are linked together in one executable. The description functions are registered with the AVS Kernel by making calls to the **AVSmodule_from_desc** routine within a user-supplied function that must be named **AVSinit_modules**.

AVS passes high-level descriptions of images and geometric data to the graphics display modules (Data Output modules). AVS implements these modules using the graphics library interface that each platform supports (PHIGS, PEX, GL, XGL, Dore, X, etc.). This provides high performance rendering in a device independent manner. AVS also uses the X-Window library for windowing and for generating the user interface widgets.

Modules

The fundamental unit of computation in AVS is the module. Modules process inputs to generate outputs. Modules are intended to be fairly high-level units of computation. For example, a module might be designed to compute a threshold for a scalar field, but it would be inappropriate to design a module to add two numbers. Modules also have parameters that the user can adjust at run time to affect the action of the computation.

Modules specify to the AVS Kernel what data inputs they expect to receive from other modules (image data or a color map, for example). Data input can be required by the module, or it can be optional. Modules can also specify data outputs. You connect these outputs to other modules that have compatible inputs. To allow user interaction, modules define user-interface parameters that are displayed and monitored by the AVS Kernel (for example, a threshold value or a file name).

The module writer assigns a data type to each user-interface parameter and can associate with it a particular widget that you use to set and view the parameter’s value. For example, an integer parameter could be displayed and controlled using a dial widget, but it could just as well utilize a slider or typein widget. AVS users generally should be allowed to control parameter values. However, a module can set a parameter value internally at any time, which may be necessary if the user sets a parameter to an illegal or nonsensical value.

You can conveniently extend the capabilities of AVS by writing a new module. Because AVS operates on fairly general data types, you can define new

modules to work with existing modules. Application developers can concentrate on algorithms that implement new functionality by building on capabilities that already exist and utilizing the flexible user interface widgets. The **Module Generator** helps developers in the process of creating these new modules. A complete description of the **Module Generator** can be found in the *Applications Guide*.

There are two kinds of modules, *subroutine modules* and *coroutine modules*. A subroutine module's computation function is invoked by AVS whenever its inputs or parameters change. A coroutine module executes independently, obtaining inputs from AVS and sending outputs to AVS whenever it wants. In this document, "module" generally refers to a subroutine module, and "coroutine" refers to a coroutine module. Most of the modules provided with AVS are subroutine modules.

Most subroutine modules consist of two main functions: the **description** function and the **computation** function. You can write these functions in either C or FORTRAN. The description function describes what data the module takes as input and what data it produces as output, as well as the parameters that control its behavior. The computation function performs the operations intended by the module developer. It is called whenever an input or user parameter changes. The data structures implicitly defined in the description function for inputs, outputs, and parameters are passed as arguments to the computation function. The computation function typically operates on the inputs and parameters to produce new output.

The flow executive calls a subroutine module's computation function when the module is marked as "changed" and it is the next changed module in the run queue. A module is defined as "changed" when an input or parameter has been modified. The run queue is only processed when the flow executive is enabled. You can enable and disable the flow executive from the "Network Tools" menu in the Network Editor.

You can convert many existing simulations and other scientific applications to AVS coroutine modules by making the application use AVS data types, inserting calls to transmit data to and from AVS, and writing a description function.

Data Types

There are two general classes of data in the system: primitive data and aggregate data. Primitive data items are simple objects such as bytes, integers, floating point numbers and text strings. Aggregate data items are the large chunks of data that characterize modern scientific applications. One fundamental type of aggregate data is called a field. Basically, a field contains computational data along with associated coordinate data. For example, pressure and temperature samples might be stored as computational data in a field, and the sampling coordinates would be stored as the coordinate data in that same field. The pressure and temperature samples can be associated as vector elements comprising each computational sample to be associated with a single point.

AVS contains aggregate data types useful for defining geometric as well as numeric information. The geometry data type provides a flexible mechanism for defining geometric objects. The unstructured cell data (UCD) type provides a way to define a geometric object composed of discrete cells with associated data. UCD definitions are particularly useful for finite element analysis and computational fluid dynamics applications. The molecular data type (MDT) provides a way to define molecular and quantum structures. MDT addresses the needs of classical, substructure and quantum chemistry fields.

AVS also has other types of aggregate data, including colormaps and pixmaps. Colormaps are data structures that define color lookup tables which you can use to map numeric values to colors. Pixmaps are used to keep track of X-Window pixel maps used to directly update the screen. Most users do not deal directly with pixmaps because AVS provides modules that create pixmap outputs from fields, geometries, and unstructured cell data types.

Any data type can be used as module input, but generally, only the primitive data types are suitable for use as parameters. The only difference between a parameter and module input is that parameters are usually associated with user interface widgets.

AVS Flow Networks

An AVS user builds an application by constructing a network of modules. A typical network might consist of modules performing three kinds of tasks:

- Importing data from outside AVS (or generating their own data) and converting it into data of one of the AVS data types.
- Transforming AVS data in some way, producing output data of the same or of a different AVS type.
- Rendering the data on a display screen, printer or plotter, or storing the data to a file.

A module can receive data through an *input port* and transmit data through an *output port*. A user who connects two modules is actually connecting an output port of one module to an input port of another module. You can connect two ports when they have matching AVS data types.

When a flow network contains remote modules (modules that are executing on a remote machine), the data architecture (for example, the byte order or the floating point format) used on the remote machine may be different. However, this is not a problem for passing data between modules because the AVS executive uses the External Data Representation (XDR) format when passing data between dissimilar machines.

Data Flow

The purpose of constructing a network is to provide a data-processing pipeline in which, at each step, the output of one module becomes the input of another. In this way, data can enter AVS, flow through the modules of a network, and finally be rendered on a display or stored outside AVS.

This process requires that each module in a network be invoked at the appropriate time. For a subroutine module, the computation function must be executed whenever the inputs or parameters change. AVS has a flow executive that is normally active during the life of the application. The flow executive supervises data movement between modules, keeping track of which inputs and parameters have changed and invoking modules in the correct order.

AVS uses a remote procedure call mechanism to establish communication between modules. When the user starts up a module, AVS creates a new process in which that module runs. (When multiple modules are combined into a single executable, a new process is not created.) AVS also sets up a connection between the module and AVS, using the Berkeley UNIX socket mechanism. Both sides use remote procedure calls and, if possible, shared memory to communicate through this connection.

AVS allows coroutine modules to execute independently. A coroutine is often a simulation or animation; an application that executes multiple times to produce a series of frames or data sets. AVS communicates with coroutine modules through the same sort of remote procedure call mechanism it uses to communicate with subroutine modules.

Modules will attempt to execute in parallel if *all* of the following conditions are met:

- The user has specified the *-parallel n* command line option, where *n* is the maximum number of modules that can execute at one time.
- There are multiple processors, local and/or remote, available for module execution.
- The module has no input dependencies upon other modules that would be executing at the same time. For example, networks often have multiple, independent, parallel branches made up of multiple filtering and mapping techniques. These independent branches can execute in parallel.
- Modules executing in parallel must be in different processes. If the modules happen to be compiled together to execute as one process (as most AVS modules are), the user can force execution as separate processes using either the blanket *-separate* option to AVS which will cause all modules to execute separately, or by fine-tuning their networks using the Module Editor panel's **Process** and **Group** controls to explicitly organize different modules into different processes. See the "Advanced Topics" chapter later in this manual and the "Module Editor" section of the *User's Guide*'s "Advanced Network Editor" chapter.

Module Life Cycle

When AVS starts, it searches for libraries of modules to load. The libraries are specified by the *avsrc* file **ModuleLibraries** entry. First, AVS always reads the system default startup file in */usr/avs/runtime/avsrc*. Users may override or supplement the options in the system startup file with a personal *avsrc* file. AVS looks for user *avsrc* files in the following order: 1) *./.avsrc* in the current directory, 2) *$HOME/.avsrc* in your home directory.

For each **ModuleLibraries** entry, the library files specified are used. The module palettes are built using information from the library file. Since the library file is ASCII text, it is easy to edit and comment out modules that are not wanted prior to starting AVS.

When a module is instanced in a network, the executable for the module is run as a UNIX process (unless it is a "builtin" module). The module description function is called and information about the module's inputs, outputs, parameters, and computation function are sent back to the AVS Kernel. The Kernel automatically builds user interface widgets for any input parameters. If the module has specified an "initialize" function, that function is called. Please note that during the initialization process, calls to **AVScommand** will not work.

When it is time for a module's computation function to run, the flow executive sends the module a message with the input port and parameter values. The first message is not sent until all input ports with the **REQUIRED** option have been attached and data is available, at which point the module's computation function is called.

When a module is "hammered" (destroyed), the AVS Kernel sends a shutdown message to the module. If the programmer has specified a destruction function, it is called before the module exits. Please note that during the destroy process, calls to **AVScommand** will not work.

Use of Shared Memory

Shared memory regions are a form of interprocess communication that allow sharing of data between processes without having to copy the data. The data must be placed in a memory buffer that is set up with UNIX system calls to be a shared memory region. Multiple programs can then map to the same pages of physical memory using their own virtual addresses, by making UNIX system calls. There can thus be a single copy of the data being accessed by multiple programs.

The AVS field and UCD data types use shared memory. Geometries, pixmaps, colormaps, user-defined data, the molecule data type, etc., do not.

The AVS kernel attempts to share data among modules when possible, by placing the data in a shared memory region. Pointers to the data are passed

between modules just as if each module had its own private buffer. This technique cuts down on memory usage and also speeds processing because the data does not have to be copied. (AVS 2 modules do not make use of shared memory until they are recompiled and linked with AVS 3 or later libraries.)

The use of shared memory is normally completely transparent to the user. However, because data is placed in a shared memory region by default when AVS modules are executing on the same machine, modules cannot directly alter data in an input port buffer. The shared data might also be in use by another module that would be affected by the change to the data. If a module wishes to modify data input data, it can set the special flag **MODIFY_IN** when creating the port. This will cause a *copy* of the input data to be passed to the module. If the module truly needs to directly modify the data in the input buffer, its users must be aware that they must run AVS with **ReadOnlySharedMemory 0** (disabled). See the description of **AVScreate_input_port** in Appendix A.

AVS creates the shared memory keys as follows: the high-order byte of the key is 0x1a; the next two bytes are the process id; the low order byte is a sequence number that is initially 0 for each process. If the desired key is already in use, AVS increases the sequence number and tries again.

The use of shared memory may be limited or unavailable on certain platforms. See the AVS Release Notes for more information.

Heterogeneous Network Support

AVS supports remote execution of modules. It uses the External Data Representation (XDR) format to provide a machine independent representation of data flowing between modules. The UNIX socket mechanism is used to pass requests across the network. The Flow Executive executes modules in the same order as if they were on a single machine. (AVS 2 modules do not use the XDR format for data representation and so must be recompiled with AVS 3 or later libraries in order to execute properly across a network.) When machines on both ends of the socket have the same data representations, the XDR translation layer is bypassed.

In AVS3, where connected modules on a remote host were in the same process, data would be passed directly between them using a pointer passed through the flow executive. (Most AVS modules were combined into a single executable to support this.) However, if the remote modules were in different processes, data flowed from a remote module to the AVS Flow Executive, and then back to a downstream remote module, resulting in system network overhead for every module-to-module connection on a remote host. A better solution is to run more than one remote module from a single UNIX process. See Chapter Four for information on how to do this. Starting with AVS 4, additional sockets are created so data can be transferred directly between modules. This is called Direct Module Communcation (DMC).

The user interface supporting remote module execution is built on the **Module Tools** sub-menu of the Network Editor. The **Read Remote Modules** button brings up a browser panel which prompts the user for the name of a remote host. The contents of the panel are constructed from a Hosts file, either the system default in */usr/avs/runtime/hosts*, or any such file specified in the user's *.avsrc* file with the **Hosts** keyword. The Hosts file contains the name of the available remote hosts, a remote command to execute to establish contact with the remote host (usually *rsh*) and the pathname to a directory of AVS modules on the remote host. AVS creates a socket connection to the remote host and looks for an executable file named *list_dir* in the specified remote directory. *list_dir* is a special program that executes on the remot host and displays a browser with the contents of the remote directory, i.e., module binaries, available for execution. For complete information regarding the loading and use of remote modules, see the *AVS User's Guide*.

Data Flow Diagram

Figure 1-1 illustrates the collective impact of running multiple modules in a single process, shared memory, and direct module communication on data flow through AVS networks.

K is the AVS kernel. M1 and M2 are AVS modules. D represents an area allocated for data.

Note that the AVS kernel is always in a separate process. Also note that some twelve modules are "builtin" to the AVS kernel and execute in its process. The next section describes which of the supplied AVS modules execute in what processes.

In the AVS 3 and AVS 4 coroutine, no direct module communication case:

- When M1 and M2 are in different processes:
  - Without shared memory (for example, when passing geometries between modules, when the host does not support shared memory, or with **-noshm** specified), the data must be copied from M1 to the AVS kernel, which then copies it to M2. There are three copies of M1's output data.
  - With shared memory, M1 creates a shared memory segment. The AVS kernel is involved in two ways: it receives control information about the output data area from M1 and passes this along to M2, and the kernel also attaches the shared memory segment itself. M2 attaches the shared memory segment. There is only one copy of M1's output data.
- When M1 and M2 are in the same process:
  - The two modules share data by passing pointers. No shared memory segments are involved. The kernel has only a control link to the modules' process. It is not involved in passing the data. There is only one copy of M1's output data.

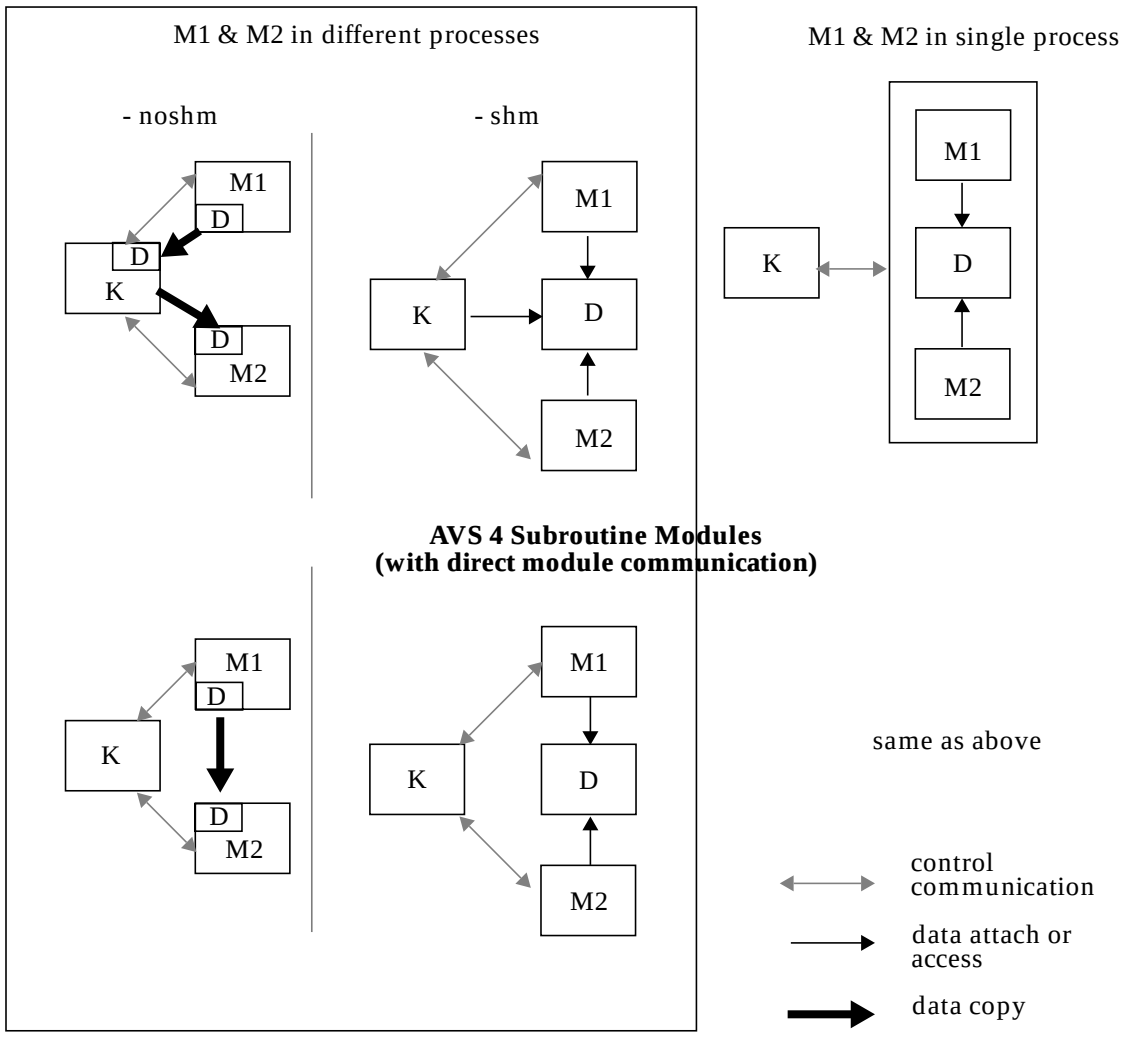

Figure 1-1 Data Flow Between Kernel and Modules

In the AVS 4 subroutine, with direct module communication case:

- When M1 and M2 are in different processes:
  - Without shared memory, the kernel's control communication channel informs M1 that M2 requires its data. M1 then contacts M2 directly, sending its output data to M2's DMC socket. M2 copies the data into its own area. There are two copies of M1's output data.
  - With shared memory, the AVS kernel's control communication channel informs M1 that M2 requires its data. M1 then contacts M2 directly. M2 attaches the shared memory segment created by M1. The kernel does not attach the shared memory segment. There is one copy of M1's output data.

- When M1 and M2 are in the same process, behavior is the same as without direct module communication.

The pathways are identical when one or both M1 and M2 are on a remote host, with one exception. Under AVS 3, remote modules in separate processes would behave has though there were no shared memory even if the remote host supported shared memory. In AVS 4, remote modules behave the same as local modules.

Note that parallel modules must execute in separate processes.

Shared memory, direct module communication, and multiple modules in a single process can all be turned on and off with command line options and/or *.avsrc* startup file keywords. Module processes can be further regulated with the Network Editor's Module Editor by adjusting the modules' group. This last option is described in the "Advanced Topics" chapter of this manual.

Some platforms' operating systems limit the number and size of shared memory segments available, and/or the number of processes that can attach to a shared memory segment. When such limits are encountered, AVS attempts to fall back on the multiple process/no shared memory approach to passing data among modules. Such limits are usually documented in the platform's release notes, along with information on how to increase the system limits, if possible.

Multiple Module Processes in AVS

There are two main categories of processes in AVS: the AVS kernel and AVS modules. The kernel always runs as a separate process. Some AVS modules are part of the kernel and run in its process. These are the so-called "builtin" modules:

generate colormap
display image
display pixmap
geometry viewer
graph viewer
image viewer
render geometry
transform pixmap (not present on all platforms)
colormap manager
image manager
render manager

Most of the rest of the AVS modules are compiled together in one of two binaries. All of the modules in one of these binaries execute in a single process. The actual partitioning of modules varies from release to release. In general:

- All UCD modules are in the binary */usr/avs/avs_library/ucd_multm*

- Most of the rest of the supported modules are in the binary */usr/avs/avs_library/mongo*

There is a set of modules, mostly coroutines, that each reside and execute in their own, separate process. The library files */usr/avs/avs_library/Supported* and */usr/avs/unsupp_mods/Unsupported* lists all modules and their corresponding binaries.

Release Compatibility

With AVS 4 there are new elements in certain data structures, including the "field" structure. To access the new elements, modules must be recompiled and relinked. In general, AVS 3 modules continue to work properly under AVS 4. You can use AVS 3 and AVS 4 modules together in a single network.

To take advantage of new AVS 4 features, such as mesh ids, existing user modules must be recompiled using the new AVS 4 libraries. Code that uses the **AVSbuild_field** routine cannot make maximal use of shared memory and is no longer the recommended approach to creating fields. AVS will, however, continue to support this routine. See Chapter Two for detailed information about allocating fields in a module.

Portability Issues

This section describes issues to consider when writing code that runs on different platforms.

Writing Portable Code

With a little effort, you can write AVS modules that do not require source code changes when porting between different platforms. Avoid the use of non-standard operating system calls and "include" files, and do not rely on hardware specific features such as integer word length.

When allocating space for data, don’t assume a particular size for a type declaration such as short, int, float, or double. For example, to allocate an array of 64 integers, don’t allocate "64*4" bytes; rather, allocate "64*sizeof(int)".

When dealing with data structures built out of bytes, do not manipulate the bytes as integers. In particular, when dealing with AVS images, the RGB data is represented as a 4-vector of bytes. Do not assume that integers are four bytes and manipulate the pixels as integers. When allocating space, for instance, do not use "width*height*sizeof(int)"; rather, use "width*height*4*sizeof(char)".

If you cast a value of "char*" to "int*", add 1 to it, and cast it back to "char *", it could have the result of adding 4 on some platforms or 8 (e.g., on a Cray) to the original pointer value, so don't assume a particular value.

When allocating memory for use within a single function, you should use the **ALLOC_LOCAL** and **FREE_LOCAL** macros in *port.h* instead of system calls peculiar to the local implementation of UNIX. This will cause AVS to use the **alloca** call on systems that support it, and **malloc** on other platforms. FORTRAN code should avoid using **malloc** on the local platform and use **AVSdata_alloc** or **AVSptr_alloc** as appropriate.

Usually, FORTRAN statements cannot exceed 72 characters.

For machines that do not support a FORTRAN BYTE or LOGICAL*1 data type, there are two routines in the AVS library, **AVSload_byte** and **AVSstore_byte**, that you can use to access and store 8-bit integer values.

You should use the FORTRAN include syntax of 'INCLUDE *file*', starting in column 7, rather than the C preprocessor form.

An appendix describes a utility program, **f77_binding**, that generates interlanguage interface functions. These functions allow code written in C to call subprograms written in FORTRAN, and vice-versa.

It is strongly recommended that user Makefiles follow the conventions of the */usr/avs/examples* **Makefile** to enhance portability. Particularly, using **Makeinclude** and the macros it provides such as **LASTLIBS**, can simplify the task of porting between platforms.

Porting Binary Data Files

When storing information in files, some data is stored in binary format. For example, the first two items of an image file are the width and height stored as a 4-byte integer. Field data is stored in binary format following an ASCII header. Geometry files contain both binary and ASCII data. Machines that use a converse byte ordering ("big-endian" vs. "little-endian") from the machine that produced the data file may reverse the order of bytes within larger data values. The supplied AVS modules that read images are written to examine the byte ordering of the size data in image files so that users may conveniently transfer image information between platforms. If you write your own module to directly read in image or field data, you need to be aware of this problem.

Starting with version 3 of AVS, geometry files are written and read using XDR format. With field files, the user has the option (on the **write field** module) of writing fields in either XDR or a system's native format. When written in XDR, the byte ordering is well defined on all machines and there is no compatibility problem. Geometry files from earlier releases begin with a 4-byte "magic number" value which is different than that for AVS 3. The supplied AVS modules that read geometries know how to read both formats. If some-

one wants to write the old geometry format, defining the environment variable **AVS_GEOM_WRITE_V2** forces the file writing routine to write the old geometry format.

Volume data files should be compatible across machines since they contain byte values only. Modules written to read volume data directly should read the data as a stream of bytes.

Program Examples Online

There are two directories that you should refer to when writing code using the AVS libraries. These directories contain many relatively simple programming examples that illustrate the subroutines and programming techniques you should use when creating new modules and geometry filters (programs that create geometries from data). Each directory contains a README file that describes the programs briefly so you can pick an appropriate template and a Makefile to show you how to build a module on your machine.

The directory */usr/avs/examples* contains module templates for both subroutine and coroutine modules. The file *README* documents what the programs do. AVS routines other than the geometry library routines are documented in Appendix A.

The */usr/avs/filter* directory contains example geometry filters that use *libgeom.a* routines to create geometries from user data. Appendix G documents *libgeom.a*.

CHAPTER 2

AVS DATA TYPES

Introduction

AVS promotes software reusability by defining a set of general, common data types for module writers to use. Some of the data types have general and specific versions; for example, a "field" is general, but a "2D field" is more specific. Modules that accept more general input data can connect to a greater number of other modules.

The data types supported in AVS can be broken into two categories: primitive data and aggregate data. Primitive data types are bytes, integers, reals, and strings. Aggregate types are fields, colormaps, geometries, and pixel maps. In general, primitive data types are used for parameters and aggregate types are used for data being passed between modules, but there are many exceptions to this. A parameter is actually an input "port" that uses a widgit to provide a value. Unlike data being passed between modules, parameters "ports" are not visible by default.

The AVS data types currently supported are following:

- *Byte* implements 8-bit unsigned integers.
- *Integer* implements standard integers (maybe 32 or 64 bit, depending on machine architecture).
- *Real* implements single-precision floating-point numbers.
- *String* and *string block* implement simple text strings.
- *Field* implements n-dimensional arrays with scalar or vector data at each point. Fields also support arbitrary rectilinear or irregular coordinate systems, and they can represent lists of points in coordinate space. Fields can contain single or double precision floating-point, integer, or byte data.
- *Colormap* implements a transfer function that you can use to map a functional value into color and opacity values.
- *Geometry* implements geometric descriptions that the geometry renderer can use to view objects. Geometry objects are usually created using calls to subroutines in the geom library; see the "Geometry Library" appendix for more information.
- *Pixel map* is actually a reference to the X server’s representation of the rendered form of an image.

- *Unstructured cell data* provides the capability to associate data and discrete geometric objects within a single structure.
- *Molecule data type* addresses the needs of classical, substructure and quantum chemistry fields. Detailed documentation on this data type is provided in the *Chemistry Developer's Guide.*
- *User-defined data* allows you to define a local data structure and pass it to other modules that also understand that particular data structure. It is currently used for upstream feedback between modules.

Fields are AVS's fundamental data type. They use the full generality of AVS's data type system to span a set of commonly used data types. This allows you to write modules that are as general as is appropriate for the application while, at the same time, allowing optimized algorithms to be used for specific cases. You can represent the output data from a typical scientific simulation as a field. AVS provides routines to facilitate the conversion of standard arrays of data to fields.

When AVS calls a C language computational routine, it usually passes an element of a certain data type as a pointer to that element. Most data types are represented as structures, which are defined in type-specific include files. Some simple types, such as integers, are simply passed directly. C routines typically get direct pointers to the data for inputs and parameters, but pointers to pointers are used to allocate the data for outputs. Therefore, a module that takes a field as input and produces a field as output is called as follows:

```
module_compute(field_in, field_out)
/* note double indirection for field_out */
AVSfield_float *field_in, **field_out;
{
  int  dims[3];
  dims[0] = MAXX(field_in);
  dims[1] = MAXY(field_in);
  dims[2] = MAXZ(field_in);
  *field_out = (AVSfield_float *)AVSdata_alloc("field 3D float",dims);

  ... compute ...

  return(1);
}
```

Since FORTRAN programs do not have direct access to C structures, there are two different ways of getting access to fields in a compute function. The first way is by having the individual elements of the C structure get passed as separate arguments. For example:

```
FUNCTION COMPUTE(F, NX, NY, NZ,...)
```

where *F* is a 3D array with dimensions *NX*, *NY*, *NZ*. AVS attempts to make the arguments to the computation function a natural representation of that data type for the programmer. The implication of this is that the computation routine written in FORTRAN often has more formal arguments than there are inputs, outputs, and parameters, with multiple formal arguments representing a single input, output, or parameter.

This approach is somewhat cumbersome and restrictive, particularly in light of issues like shared memory allocation, the range of field types that can be handled by a module, the ease of producing an invalid field, etc. It is retained from earlier AVS releases for the sake of compatibility and is documented more fully in Appendix F.

The preferred approach is to pass a field as a single integer argument that is used by many of the same field accessor functions that a C module calls, as well as by additional functions provided specifically for FORTRAN. The module MUST include an **AVSset_module_flags** call to use the single argument approach since the default is to pass multiple arguments. The single argument approach is illustrated fully in */usr/avs/examples/test_fld2_f.f*.

Table 2-1 summarizes the type declarations used for arguments to module computation functions that correspond to input ports, parameters, and output ports:

Table 2-1 C and FORTRAN Type Declarations for AVS Data Types

AVS Data Type	C Input or Parameter Data Type	C Output Data Type	FORTRAN Input or Parameter Data Type	FORTRAN Output Data Type
byte	char	char *	BYTE	BYTE
integer	int	int *	INTEGER	INTEGER
real	float *	float **	REAL	REAL
string	char *	char **	CHARACTER*(*)	CHARACTER*(*)
field	AVSfield *	AVSfield **	INTEGER (or mult. args.)	INTEGER (or mult.args.)
colormap	AVScolormap *	AVScolormap **	INTEGER (or mult. args.)	INTEGER
geometry	GEOMedit_list	GEOMedit_list *	INTEGER	INTEGER
pixel map	AVSpixdata *	AVSpixdata **		
UCD	UCD_structure	UCD_structure	INTEGER	INTEGER
User-Defined	structure *	structure	INTEGER	INTEGER

Colormaps and User-Defined data can also be passed a single or multiple arguments. The routine **AVSset_module_flags** must be called to specify the single integer argument method. Pixmaps cannot be passed as arguments to a FORTRAN computation routine.

Bytes

Bytes are declared using the data type "byte". A byte is passed to a computation routine in C as a **char** (**char *** for output) and to a subroutine in FORTRAN as a **BYTE**.

Integers

Integers are declared using the type "integer". An integer is passed to a subroutine in C as an **int** (**int *** for output) and to a subroutine in FORTRAN as an **INTEGER**. AVS has a number of data types for parameters that are also repre-

sented as integers: "boolean", "tristate", and "oneshot". See the documentation for the **AVSadd_parameter** routine in Appendix A, "AVS Routines".

Floating-Point Numbers

AVS supports floating-point data. Single-precision floating-point numbers are declared using the type "real". This corresponds to the C type **float** and to the FORTRAN type **REAL** or **REAL*4**. A single-precision floating-point number is passed to a computation routine in C as a **float *** (**float **** for output) and to a subroutine in FORTRAN as a **REAL** (a pointer to a **REAL** for output).

Text Strings

Text strings are the standard one-dimensional character strings. A character string is declared using the type "string". It is passed to a computation routine in C as a **char *** (**char **** for output) and to a subroutine in FORTRAN as a **CHARACTER *(*)** (a pointer to a **CHARACTER *(*)** for output).

There is also a multiple line string parameter of type "string_block" which is a character string that expects to handle embedded newlines.

Fields

A field is a general representation for an array of data. The array can have any number of dimensions, and the dimensions can be of any size. Each data element in the array can consist of one value or a vector of values. All values in the array are of one of four types: unsigned character (byte), integer, single-precision floating-point, or double-precision floating-point.

A field is often used to represent data elements that correspond to points in space. For example, each data element of a three-dimensional field might be a vector of values representing temperature, pressure, and velocity at some point in a volume of fluid. The field has an implicit or explicit mapping of data elements to coordinates that represent the corresponding points in space. In other words, a field is a relation between two kinds of space: the *computational* space of the field data and the *coordinate* space to which the field data is mapped.

Mapping Computational Space to Coordinate Space

AVS assumes that the computational space is logically rectangular. In the computational domain, the mesh is similar to a uniformly spaced lattice in Cartesian space. In this logical space, each dimension of the data array forms

a perpendicular axis beginning at the origin, and the interval between data elements is 1 for each dimension.

AVS supports three types of mapping between computational and coordinate space: *uniform*, *rectilinear*, and *irregular*.

Uniform Fields

In uniform fields, the coordinate mapping is direct and implicit. Each dimension of computational space is implicitly mapped to the corresponding axis of coordinate space. The first dimension of computational space is implicitly mapped to the *X* axis, the second dimension is implicitly mapped to the *Y* axis, and so on. In each dimension, the coordinate that corresponds to a given data element is the index of that element in the data array. The data is mapped to a uniformly spaced lattice in Cartesian space between the minimum and maximum extent values supplied for the field. Each cell is a constant-length line segment for a 1D field, a square for a 2D field, a cube for a 3D field, or a hypercube for a field of higher dimensions.

Because the coordinate mapping is uniformly spaced along each coordinate axis, uniform fields need to specify a only minimum and maximum value for each axis. These values represent the range over which the data extends and are specified in the same data type as the data (e.g., if the data is comprized of real values, you need to specify the extents with real numbers).

The minimum and maximum data values may be different from the data extend values if the field has been subsetted in some fashion (such as cropping, downsizing, or interpolation). Then, the field data structure contains the original field's minimum and maximum values, while the coordinates array contains the minimum and maximum extent of the subsetted data. The extents in the coordinate array are stored in this order: minimum x, maximum x, minimum y, maximum y, minimum z, maximum z, etc.

Mapper modules use the extents information to properly position their geometric representation of the subsetted data in world coordinate space. For example, a downsized data set should not appear smaller than the original data set; it should appear at the same coordinates but with less resolution (fewer computational values within the coordinate area). The computational data is treated as lying at regular intervals between the minimum and maximum extents, derived from the original data set.

As another example of how AVS uses extents, consider a data set that has been cropped. The cropped portion of the data set should not necessarily be positioned at the original minimum value along each axis. It should be positioned between the minimum and maximum extents that apply to the cropped data so that it is positioned correctly relative to other cropped portions of the data set and does not appear to be layered on top of them. The cropped data extents are stored in the physical coordinates array, and the original data extents are stored in the **min_ext** and **max_ext** arrays in the field data structure.

In the case of a "slicer" module, the extent information in the coordinates array is used to position the slice correctly in space. For example, when taking a 2D slice from a 3D data set, the computational dimension of the field representing the slice is two, but the physical (*n-space*) dimension is three. If the slice is orthogonal to the Z axis, the X and Y extents for the slice are the same as for the original data set, and are the same in both the coordinates array and the **min_ext** and **max_ext** arrays in the field data structure. However, in the coordinates array of the slice, the Z min and max are equal to each other and are used to position the slice along the Z axis in 3D space. The Z min and max in the **min_ext** and **max_ext** arrays are the same as in the original data set.

In some cases a uniform field can have a physical (*n-space*) dimension different from the computational (*ndim*) dimension. Such a situation occurs, for example, when a 2D slice of data is extracted from a 3D data set. In order to retain a sense of the original positioning of the 2D data, a third dimension can be specified in the coordinate extents arrays. The additional dimension allows a mapper module to position the data slice (and geometries derived from it, such as a uniform mesh) correctly relative to other representations derived from the original data (such as a volume bounding box or isosurface).

Rectilinear Fields

In rectilinear fields coordinate space has the same number of dimensions as computational space. Each dimension of computational space is explicitly mapped to the corresponding axis of coordinate space. The first dimension of computational space is mapped to the *X* axis, the second dimension is mapped to the *Y* axis, and so on. As in uniform fields, the data is mapped to a lattice in Cartesian space. However, each dimension of the data array has a separate and explicit coordinate mapping. The spacing of data elements along each axis need not be uniform. Each cell is a variable-length line segment for a 1D field, a rectangle for a 2D field, a rectangular parallelepiped for a 3D field, and so on. The cell dimensions can vary from one cell to the next within the field.

Irregular Fields

In irregular fields, coordinate space might not have the same number of dimensions as computational space. Each data element in computational space is explicitly mapped to a point in coordinate space. This allows for a variety of mappings. For example, a 3D computational space can be mapped to a 3D coordinate space in which each cell has curvilinear bounds. A 1D computational space can be mapped to a 2D or 3D coordinate space that does not have cells, but rather consists of a set of "scattered" points with a data element at each point.

Figure 2-1 shows the various computational to coordinate space mappings for irregular fields. The *ndim* is the dimensionality of the data array (a 1D array of numbers, a 2D array of numbers, a 3D array of numbers). The *nspace* is the line (1D), plane (2D), or volume (3D) within which the data points exist. To establish the mapping between the two, each element of the data array must have an explicit X (1D), XY (2D), or XYZ (3D) location in space defined for it.

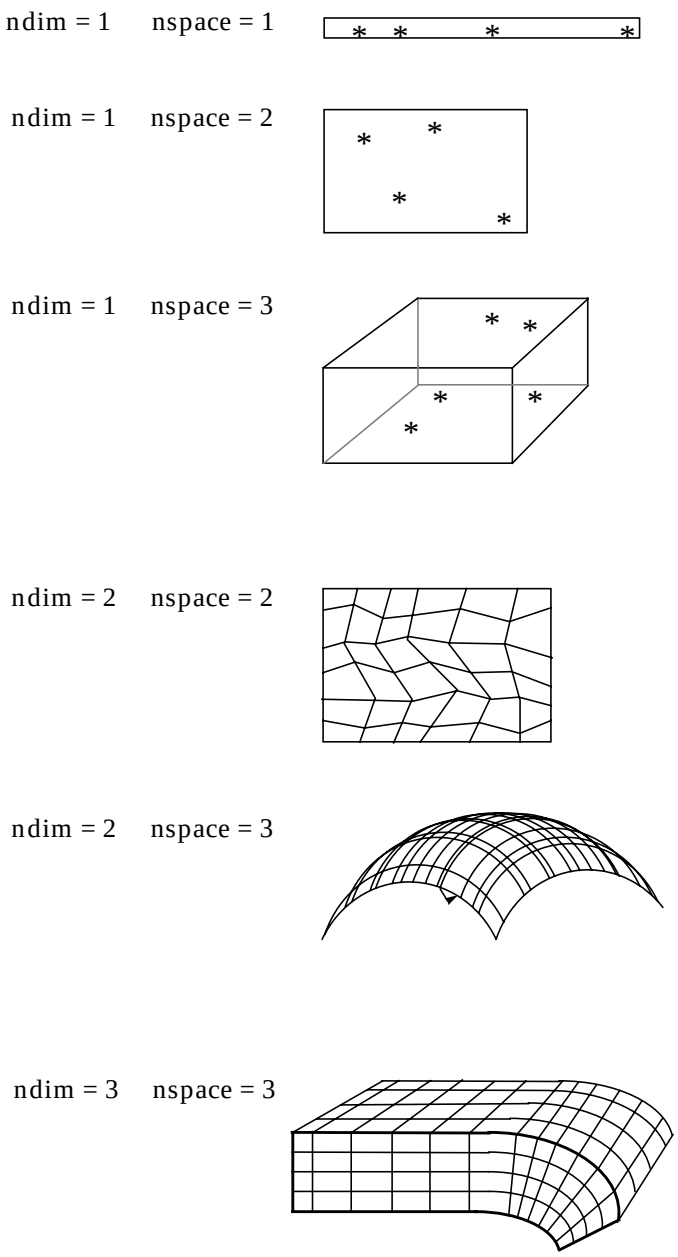

Figure 2-1 Irregular Field Computational and Coordinate Space Mappings

AVS Mapping Information

AVS needs information in different forms to specify the three mappings.

For a uniform field AVS needs only the minimum and maximum coordinates along each axis. The coordinates for each data element are implicitly assumed to be equally spaced between the minimum and maximum coordinates. The min/max coordinate values are placed in the coordinates array as well as in the arrays **min_ext** and **max_ext** in the field data structure.

For a rectilinear field AVS needs a mapping from each dimension of computational space to the corresponding axis of coordinate space. The mapping consists of one *X* value for each subscript along the first dimension of computational space, one *Y* value for each subscript along the second dimension of computational space, and so on. The total number of values in the mapping is the sum of the dimensions of the field in computational space.

For an irregular field AVS needs a mapping from each data element in computational space to a point in coordinate space. The mapping consists of a set of coordinates (*X, Y,* and so on) for each data element. The total number of values in the mapping is the product of each dimension in computational space and the number of dimensions in coordinate space.

Table 2-2 summarizes these mappings:

Table 2-2 Field Mappings of Computational to Coordinate Space

Mapping	Mapping Information	Coordinates for Data Element *(i, j, ...)*
Uniform	Implicit—Computational Dimension to Coordinate Axes	*X=i* *Y=j* ...
Rectilinear	Explicit—Computational Dimension to Coordinate Axes	*X = X(i)* *Y = Y(j)* ...
Irregular	Explicit—Computational Element to Coordinate Point	*X = X(i, j, ...)* *Y = Y(i, j, ...)* ...

Examples of Field Mappings

This section presents several examples of fields and their mappings from computational to coordinate space.

Example 1

A data set consists of 25 data elements, each representing *F(X)* for a given value of *X*. The field consists of 25 elements:

$$\{F(X(i)), i = 1, 25\}$$

The computational space is one dimensional with 25 values for *F(X)*. The coordinate space is also one dimensional with 25 *X* coordinates, one for each value of *F(X)*. The spacing between points in *X* is not constant, so the field is rectilinear or irregular.

Figure 2-2 shows the mapping between computational and coordinate space. It also presents a line graph, *F(X(i)) vs. X(i)*, of the relation between the data elements and the coordinate values.

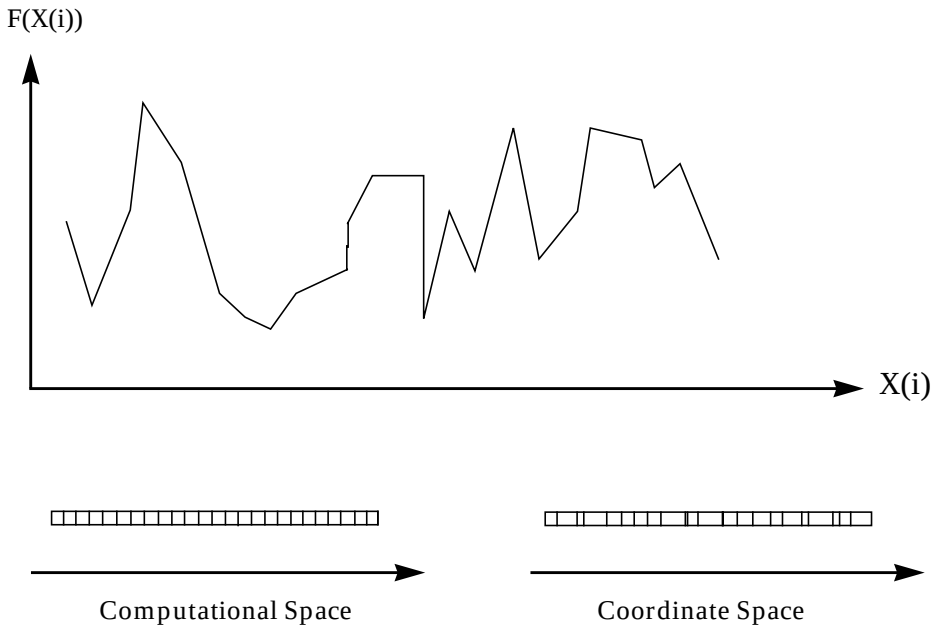

Figure 2-2 1D Computational and Coordinate Field

The following is a summary of the field characteristics:

Data type:	Floating-point
Number of values per data element:	1
Number of computational dimensions:	1
Computational dimensions:	25
Number of computational values:	1 * 25 = 25
Mapping type:	Rectilinear or irregular
Number of coordinate dimensions:	1
Number of coordinate values:	25

Suppose that each data element in this example consisted of a two-component velocity vector. In this case the field characteristics would be as follows:

Data type:	Floating-point
Number of values per data element:	2
Number of computational dimensions:	1
Computational dimensions:	25
Number of computational values:	2 * 25 = 50
Mapping type:	Rectilinear or irregular
Number of coordinate dimensions:	1
Number of coordinate values:	25

Example 2

A scalar field is defined as a two-dimensional mesh, with nonconstant spacing between both *X* and *Y* values. The field consists of 500 elements:

$$\{F(X(i), Y(j)), i = 1, 20, j = 1, 25\}$$

The field is rectilinear, with 20 *X* coordinates and 25 *Y* coordinates. Each cell in coordinate space is rectangular. Figure 2-3 shows the mapping between computational and coordinate space.

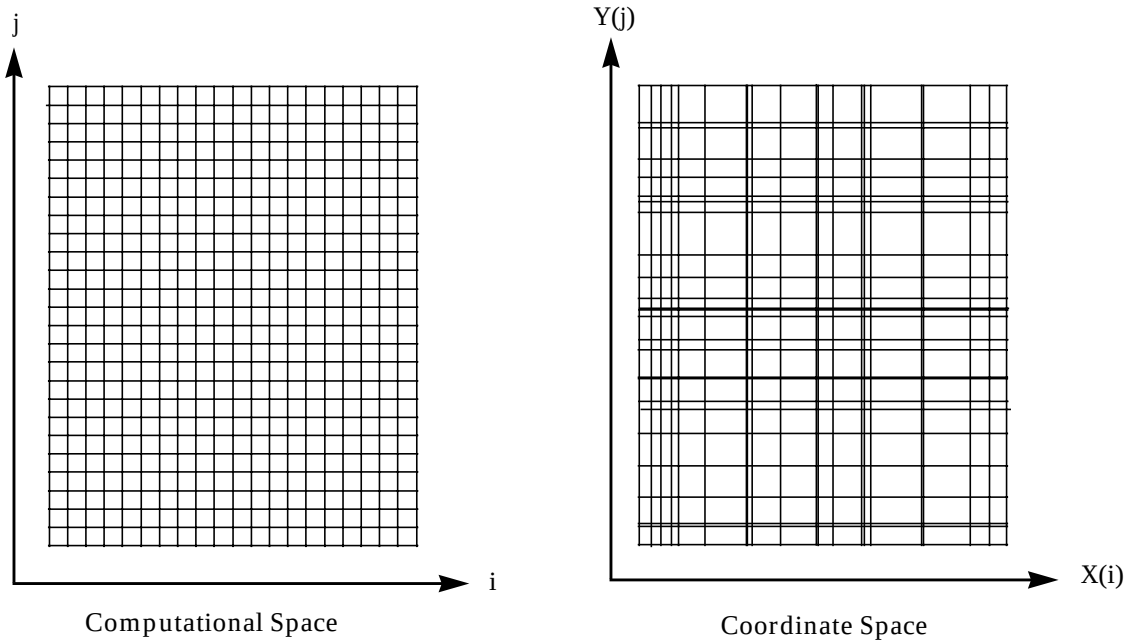

Figure 2-3 2D Rectilinear Coordinate Field

The following is a summary of the field characteristics:

Data type:	Floating-point
Number of values per data element:	1
Number of computational dimensions:	2
Computational dimensions:	20 x 25
Number of computational values:	1 * 20 * 25 = 500
Mapping type:	Rectilinear
Number of coordinate dimensions:	2
Number of coordinate values:	20 + 25 = 45

Example 3

A two-dimensional mesh is mapped to a sphere. One dimension of the mesh, *u*, corresponds to lines of equal longitude on the sphere. The other dimension

of the mesh, *v*, corresponds to lines of equal latitude on the sphere. The field consists of 500 elements:

$$\{F(X(u, v), Y(u, v), Z(u, v)), u = 1, 20, v = 1, 25\}$$

The field is irregular, with 500 *X* coordinates, 500 *Y* coordinates, and 500 *Z* coordinates. Each cell in coordinate space has curvilinear bounds. Figure 2-4 shows the mapping between computational and coordinate space.

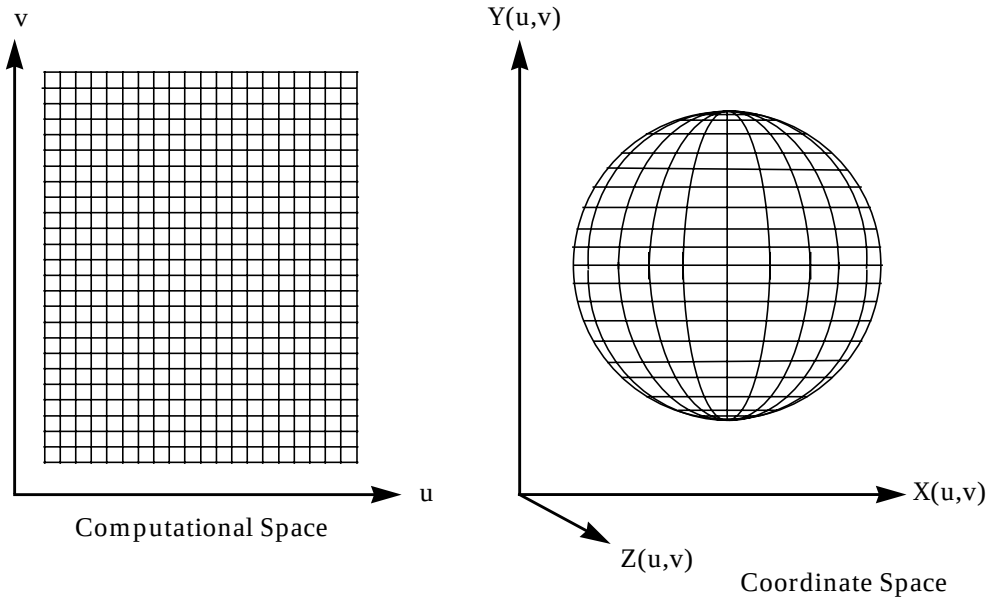

Figure 2-4 2D Computational, 3D Coordinate Irregular Field

The following is a summary of the field characteristics:

Data type:	Floating-point
Number of values per data element:	1
Number of computational dimensions:	2
Computational dimensions:	20 x 25
Number of computational values:	1 * 20 * 25 = 500
Mapping type:	Irregular
Number of coordinate dimensions:	3
Number of coordinate values:	3 * 20 * 25 = 1500

Example 4

A two-dimensional image is represented by a mesh of data elements, each of which specifies the value of a pixel. Each data element is a vector of four bytes

that specify the three color components and an alpha channel. The field consists of 65536 elements, each with four values:

$$\{V_n(i,j), i = 1, 256, n = 1, 4\}$$

The field is uniform.

The following is a summary of the field characteristics:

Data type:	Byte
Number of values per data element:	4
Number of computational dimensions:	2
Computational dimensions:	256 x 256
Number of computational values:	4 * 256 * 256 = 262144
Mapping type:	Uniform
Number of coordinate dimensions:	2
Number of coordinate values:	0

Example 5

A medical imaging data set contains 100 evenly spaced scan planes, each with a resolution of 256 x 256 pixels. Each data element is a single byte. The field consists of 6553600 elements:

$$\{F(i,j,k), i = 1, 256, j = 1, 256, k = 1, 100\}$$

The field is uniform.

The following is a summary of the field characteristics:

Data type:	Byte
Number of values per data element:	1
Number of computational dimensions:	3
Computational dimensions:	256 x 256 x 100
Number of computational values:	1 * 256 * 256 * 100 = 6553600
Mapping type:	Uniform
Number of coordinate dimensions:	3
Number of coordinate values:	0

Example 6

A fluid dynamics application is a three-dimensional simulation of fluid flow through a nozzle. Each data element has five values: a three-component velocity vector, temperature, and density. The field consists of 576 elements, each with five values:

$$\{V_n(X(i,j,k), Y(i,j,k), Z(i,j,k)), i = 1, 12, j = 1, 12, k = 1, 6, n = 1, 5\}$$

The field is irregular, with 576 *X* coordinates, 576 *Y* coordinates, and 576 *Z* coordinates. Many of the cells in coordinate space have curvilinear bounds. Figure 4 shows the mapping between computational and coordinate space.

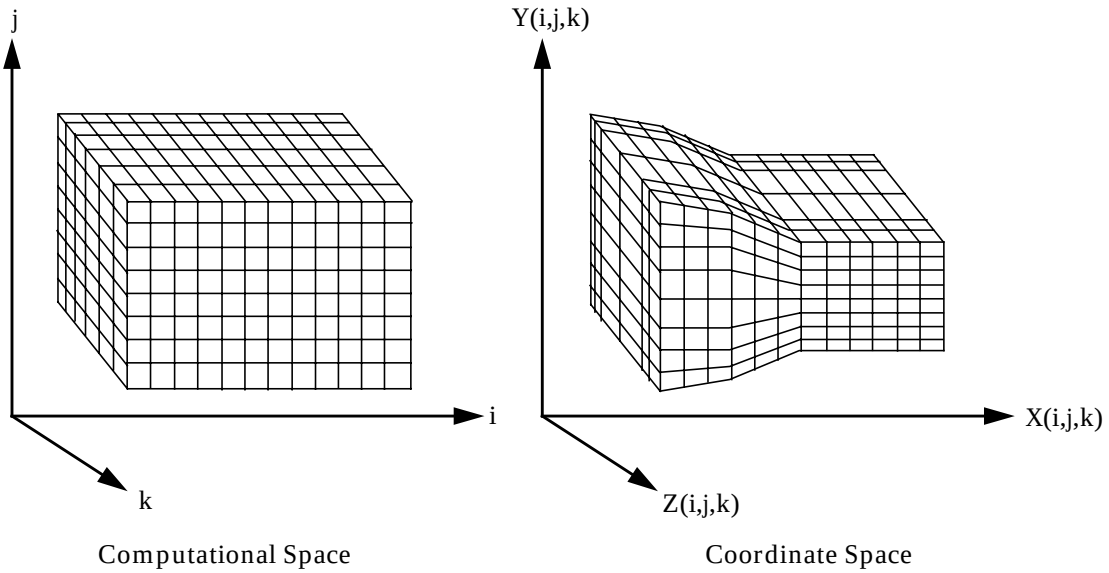

Figure 2-5 3D Computational, 3D Coordinate Irregular Field

The following is a summary of the field characteristics:

Data type:	Floating-point
Number of values per data element:	5
Number of computational dimensions:	3
Computational dimensions:	12 x 8 x 6
Number of computational values:	5 * 12 * 8 * 6 = 2880
Mapping type:	Irregular
Number of coordinate dimensions:	3
Number of coordinate values:	3 * 12 * 8 * 6 = 1728

Field Components

As represented in AVS, a field has the following components:

- The number of dimensions in computational space. This is an integer.
- The dimensions in computational space. This is an array of integers whose length is the number of dimensions in computational space. Each element of the array is the number of data elements along the corresponding dimension of computational space.
- The number of variables or values for each data element. This is an integer. A field with one value for each data element is a *scalar* field. A field with more than one value for each data element is a *vector* field. A field

can also consist only of coordinates, with no values for each data element; in this case the field represents a list of points in coordinate space.

- The data type of each value for the data elements. This is an integer. The data type can be unsigned character (byte), integer, single-precision floating-point, or double-precision floating-point. AVS defines a constant to represent each data type: **AVS_TYPE_BYTE**, **AVS_TYPE_INTEGER**, **AVS_TYPE_REAL**, and **AVS_TYPE_DOUBLE**. These constants are defined in the include files *<avs/avs.h>* for C programs and *<avs/avs.inc>* for FORTRAN programs.
- MIN / MAX information for computational data elements. The minimum values for each variable in the array of data elements are stored in an array whose data type is the same as that of the data elements. The size of this array is equal to the vector length of the field. The maximum values for each variable in the array of data elements are also stored in an array whose data type is the same as that of the data elements. The size of this array is equal to the vector length of the field.
- MIN / MAX extents for coordinates in each dimension of n-space. The minimum extent is an array of floating-point numbers, with a size equal to the number of dimensions in coordinate space. The maximum extent is also an array of floating-point numbers with a size equal to the number of dimensions in coordinate space.
- Labeling information for each vector element in the array of computational data. The labels are stored in a character array with a delimiter character as the first character in the array. The delimiter is followed by string / delimiter pairs. The number of pairs is equal to the vector length of the field. The labels are useful for defining what each variable in the array of data elements is. For instance one variable might be temperature, a second one might be pressure and a third might be density.
- The unit label associated with each vector element in the array of computational data. This is a character array with a delimiter character as the first character in the array. The delimiter is followed by string / delimiter pairs. The number of pairs is equal to the vector length of the field. The unit labels are useful for defining measurement units for each variable in the array of data. For instance one variable unit might be degrees centigrade and another might be pounds per square inch.
- The array of data elements representing the computational space of the field. Each element of the array is a value for a data element of the field. For a vector field, this array has one more dimension than the number of dimensions in computational space; the extra array dimension is the number of values per data element. The size of the array is the product of each dimension in computational space and the number of values per data element. The elements of the array are stored in "FORTRAN" order, with all values for each data element kept together. The array subscript for the value per data element varies fastest, followed by the subscript for the first dimension, the subscript for the second dimension, and so on. If *n_value* is the subscript for the value per data element and *i, j,* and *k* are the subscripts for the first, second, and third dimensions, respectively, the array is accessed in C as follows:

  ```
  data[k][j][i][n_value]
  ```

  The same array is accessed in FORTRAN as follows:

```
DATA(N_VALUE, I, J, K)
```

AVS has a number of macros to make access to this array more convenient for C language programmers. See Appendix B "AVS C Language Field Macros".

- A flag indicating the type of mapping from computational space to coordinate space. This is an integer, one of the following constants: **UNIFORM**, **RECTILINEAR**, or **IRREGULAR**. These constants are defined in the include files *<avs/field.h>* for C programs and *<avs/avs.inc>* for FORTRAN programs.
- The number of dimensions in coordinate space. This is an integer. For a uniform or rectilinear field, this is the same as the number of dimensions in computational space. For an irregular field, this can differ from the number of dimensions in computational space.
- For a **UNIFORM**, **RECTILINEAR**, or **IRREGULAR** field, an array of floating-point values representing the coordinates of the field. (The term **points** is used interchangeably with the term **coordinates** throughout the documentation.)

  For a **UNIFORM** field, coordinate information is limited to minimum and maximum extent fullword values for each physical dimension (n-space) of the data. The minimum and maximum extent values in the coordinate binary area are copies of the **min_ext** and **max_ext** values in the field data structure, *except* when the field has been cropped, downsized, or interpolated. Then the field data structure contains the original field's **min_ext** and **max_ext** values, while the coordinate section of the binary area contains the minimum and maximum extent of the subsetted data. Mapper modules can use this additional extent information to properly locate their geometric representation of the subsetted data in world coordinate space. The extents in the coordinate binary area are stored in the following order: minimum X, maximum X, minimum Y, maximum Y, minimum Z, maximum Z.

  For a **RECTILINEAR** field, this array contains one *X* value for each subscript along the first dimension of computational space, one *Y* value for each subscript along the second dimension of computational space, and so on. The coordinate array has one dimension, and the size of the array is the sum of the dimensions in computational space. All the *X* coordinates corresponding to the first dimension of computational space are stored first; all the *Y* coordinates corresponding to the second dimension of computational space are stored second; and so on. If *i, j,* and *k* are the subscripts for the first, second, and third dimensions of computational space, and if *idim1, idim2,* and *idim3* are the first, second, and third dimensions of computational space, the *X, Y,* and *Z* coordinates are obtained in C as follows:

  ```
  x = coords[i]
  y = coords[idim1 + j]
  z = coords[idim1 + idim2 + k]
  ```

  The coordinates are obtained in FORTRAN as follows:

  ```
  X = COORDS(I)
  Y = COORDS(IDIM1 + J)
  Z = COORDS(IDIM1 + IDIM2 + K)
  ```

For an **IRREGULAR** field, this array contains a set of coordinates (*X*, *Y*, and so on) for each data element in computational space. The coordinate array has one more dimension than the number of dimensions in computational space; the extra array dimension is the number of dimensions in coordinate space. The size of the array is the product of each dimension in computational space and the number of dimensions in coordinate space. All the *X* coordinates are stored first, then all the *Y* coordinates, and so on. The subscript for the first dimension of computational space varies fastest, followed by the subscript for the second dimension of computational space, and so on. The subscript for the dimension of coordinate space (*X*, *Y*, and so on) varies most slowly. If *n_coord* is the subscript for the dimension of coordinate space and *i*, *j*, and *k* are the subscripts for the first, second, and third dimensions of computational space, the array is accessed in C as follows:

```
coords[n_coord][k][j][i]
```

The same array is accessed in FORTRAN as follows:

```
COORDS(I, J, K, N_COORD)
```

AVS has a number of macros to make access to this array more convenient for C language programmers. See Appendix B "AVS C Language Field Macros".

- A unique, integer mesh_id identifier for a field representing the coordinate mesh of the field. Fields have two main components: the data itself and the X,Y,[Z] grid of coordinates in space at which the data exists. This grid of coordinates is also called the *mesh*.

  Often, when processed by an AVS network, the field data entering a mapper module will change (for example, a change in an **extract scalar** parameter, or in the **field legend** settings) while the mesh remains the same.

  Prior to AVS 4, a change in the data would cause the module to recompute internal data structures related to the mesh such as the block table, or geometries produced from the mesh, even though the grid of coordinates had not changed.

  In AVS 4, a new element has been added to the field data structure: the *mesh_id*. A module can assign a unique mesh_id to a particular field data structure. It changes the mesh_id only when it has changed the mesh. Downstream modules can compare the mesh_ids of incoming field data with that of the previous input. If the mesh_id is the same, it can elect to not recompute values related to the coordinate mesh. This can substantially improve performance.

Declaring Fields

When declaring or allocating fields, a programmer uses a field type string. This string consists of the word "field" followed by words describing each of the ways in which the field is specialized, such as "field 3D scalar uniform float". When declaring input and output ports (with **AVSadd_input_port** or **AVSadd_output_port**), you can leave out particular specifications to indicate that your module can accept or produce a more general data type. For exam-

ple, a module writer can declare an input port as accepting "field scalar" to indicate that that module accepts any type of scalar field.

The AVS flow executive does not permit a user to connect a module's output to another module's input if the output and input are declared to be conflicting types of fields. For example, AVS does not allow a "field 2D" output to be connected to a "field 3D" input. However, AVS does allow an output and an input to be connected if one is a subtype of another. For example, AVS allows a "field" output to be connected to a "field 2D" input.

The flow executive will not allow incompatible fields to be passed to a module. If you declare an input port as accepting a field of type: "field scalar uniform float", but the upstream module outputs a field of type "field 2D scalar uniform integer", the flow-executive will generate an error and not execute your module.

In rare situations, you might have to check if the data type description is not specific enough. If your data type description is: "field" but you really only wanted 2D or 3D fields (and couldn't handle 1D for example) your module should check to ensure that a field of the appropriate dimension was received.

In a field declaration, the word "field" is mandatory and is always the first word in the string. Specializing words are optional and can appear in any order. The following table lists possible specializing words:

Table 2-3 Field Declarations

Field Component	Value	Specializing Words
Number of Dimensions	*n*	"*n*D"
Vector Length	1 *n*	"scalar", "1-vector" "*n*-vector"
Data Type	byte integer real double	"byte", "char" "integer", "int" "real", "float" "double", "real*8"
Number of Coord Dims	*n*	"*n*-coord", "*n*-space"
Mapping Type	uniform rectilinear irregular	"uniform" "rectilinear" "irregular"

For the number of dimensions of coordinate space, any string beginning with "*n*-coord" is acceptable. For example, AVS recognizes "*n*-coords", "*n*-coordinate", and "*n*-coordinates".

Manipulating Fields from C

When a C language module has declared an input port, output port, or parameter to be a field, the computation routine is called with one argument corresponding to each field. If the field is an input port or parameter argu-

ment, the subroutine parameter is declared as **AVSfield ***. If the field is an output port, the subroutine parameter is declared as **AVSfield ****.

The type **AVSfield** is a structure defined in *<avs/field.h>*. Actually, there are four different kinds of field, one for each of the data types that fields support:

Field Type	Data Type
AVSfield_char	Byte
AVSfield_int	Integer
AVSfield_float	Real
AVSfield_double	Double

The only difference between these types is the type declaration for the data array. For the generic type **AVSfield**, the data is defined to be a union. See *<avs/field.h>* for more information.

An **AVSfield** structure is laid out as follows (using **AVSfield_float** as an example):

```
typedef struct {
  int ndim;               /* no. of computational dimensions */
  int nspace;             /* no. of coordinate dimensions */
  int veclen;             /* no. of values per data element */
  int type;               /* data type */
  int size;               /* size of each value in data element */
  int single_block;       /* internal, type of memory allocation */
  int uniform;            /* mapping type: Uniform, Rectilenear, or Irreg. */
  int flags;              /* data validity flags */
  int *dimensions;        /* dimension along each axis; length is ndim */
  float *points;          /* coordinates for fields */
  float *data;            /* the field data itself as floats */
  float *min_extent;      /* range of the data, array size is nspace */
  float *max_extent;      /* range of the data, array size is nspace */
  char  *labels;          /* labels for each value in a data element */
  float *minimum;        /* min data values for each value in a data element */
  float *maximum;        /* max data values for each value in a data element */
  int shm_key;            /* internal, shared memory key */
  int shm_id;             /* internal, shared memory identifier */
  char *shm_base;         /* internal, shared memory base address */
  char *units;            /* units for each component */
  int  shm_size;          /* internal, shared memory segment size */
  int  mesh_id;           /* unique id for the "points" information */
} AVSfield_float;
```

To illustrate the relation between field declarations and elements of the field structure, we use the example of a field representing fluid flow through a nozzle. The field has three dimensions in computational space, 12 x 8 x 6. Each data element has five floating-point values. The field is irregular with a three-dimensional coordinate space. The declaration for that field is as follows:

```
"field 3D 5-vector real 3-coordinate irregular"
```

The corresponding members of the **AVSfield** structure and their values are as follows:

```
ndim              3
nspace            3
veclen            5
type              AVS_TYPE_REAL
size              sizeof(float)
single_block      true if field is single malloc
uniform           IRREGULAR
dimensions        dims[3] = { 12, 8, 6 }
points            coords[3][6][8][12]
data              data[6][8][12][5]
min_extent        min extent of coords in each dim
max_extent        max extent of coords in each dim
labels            labels for each component
minimum           min data value for each component
maximum           max data value for each component
shm_key           shared memory key
shm_id            shared memory identifier
shm_base          shared memory base address
units             units of each component in data
mesh_id           unique id for the "points" information
```

The include file <*avs/field.h*> defines preprocessor macros to help C programmers gain access to the components of a field, including the dimensions in computational space, the data array, and the coordinate array. See Appendix B "AVS C Language Field Macros" for more information.

Manipulating Fields from FORTRAN

The preferred mode of accessing a field input or output port in FORTRAN is to pass the module's computation function a single integer argument for each field, rather than using the older method of passing several arguments. However, the module writer must specifically request the single integer argument mode by adding the following call to the description function for the module:

```
CALL AVSSET_MODULE_FLAGS( SINGLE_ARG_DATA | other flags )
```

The module's computation function receives a single integer argument that is a pointer to the field, rather than having the components of the field passed as multiple arguments. This field pointer value can then be passed directly to field accessor functions (e.g., **AVSfield_get_minmax**) in order to access any desired field element. When using the old multiple argument passing technique, in order to access field elements that are new in AVS3 (*min_extent, max_extent, labels, minimum, maximum*), it is necessary to call the routine **AVSport_field** in order to retrieve the field pointer required by the field accessor functions. The field accessor functions can then be used to retrieve any desired value from the field.

The FORTRAN module then accesses field structures using accessor functions on the single argument rather than by directly accessing the structure as a C module does. For both input and output fields, the integer argument is actually a pointer to a field pointer. This is unlike C which declares input fields and

output fields differently. For example, a computation routine that takes as its first input port a "field 3D 3-vector real rectilinear" (or any field) is defined as

```
FUNCTION COMPUTE(INFIELD, ...)
INTEGER INFIELD
```

Most of the accessor functions either return the requested information or the information is copied into an array passed in by the FORTRAN routine. For instance, instead of referencing *infield->ndim* the FORTRAN routine would call **AVSfield_get_int**:

```
LOCAL_NDIM = AVSFIELD_GET_INT(INFIELD, AVS_FIELD_NDIM)
```

The include file *<avs/avs.inc>* includes the necessary function declarations and accessor constants. Those field arrays which are of predictable size, such as the dimensions array, are filled directly by the accessor functions and it is incumbent upon the FORTRAN module writer to ensure that the arrays that are passed in are large enough for the maximum expected dimensions. Examples of using accessor functions such as **AVSfield_get_int** are provided in the program */usr/avs/examples/test_fld2_f.f*.

Accessing either the data or points array in a field is a little more involved since the arrays are arbitrarily large. There are two approaches to accessing each array. The first approach returns an offset index *N* between a local FORTRAN array and the actual field data array. The *N+1_th* element of the local FORTRAN array is the same as the first element of the desired array. This element reference can then be passed into a second function which declares it to be an array of a particular type and dimensionality. This approach is a little awkward but is generally portable. An example of using this technique is provided in the program */usr/avs/examples/test_fld2_f.f*. The appropriate library routines are **AVSfield_data_offset** and **AVSfield_points_offset**.

The second approach is to use the **AVSfield_data_ptr** and **AVSfield_points_ptr** routines to retrieve the data pointer as an integer from the field structure. Then pass the **%VAL()** of this integer to a second FORTRAN function which can then declare an array of the anticipated type and dimensions. This is easier, but less portable, than the first technique since some FORTRAN compilers support **%VAL**, others **%LOC**, and some may not support this non-ANSI FORTRAN feature.

Creating Fields

For allocating and freeing field structures, AVS provides several routines that are accessible from both C and FORTRAN. These routines ensure that a field is created which is internally self consistent (e.g., if it contains a 20 x 30 2D computational array the appropriate data and points arrays are automatically allocated) and which takes advantage of shared memory storage when possible. For instance, to create a "field 3D 3-vector real rectilinear" of size 20 x 20 x 20 make the following call in C:

```
output_field=AVSdata_alloc("field 3D 3-vector real rectilinear",dims)
```

where *dims* is a 3 element integer array containing 20,20,20. If an existing field is available as a template, **AVSfield_alloc** may be called to make a duplicate.

Before allocating new data, modules must free the data left over from their previous invocation. Failure to do this will eventually consume all available memory, shared memory segments, and swap space, causing the module (and perhaps AVS) to die. Fields may be freed using **AVSdata_free** or **AVSfield_free**:

```
AVSdata_free("field", output_field);
```

or

```
AVSfield_free(output_field);
```

Examples of creating fields may be found in the directory */usr/avs/examples*. See especially *read_image.c*, *read_vol.c*, *threshold.c*, and *test_fld2_f.f*.

Scatter Data

A *scatter* is a list of points in coordinate space with an optional scalar or vector data element for each point. AVS represents scatters as 1D irregular fields. For example, a scatter with scalar real data and 3D coordinates would be declared as a "field 1D scalar real 3-coordinate irregular". The one dimension of the field in computational space is the number of points in the scatter. The length of the data array is the product of the number of points in the scatter and the number of values per data element at each point.

A module can declare a scatter to have no data by declaring the vector length to be 0. For example, a scatter with no data and 3D coordinates is declared as "field 1D 0-vector 3-coordinate irregular". Such a field has no data array. The number of dimensions is declared as 0, and the one dimension of the field in computational space is the number of points in the scatter. This dimension is necessary to calculate the length of the coordinate array.

Image Data

AVS generally represents two-dimensional images as 2D uniform vector fields. Each vector contains four elements of byte data, and each byte represents one component of a pixel value. Thus, an image is usually declared as a "field 2D 4-vector byte". The following table shows which vector element corresponds to each component of the pixel value. The table is zero-based, as in a C language vector; in FORTRAN the vector index is one-based. For portability of modules to machines with different byte/integer organization, it is important that images be treated as "byte" arrays rather than "integer" arrays.

Byte	**Component**
0	alpha
1	red

2 green
3 blue

The alpha byte is not used in determining color; some modules use it to convey other information, such as opacity.

You can find examples of how to creating a 2D uniform vector field for use as an image in */usr/avs/examples/read_image.c* and */usr/avs/examples/read_image_f.f*.

Volume Data

AVS represents some volumes as 3D scalar fields of bytes, usually declared as "field 3D scalar uniform byte". The value of each byte is between 0 and 255 inclusive. Some modules use the field data as indices into colormaps. For the **read volume** module each dimension of the field must be less than 256.

You can find examples of how to create a 3D scalar uniform byte field for use as a volume in */usr/avs/examples/read_vol.c* and */usr/avs/examples/read_vol_f.f*.

Colormaps

A colormap is a data structure that implements a transfer function that assigns a color to each value between an upper and a lower bound. A colormap consists of four arrays of floating-point values, one each for hue, saturation, value, and opacity. Each value is between 0.0 and 1.0 inclusive. A colormap also has an integer size or number of colors, which is the length of each of the four arrays. A colormap has floating-point lower and upper bounds that determine the resolution of the colormap. The lower bound is an index that maps to the first element of each array. The upper bound is an index that maps to the last element in each array.

In C, a colormap is represented by an **AVScolormap** structure, defined in *<avs/colormap.h>* as follows:

```
typedef struct  {
  int size;           /* number of entries in each array */
  float lower;        /* 0th entry maps to this value */
  float upper;        /* size-th entry maps to this value */
  float *hue;
  float *saturation;
  float *value;
  float *alpha;
} AVScolormap;
```

A C routine declares a colormap input argument as **AVScolormap *** and a colormap output argument as **AVScolormap ****.

A FORTRAN computation routine can input a colormap by passing a single argument, which is an integer colormap id, and use accessor functions to ac-

cess the contents of the colormap. To do this, a module must set the SINGLE_-ARG_DATA module flag which tells AVS to pass both colormaps and fields as single arguments:

```
AVSset_module_flags(single_arg_data)
```

Use the **AVScolormap_get** and **AVScolormap_set** routines to access the contents of the colormap. In either C or FORTRAN, a new colormap can be created using **AVSdata_alloc** as in:

```
colormap_out = AVSdata_alloc("colormap", dimensions)
```

where dimensions is a one element integer array with the colormap size as the first element of the array.

If the module flags are **not** set to single-arg-data, a FORTRAN computation routine inputs a colormap by declaring a series of parameters:

```
INTEGER FUNCTION my_module(size,lower,upper,hue,sat,val,alpha)
      INTEGER size
      REAL lower, upper
      REAL hue(size), sat(size), val(size), alpha(size)
```

Using this older approach, a FORTRAN routine outputs a colormap as follows. Note the use of POINTER variables to supply an extra level of indirection:

```
INTEGER FUNCTION my_module(size, lower, upper, phue, psat, pval, palpha)
      POINTER (phue,hue), (psat,sat), (pval,val), (palpha,alpha)
      REAL hue(size), sat(size), val(size), alpha(size)
```

FORTRAN programmers can also use the more portable **AVSptr_offset** function to return an offset index between the colormap array and a local reference array when the multiple argument approach is used.

An example of using colormap input within a FORTRAN module using both of these approaches is provided in */usr/avs/examples/colorizer_f.f*.

Geometries

AVS passes geometric information between modules by using a data structure called an *edit list*. The edit list describes changes to the geometry of a particular scene. Generally a user module sends edit lists as outputs. It is possible for a module to use edit lists as inputs, but AVS3 does not support routines to extract geometric information from an edit list. Geometry output is typically used as input to an AVS-supplied renderer module such as the geometry viewer.

A geometry data object must be inserted into an edit list in order to be passed along via an output port. The edit list can also contain an arbitrarily long list of changes to be made in the current scene. Each change pertains to a particular object, camera, or light source. Changes are made in the order specified in

the edit list. The AVS data type for an edit list is **GEOMedit_list**. A C language module computation routine declares an argument representing an input port or parameter as **GEOMedit_list** and an argument representing an output port as **GEOMedit_list *** (note the single asterisk). In FORTRAN both kinds of argument are declared as **INTEGER**.

Each object, camera, or light is referred to by a name that is an ASCII string. By default, an object name is modified by the port through which it is communicated. This prevents two different modules from modifying each other's objects. For example, two "arbitrary slice" modules would each try to modify the data for the object named "arbitrary slice". Since the name is modified by the port, the first arbitrary slice module modifies "arbitrary slice.0", and the second modifies "arbitrary slice.1". When it is desirable for a module to use the absolute name of an object, it can precede the object name by a "%" character (e.g., "%arbitrary slice").

AVS creates any object that doesn't already exist the first time an attempt is made to change that particular object.

Camera names are ASCII strings of the form: **camera***n*, where *n* ranges from 1 to the number of views on the particular scene.

Light names are ASCII strings of the form **light***n*, where *n* ranges from 1 to 16.

AVS has routines that allow a module to change several properties of an object in an edit list:

- Geometric data defining the object
- Surface or line color
- Render mode (Gouraud, Phong, wireframe, etc.)
- Parent (the name of the parent object)
- Object material properties
- Object, camera, and light transformation
- Object visibility, deletion
- Object color, light source color and camera background color
- Camera background color
- Light source on / off, type
- Texture mapping
- Transformation mode (controls how objects are transformed)
- Selection mode (controls how objects are picked)
- Center of rotation and scaling
- Viewable region of data
- Viewing projection

Manipulating Edit Lists

When a module is invoked, it typically initializes the edit list from the previous execution. This both frees the data from the previous run and creates an empty edit list for use on the current run. The module places into the edit list, changes that it wants to make for this invocation. A module uses routines in the geom library to create and use edit lists, geometry objects, and light sources. See the "Geometry Library" appendix for more information.

A module typically uses the following steps in preparing an edit list for output:

- Initialize the edit list, using **GEOMinit_edit_list** in C or **GEOM_INIT_EDIT_LIST** in FORTRAN. This creates a new list or empties an existing list.
- Create and/or modify geometry objects, cameras, or lights sources, using routines in the geom library.
- Modify the edit list, using routines whose names begin with **GEOMedit** in C or **GEOM_EDIT** in FORTRAN (such as **GEOMedit_geometry** or **GEOM_EDIT_GEOMETRY**).
- For a coroutine module, use **AVScorout_output** to output the list, and then use **GEOMdestroy_edit_list** in C or **GEOM_DESTROY_EDIT_LIST** in FORTRAN to deallocate the list.

A module must deallocate an existing edit list before reusing the list. For a subroutine module, the edit list passed to the module as an output argument is the edit list the module created on its last execution. The module must deallocate this list at the start of each invocation of the module, normally by calling the **GEOMinit_edit_list** routine in C or **GEOM_INIT_EDIT_LIST** in FORTRAN before modifying the list:

```
/* C */
my_module(output)
GEOMedit_list *output;
{
  /*
   * Deallocate edit list from last invocation;
   * initialize edit list for this invocation.
   */
  *output = GEOMinit_edit_list(*output);

  < rest of module >
}

C     FORTRAN
      FUNCTION MY_MODULE(OUTPUT)
      EXTERNAL GEOM_INIT_EDIT_LIST
      INTEGER OUTPUT, GEOM_INIT_EDIT_LIST
      OUTPUT = GEOM_INIT_EDIT_LIST(OUTPUT)

      < rest of module >
```

A coroutine module can use **GEOMdestroy_edit_list** in C or **GEOM_DESTROY_EDIT_LIST** in FORTRAN to deallocate a list after calling **AVScorout_output**:

```
/* C */
  ...
  GEOMedit_list output;

  < generate edit list "output" >

  AVScorout_output(output);
  GEOMdestroy_edit_list(output);

C     FORTRAN
      ...
      INTEGER OUTPUT

      < generate edit list "OUTPUT" >

      CALL AVSCOROUT_OUTPUT(OUTPUT)
      CALL GEOM_DESTROY_EDIT_LIST(OUTPUT)
```

You can find examples of manipulating geometry edit lists in the directory */usr/avs/examples*. The programs *polygon.c* and *polygon_f.f* are subroutine modules; *qix.c* and *qix_f.f* are coroutine modules.

Templates for New Filter Utilities

AVS provides several C-language and FORTRAN-language templates for those who wish to write their own filter utilities. (If your data format is simple enough, you may be able to use one of the templates without modifying it. The mesh format, in particular, can often be used without modification.)

Each template handles a particular type of object defined in the Geometry Library. Table 2-4lists the AVS-supplied filter templates. Each one reads a file from *stdin*, writes a file to *stdout*, and accepts no command-line options.

Table 2-4 Template for Filter Utilities

Source Filename(s)	Executable Filename	Object Type
mesh.c, mesh.f	mesh_to_geom	Mesh
polygon.c, polygon.f	polyg_to_geom	Disjoint polygon
polyh.c	polyh_to_geom	Polyhedron
sphere.c	sphere_to_geom	Sphere

The filters are all located in directory */usr/avs/filter*.

Writing a New Filter Utility

This section provides pointers for those who wish to create new filter utilities, using the template programs listed in the table above.

The basic procedure for creating a *geom*-format object is:

1. Decide which of the *geom*-format objects conforms most closely to the application data:

   Polyhedron
   A list of vertices with an indirect list of pointers into these vertices for each polygon.

   Polygon
   A list of vertices for each polygon.

   Mesh
   A 2D array of values, either scalars (for a height field) or vertices.

   Sphere
   A list of center points and radii.

   Polytriangle
   A single list of vertices representing polylines, disjoint lines, or a triangle mesh, where the connectivity is implied by the particular data type.

   Note that no tools exist for direct conversion of non-linear geometries, such as spline surfaces and quadrics.
2. Create an instance of that *geom*-format.
3. Perform any necessary processing on the object, such as generating normals.
4. If necessary, convert this object to an optimized-format object, such as a polytriangle.
5. Write the object to a file.

The Geometry Library contains routines that help with these tasks.

The following sections describe the steps for converting a variety of object types to *geom* format.

Converting a Polyhedron

Start with the template *polyh.c*, then:

- Create a polyhedron object.
- Add vertices.
- Add a list of polygons (as a list of pointers).
- Generate normals (if necessary).
- Convert to polytriangle object — both wireframe and surface descriptions.

Converting a Polygon

Start with the template *polygon.c* or *polygon.f*, then:

- Create a polyhedron object.
- Add disjoint polygons (either faceted or smooth).
- Generate normals (if necessary).
- Convert to polytriangle object — both wireframe and surface descriptions.

Converting a Scalar Mesh

Start with the template *mesh.c* or *mesh.f*, then:

- Create a mesh from a list of scalars.
- Generate normals (if necessary).
- Convert to polytriangle object — both wireframe and surface descriptions.

Converting a Mesh

Start with the template *mesh.c* or *mesh.f*, then:

- Create a mesh from the vertices.
- Generate normals (if necessary).
- Convert to polytriangle object — both wireframe and surface descriptions.

Converting a Sphere

Start with the template *sphere.c*, then:

- Create a sphere object from the sphere centers and radii.

Converting a Disjoint Line

There is no starting template for this case. You should do the following:

- Create a polytriangle object.
- Add disjoint lines to this object.

Converting a Polyline

There is no starting template for this case. You should do the following:

- Create polytriangle object.
- Add zero or more polylines to this object.

Pixel Maps

A pixel map is a data structure that incorporates a reference to an X Window System pixmap. An X pixmap is an array of pixel values that can be a destination for a rendered image. It resides in the X server. (In contrast, an image is a data structure that includes an array of colors and resides in client memory.) A pixel value can be a colormap index on a pseudo color system.

A pixel map data structure includes an Xlib **Pixmap** id, the Xlib **Window** id of the window associated with the pixmap, the **Window** id of that window's parent window, and other information which is dependent on extensions to the X-Window Server.

In C, a pixel map is defined as an **AVSpixdata** data type. A pixel map input argument is declared as **AVSpixdata ***, and a pixel map output argument is declared as **AVSpixdata ****. **AVSpixdata** is a structure defined in <*avs/avs_pixdata.h*> with the following components:

```
typedef struct _AVSpixdata {
  int parent;
  int window;
  int pixmap;
  int is_buffer; /* 1 if pixmap is from the render geometry module */
} AVSpixdata;
```

A FORTRAN computation routine cannot take a pixel map as an argument.

Because pixel maps rely heavily on specific hardware and software features, they are not very portable or easy to use. Programmers should not try to use pixel maps to perform image processing; the "image" field type ("field 2D 4-vector byte uniform") is more portable and interfaces to a wider variety of other modules.

Unstructured Cell Data

The Unstructured Cell Data (UCD) type provides a way to aggregate 3D primitive objects and associated data into a single data structure. The 3D objects do not have to be connected, i.e., they are not required to share nodes or define a surface. UCDs are useful to represent volume information that is not structured enough to be represented as a field data type.

The use of unstructured cell data is detailed in Appendix E. An example of creating a UCD data structure from a definition in a file is provided in */usr/avs/examples/read_ucd.c*. An example of creating a UCD data structure based on user parameter input is provided in */usr/avs/examples/gen_ucd.f*. Examples of manipulating UCD data structures are found in */usr/avs/examples/ucd_extract.c* and *ucd_thresh.c*.

Molecular Data Type

In order to better address the needs of the chemistry community, a Molecule Data Type has been added to AVS. This data type addresses the general needs of classical, substructure and quantum chemistry fields.

The use of the Molecular Data Type is detailed in the *Chemistry Developer's Guide*. Module source examples working with this data type can be found in the directory */usr/avs/examples/chemistry*.

User-Defined Data Types

AVS provides limited capabilities for users to implement their own data types. There are also two standard AVS data types that are defined using this mechanism. User-defined data types may be useful for problems that are best defined using data structures that are different from those built into AVS. However, existing modules are unlikely to be able to deal directly with the new data type; the user has to convert to a more standard type eventually (such as "field" or "geometry") or simply not use existing modules.

The user-defined data type may also be useful for sending a subset of data back "upstream" in a network to feed back information to a module that is sending data "downstream". The two module must both recognize the data type defined for this purpose.

Chapter 4 discusses in detail upstream feedback and the declaration and use of user-defined data types. For an example module that uses upstream feedback and a user-defined data type, see the sample programs, *pick_cube.c*, *user_data.c*, and *user_data_f.f* in the */usr/avs/examples* directory.

CHAPTER 3

AVS MODULES

Modules

A module is the fundamental building block in an AVS network. A module typically has one of three purposes:

- To import data from outside AVS (or generate its own data) and convert it into data of one of the AVS data types.
- To transform AVS data in some way, producing output data of the same or of a different AVS type.
- To render or store AVS data on an external device, such as the display screen or a file.

AVS has a library of modules that perform these tasks for many types of data. This chapter describes how to write your own modules. To simplify the process of writing new modules a **Module Generator** has been provided. Documentation on the **Module Generator** can be found in the *Applications Guide.*

Module Components

This section describes the anatomy of AVS modules.

Name

The name of a module is a string that identifies the module to the user. The name appears on the module icon in the module palette and in the workspace.

Type

A module is of one of four types, depending on its function:

Data Input
A module that generates data or imports data from outside AVS and converts it into one of the AVS data types.

Filter
A module that transforms AVS data in some way, producing output data of the same or of a different AVS type.

Mapper
A module that converts AVS data to a "geometry" data type.

Data Output
A module that renders or stores AVS data, usually of the type geometry, on an external device, such as the display screen or a file.

These module type distinctions affect only the presentation of the module in the AVS user interface. The module type determines in which menu the module icon appears in the module palette.

Ports

A module may have zero or more *input ports* and zero or more *output ports*. A port is a channel through which data passes to or from other modules. Each port has a name and an AVS data type associated with it. An input port is represented in the Network Editor by a colored bar at the top of the module icon, and an output port is represented by a colored bar at the bottom of the icon. The color (or colors) of each bar indicate the port's data type.

Data modules usually read or generate their own data and therefore do not generally have input ports. Renderer modules often display or write their own output data and therefore do not generally have output ports.

When you instance a module in AVS (that is, move the module icon from the Network Editor module palette to the workspace), you connect each input port to an appropriate output port of another module, and connect each output port to an appropriate input port of another module. You can connect a pair of ports only when the data types of the ports match. The data types match when they are the same or when one is a subtype of the other. For example, a port declared to be of type "field" matches a port of type "field 2D", but a port of type "field 2D" does not match a port of type "field 3D". You cannot connect an output port to an input port of the same module.

Some input ports require a connection to an output port of another module before the module can be invoked (executed by AVS). For other input ports, a connection is optional. The module developer controls this using the **AVScreate_input_port** routine.

Parameters

A parameter is a variable that has a constant value during an invocation of a module. The AVS user can change the value of the parameter between module invocations by manipulating a user interface "widget" attached to the parameter. A widget is a virtual input device such as a dial or a file browser.

A parameter has a name, a type, and an initial value. Some parameters also have bounding information, such as a range of allowed values; AVS then ensures that the value of the parameter remains within the bounds. Parameter types include most primitive AVS data types along with constrained variants such as "boolean" and "choice". For information on parameter types, see the documentation for the **AVSadd_parameter** routine in Appendix A.

Each parameter is usually connected to a widget that enables the user to change the value of the parameter between module invocations. You can connect a parameter only to a widget that is compatible with the parameter's data type. Each parameter type has a default widget type, but the module can override the default and attach a parameter to another compatible widget. For information on the permissible widget types and the default widget type for each parameter type, see the documentation for the **AVSconnect_widget** routine in Appendix A

A parameter can also have *properties*. A property usually determines some aspect of how the associated widget presents the parameter. By setting properties on a parameter, a module can customize how the user interface handles the parameter. Each property is meaningful only with certain widgets. For a description of the available properties, see the documentation for the **AVSadd_parameter_prop** routine in Appendix A

A module can dynamically alter the current value or bounds of a parameter. AVS then updates any widget associated with the parameter. See the documentation for the **AVSmodify_parameter** routine in Appendix A for more information.

In some cases properties can be updated during computation using **AVSmodify_parameter_prop**. For instance, the default text shown for a boolean parameter could be changed to a new value based on labels in an input field. Some changes do not have a noticeable effect if the widget currently attached to the parameter can not accommodate the change.

Examples of defining parameters are provided in the example program */usr/avs/examples/widgets.c*.

Parameters As Input Ports

AVS makes a distinction between parameters and inputs. By default, a parameter is attached to a widget and input is received through a port. From the Network Editor, a user can turn any parameter into a port on that module (see

the *AVS User's Guide* for information on how this is done using the parameter edit capability of the Network Editor). Once a parameter has a port, it behaves very much like an input port. The only difference is that when a new value is generated for that port, the widget associated with that parameter (if any) is updated with the value. You can disconnect the widget from the module if this behavior is not desired. Assigning a port to a parameter allows the you to simultaneously feed a parameter value to multiple modules.

While it may appear that parameters are just a special form of input port, there are a couple of important differences:

- Parameter ports are invisible by default and there is no way to make them visible within the module code. The user makes them visible when invoking the widget that is associated with the parameter.
- Parameters do not accept any arbitrary data type. For example, modules cannot declare pixmaps as a parameter data type.

Functions

Each module has one or more functions associated with it. The module writer supplies these functions, and AVS invokes them at various times during the life of the module. The following list describes the basic kinds of functions found in a module and discusses the purpose of each:

- Each module has a *description* function. The description function identifies the module to AVS and declares its name, ports, and parameters. AVS invokes this procedure when it first learns about a module's availability and again when the user makes an instance of the module, by moving the module icon from the Network Editor module palette to the workspace.
- Each subroutine module has a *computation* function. This function does the computational work of the module, typically using the input data and parameters to produce output data. AVS invokes this function when the flow executive is active and when the module's input data or parameters change. The arguments to the computation function correspond to the module's input ports, output ports, and parameters.

  A coroutine module does not have a computation function; the module's main program itself determines when to perform its computation.
- A module may have an *initialization* function. The initialization function may take such actions as allocating memory or creating a window. AVS invokes this function when the user makes an instance of the module (by moving the module icon from the Network Editor module palette to the workspace). The initialization function is called before the Kernel has finished creating the module. Some functions will not work in this context, notably **AVScommand**. The initialization function has no arguments and returns no value.
- A module may have a *destruction* function. The destruction function may take such actions as freeing memory or destroying a window. AVS invokes this function when the user destroys the module (as by moving the

module icon from the Network Editor workspace to the "hammer" icon). The destruction function has no arguments and returns no value.

The Description Function

Using a set of library functions, the description function describes the module's name, type, inputs, outputs, and parameters. C and FORTRAN source files can contain more than one module and therefore more than one description function. The source file must contain a user-written routine named **AVSinit_modules** that declares all the description functions in the file. Within the **AVSinit_modules** routine, use the library function, **AVSmodule_from_desc**, to declare each module defined in the file. FORTRAN programmers can use the **AVSinit_modules** routine itself as the description function if there is just one module defined in the source file. The description function takes no arguments and returns no value.

The following is the C language version of a sample description function for a module that computes the threshold of a 3-dimensional scalar field. The threshold module has one input port, one output port, and two parameters.

```
void threshold()
{
  int thresh_compute();
  int in_port, out_port;

  AVSset_module_name("threshold", MODULE_FILTER);
  in_port = AVScreate_input_port("Input Field", "field 3D scalar",
                    REQUIRED);
  out_port = AVScreate_output_port("Output Field", "field 3D scalar");
  AVSinitialize_output(in_port, out_port);
  AVSadd_float_parameter("thresh_min", 0.0, FLOAT_UNBOUND,
                FLOAT_UNBOUND);
  AVSadd_float_parameter("thresh_max", 255.0, FLOAT_UNBOUND,
                FLOAT_UNBOUND);
  AVSset_compute_proc(thresh_compute);
}
```

The following is the FORTRAN version of the same routine:

```
      SUBROUTINE AVSINIT_MODULES
#include 'avs/avs.inc'
      EXTERNAL AVSCREATE_INPUT_PORT, AVSCREATE_OUTPUT_PORT
      INTEGER IN_PORT, AVSCREATE_INPUT_PORT
      INTEGER OUT_PORT, AVSCREATE_OUTPUT_PORT
      EXTERNAL THRESH_COMPUTE
      CALL AVSSET_MODULE_NAME('threshold', 'filter')
      IN_PORT = AVSCREATE_INPUT_PORT('Input Field',
     +     'field 3D scalar', REQUIRED)
      OUT_PORT = AVSCREATE_OUTPUT_PORT('Output Field',
     +     'field 3D scalar')
      CALL AVSINITIALIZE_OUTPUT(IN_PORT, OUT_PORT)
      CALL AVSADD_PARAMETER('thresh_min', 'real', 0.0,
     +     FLOAT_UNBOUND, FLOAT_UNBOUND)
      CALL AVSADD_PARAMETER('thresh_max', 'real', 255.0,
     +     FLOAT_UNBOUND, FLOAT_UNBOUND)
```

```
CALL AVSSET_COMPUTE_PROC(THRESH_COMPUTE)
RETURN
END
```

In general, description functions perform the following tasks:

- Set the module name and type using **AVSset_module_name**. A description function must call this routine.
- Create the input and output ports using **AVScreate_input_port** and **AVScreate_output_port**. A description function may have zero or more calls to each of these routines, depending on how many input and output ports it has. Each routine returns an integer port identifier for use as an argument to other routines, such as **AVSinitialize_output**.
- Create the parameters using **AVSadd_parameter** or **AVSadd_float_parameter**. A description function may have zero or more calls to each of these routines, depending on how many parameters it has. Each routine returns an integer parameter identifier for use as an argument to other routines, such as **AVSconnect_widget**.
- Set the computation function using **AVSset_compute_proc**. A description function for a subroutine module must call this routine. A description function for a coroutine module does not call this routine.
- Specify special treatment with **AVSset_module_flags** (for example, specifying **SINGLE_ARG_DATA** in order to receive field inputs or outputs as single arguments in FORTRAN).

A description function can also take the following optional steps:

- Use the **AVSinitialize_output** routine to tell AVS to preallocate memory for output data before invoking the module computation function. This routine pairs an output port with an input port. Before invoking the module computation function, AVS frees data at the output port and allocates a new data structure of the same size and dimensions as the data at the input port. This frees the computation routine from the necessity of allocating memory for the data structure.
- Use the **AVSautofree_output** routine to tell AVS to free memory allocated for output data before invoking the module computation function. By default, AVS does not free the memory allocated for output data during the previous invocation of the module computation function. **AVSautofree_output** and **AVSinitialize_output** are mutually exclusive. For futher information on memory management see the *Memory Allocation Debugging* section in Chapter Four.
- Set an initialization function using the **AVSset_init_proc** routine.
- Set a destruction function using the **AVSset_destroy_proc** routine.
- Use the **AVSconnect_widget** routine to declare a preference that a parameter be attached to a widget of a given type. Each type of parameter is associated with a default widget type. This routine allows the module to override the default.

  For example, a module can use a parameter of type "string" for a file pathname. The default widget for a string parameter is a text type-in. The

module description function can use **AVSconnect_widget** to connect the parameter to a file browser. The following is a C language example:

```
int p;
p = AVSadd_parameter("Data File", "string", "/mydata", "", "");
AVSconnect_widget(p, "browser");
```

The following is a FORTRAN example. Note that a space is required when specifying empty strings:

```
EXTERNAL AVSADD_PARAMETER
INTEGER P, AVSADD_PARAMETER
P = AVSADD_PARAMETER('Data File', 'string', '/mydata', ' ', ' ')
CALL AVSCONNECT_WIDGET(P, 'browser')
```

- Use the **AVSadd_parameter_prop** routine to add a property to a parameter. By calling this routine, a module can customize how the user interface handles the parameter.

The Computation Function

Each subroutine module must have a computation function in addition to a description function. AVS invokes the computation function when the flow executive is active and the module's inputs or parameters change.

The computation function can have any name. The module identifies the computation function to AVS by calling the **AVSset_compute_proc** routine in the description function. You must declare the computation function to return an integer. It should return a value of 0 to indicate an error and 1 to indicate success. In the case of an error, the flow executive does not invoke any other modules whose inputs depend on the erring module's outputs.

The arguments to the computation function correspond to the module's inputs, outputs, and parameters. A C language computation function has one argument for each input port, output port, and parameter declared in the description function. In the parameter list, all the input ports are represented first, then all the output ports, then all the parameters. Within each category, the arguments appear in the order in which the ports or parameters are declared in the description function.

For a FORTRAN computation function, the general ordering of ports and parameters is the same as in C. However, there are two alternatives for passing arguments. The default approach is to pass aggregate structures such as *fields* and *colormaps* as multiple arguments in order to gain direct access to each element of the structure. Another approach is to set the module flag (using **AVSset_module_flags**) to **single_arg_data**. This causes AVS to pass fields, colormaps, and user-defined data types as a single argument. The argument is actually a pointer to the data structure pointer itself, and can be used as an argument to language independent access routines. For more information on the use of **single_arg_data** and on declaring arguments to FORTRAN computation functions, see Chapter 2.

C language computation functions pass input port and parameter arguments as pointers to an object of the same C data type as the AVS data type declared in the description function for that port or parameter. An argument that represents an output port is usually passed as a pointer to a pointer to an object of the appropriate data type. This double indirection is provided to allow the computation routine to allocate memory for the output data. For example, a C language computation function declares an input field argument as **AVSfield *** and an output field argument as **AVSfield ****. Arguments that represent ports or parameters of some data types, such as integer, are passed as the objects themselves.

Because FORTRAN arguments are passed by reference, a FORTRAN computation routine usually declares an argument to be of the FORTRAN type that corresponds to the AVS data type of the port or parameter. For example, an argument that represents a floating-point input port, output port, or parameter is declared to be of type **REAL**.

The computation routine usually performs some operations on the input data and parameters to produce output data. By default, the computation function is responsible for freeing memory allocated for output data on previous invocations of the module and for allocating memory for output data on the current invocation.

Note: Failure to free the memory allocated on previous module invocations will eventually consume all available memory, shared memory segments, and swap space, causing the module (and perhaps AVS) to die.

Rather than using *malloc* and *free*, modules should call **AVSdata_alloc**, **AVSfield_alloc**, and **AVSfield_free** since these routines automatically make a field of the desired dimensions and free fields in an appropriate manner. The module can use the **AVSinitialize_output** and **AVSautofree_output** routines in the description function to eliminate the need for some of this memory management. For futher information on memory management see the "Memory Allocation Debugging" section in Chapter Four.

Initialization Function

If a module defines an initialization function, AVS invokes the it when the user instances the module (moves the icon from the Network Editor module palette to the workspace). An initialization function performs tasks like allocating memory or creating a window.

Use the **AVSset_init_proc** routine to declare the initialization routine from within the description function.

Destruction Function

If a module defines a destruction function, AVS invokes it when the user destroys a module (moves the module icon from the Network Editor workspace to the "hammer" icon). A destruction function performs tasks like freeing memory or destroying a window.

Subroutines and Coroutines

AVS has two types of modules: *subroutines* and *coroutines*. The chief difference between the two is the way they interact with AVS to do their computational work. In essence, a subroutine module does its computation whenever AVS asks it to, usually when the module's input ports or parameters change. A coroutine module does its computation whenever it wants.

Subroutines are the most common type of AVS module. They are used in the demand-driven portions of a network where a module needs to compute only when input data or a parameter has changed. Coroutine modules are typically simulations or animations. A coroutine usually performs a number of independent computations, each of which represents one iteration of a series, and sends output to AVS after each iteration. For example, the AVS particle advector module is a coroutine.

Subroutine Modules

A basic subroutine module as written by a programmer consists of a description function and a computation function, with optional initialization and destruction functions. The programmer does **not** supply a main program; instead, the AVS library supplies the main program for a module's executable file.

An executable file may contain more than one module, including description and computation functions for each module, but it has only one main program. In addition to the description and computation functions, the programmer supplies a function called **AVSinit_modules** to invoke the description functions for all modules in the file. This routine takes no arguments and returns no values. It must make one call to **AVSmodule_from_desc** for each module in the file. The **AVSinit_modules** routine can call **AVSmodule_from_desc** either directly for each module in the file or indirectly, for a list of modules, through a single call to **AVSinit_from_module_list**. **AVSmodule_from_desc** invokes the given module's description function. The following is a simple example of an **AVSinit_modules** routine for a file that contains a single threshold module:

```
AVSinit_modules()
{
  /* threshold is the module description function */
   int threshold();
  /* this invokes the threshold routine */
  AVSmodule_from_desc(threshold);}
```

The following is an example of an **AVSinit_modules** routine for a file that contains more than one module:

```
int ((*mod_list[])()) = {
  module_1_desc,
```

```
  module_2_desc,
  module_3_desc
};

#define NMODS (sizeof(mod_list) / sizeof(char *))

AVSinit_modules()
{
  AVSinit_from_module_list(mod_list, NMODS);
}
```

A FORTRAN source file that contains only one module does not need a separate **AVSINIT_MODULES** function; instead, its description function can itself be called **AVSINIT_MODULES**.

A FORTRAN file that contains more than one module must make multiple calls to **AVSMODULE_FROM_DESC** from within the **AVSINIT_MODULES** function in the same way as a C source file that contains more than one module.

AVS normally invokes the module's main program twice: once when the user reads the module into AVS, as by executing the **Read Module** Network Editor command, and once when the user makes an instance of the module, by moving the module icon from the Network Editor palette to the workspace. In both cases, AVS creates a new process and invokes the module executable file in that process.

When AVS invokes the module's main program the first time, it does so for *identification*. The module's main program then does the following:

- Sets up a connection to AVS.
- Invokes the **AVSinit_modules** routine. This routine in turn invokes the description functions of all modules in the executable file.
- Conveys to AVS the module declarations for all modules in the executable file.
- Terminates the module's process.

When AVS receives the module declarations, it adds the module icons to the Network Editor palette.

When AVS invokes the module's main program a second time, it does so for *instantiation*. The module's main program then does the following:

- Sets up a connection to AVS.
- Invokes the **AVSinit_modules** routine. This routine in turn invokes the description functions of all modules in the executable file.
- Conveys to AVS the module declarations for all modules in the executable file.
- Sets up an instance of the module that can receive data from and send data to AVS.
- Invokes the module initialization function, if one exists.

- Enters a server routine that loops indefinitely waiting for remote procedure calls from AVS, and then, executes these requests.

When the flow executive is active, AVS issues a remote procedure call whenever any of the module's input ports or parameters change. When the module's server routine receives a computation request, it reads the module's inputs and parameters from AVS, invokes the module's computation function, and conveys the module's outputs to AVS. If another module's input port is connected to the current module's output port, AVS marks the other module's input port as having changed data. This may cause AVS to send a remote procedure call to the second module.

AVS may issue remote procedure calls other than computation requests during the lifetime of the module. For example, the user may destroy the module by dragging the module icon to the "hammer" icon. AVS then issues a remote procedure call that causes the module server routine to invoke the module's destruction function, if one exists, and then terminates the module's process. The module's computation function may also issue callbacks to AVS, as when reporting errors via the **AVSmessage** routine.

Coroutine Modules

A basic coroutine module as written by a programmer consists of a main program and a description function, with optional initialization function. (Coroutines do not support destruction functions.) Each executable file can contain only one module. The description function can have any name.

As with subroutine modules, AVS normally invokes the coroutine module's main program twice: once when the user reads the module into AVS, as by executing the **Read Module** Network Editor command, and once when the user makes an instance of the module, as by moving the module icon from the Network Editor palette to the workspace. In both cases, AVS creates a new process and invokes the module executable file in that process.

When AVS invokes the module's main program the first time, it does so for "identification". Because AVS does not supply the main program, the programmer is responsible for ensuring that the main program responds properly to this invocation. The main program must call the **AVScorout_init** routine early on, before attempting to do any computation. The **AVScorout_init** routine does the following during the identification phase:

- Set up a connection to AVS.
- Invoke the module's description function.
- Convey to AVS the module declarations.
- Terminate the module's process.

When AVS receives the module declarations, it adds the module icon to the Network Editor palette.

When AVS invokes the module's main program a second time, it does so for *instantiation*. When the main program invokes **AVScorout_init** during the instantiation phase, that routine does the following:

- Set up a connection to AVS.
- Invoke the module's description function.
- Convey to AVS the module declarations for the module.
- Set up an instance of the module that can receive data from and send data to AVS.
- Invoke the module initialization function, if any.
- Return.

The main program can then interact with AVS at any time it wants. For example, the main program can behave like a subroutine module by looping indefinitely, taking the following steps on each iteration:

- Call the **AVScorout_exec** routine. This routine waits until the flow executive has stopped running and then returns.
- Call the **AVScorout_wait** routine. This routine waits until one of the module's inputs or parameters changes and then returns.
- Call the **AVScorout_input** routine. This routine obtains the module's inputs and parameters from AVS.
- Perform the module's computation.
- Call the **AVScorout_output** routine. This routine conveys the module's outputs to AVS.

More typically, a coroutine module performs a series of independent computations, sending output to AVS after each iteration. The main program can accomplish this by means of the loop described above, except that in order to compute continuously it must call the routine **AVScorout_mark_changed** before calling **AVScorout_wait**. This causes the **AVScorout_wait** routine to return immediately.

If a coroutine module computes continuously, it might provide a parameter to allow the user to stop the computation. The module can check this value of this parameter after the call to **AVScorout_input**. If the value indicates that the module should process continuously, it should call **AVScorout_mark_changed**. The next call to **AVScorout_wait** returns immediately, rather than waiting for the user to modify a parameter or for an upstream module to produce new data.

If the module has any input ports created with the flag: **REQUIRED**, the first call to **AVScorout_input** causes the module to wait for these ports to be connected and for data to be available.

You can use the routines **AVSinput_changed** and **AVSparameter_changed** with coroutine modules to indicate when an input or parameter has been modified. The return value is true when the input or parameter changes before the most recent call to **AVScorout_input**. In other words, you must call

AVScorout_input in order to update the knowledge of when inputs or parameters have changed.

A typical structure of a coroutine module is presented in the following "pseudo code":

```
Description Function:
   (describe inputs, outputs and parameters)

Main Routine

   Call AVScorout_init with "Description Function" name

   Loop for ever

      If we are running continuously
         Call AVScorout_mark_changed (mark module as changed)

      Call AVScorout_wait (wait for module to be changed)

      Call AVScorout_input (get input and parameter values)

      Optionally call AVSinput_changed or AVSparameter_changed
      to see what's new

      Compute your results

      Call AVScorout_output (send outputs to avs)

   Repeat loop
```

You can use coroutine modules to handle a variety of different synchronization problems such as the use of the X window events from within a module or synchronizing with other devices and polling the state of inputs and parameters. For more information on these types of synchronization and a more detailed description of how coroutine modules synchronize with the kernel and other modules, see the "Coroutine Synchronization" section in Chapter 4.

Handling Errors in Modules

AVS provides a mechanism for module computation routines and coroutine main programs to report errors. The **AVSmessage** routine causes AVS to present the user with a message from a module computation routine, along with information about the module and function sending the message. If the sender indicates that the message represents a warning or error, AVS stops executing and presents the message in a dialog box, along with a set of choices. The user must acknowledge the message by selecting one of the choices before AVS can continue. The icon for the module that sends the message is highlighted in yellow in the Network Editor. The **AVSmessage** routine also records the message in a log file for later review.

AVS treats error reports differently depending on their severity. The severity that the module declares determines how AVS presents the message to the user and whether or not the user must acknowledge the message before AVS can continue. If the message appears in a dialog box, the border of the dialog box is color coded to indicate the severity. Following are the possible levels of severity:

AVS_Information
The message does not indicate an error. The message is written to *stderr*, and AVS continues executing. No choices are presented to the user.

AVS_Debug
The message does not indicate an error; it conveys information during module testing. The message is written to *stderr*, and AVS continues executing. No choices are presented to the user.

AVS_Warning
The message indicates a problem that is not fatal to module execution. The message and choices are presented in a dialog box with a yellow border. The user must make a choice before AVS can continue.

AVS_Error
The message indicates a serious problem that may cause the module to produce erroneous results but is not permanently fatal to module execution. The message and choices are presented in a dialog box with a red border. The user must make a choice before AVS can continue.

AVS_Fatal
The message indicates a problem that is permanently fatal to module execution. The message and choices are presented in a dialog box with a black border. The user must make a choice before AVS can continue. The module is marked as dead, and the module icon in the Network Editor workspace turns black. The flow executive no longer executes the module.

Whenever a subroutine module computation function encounters an error that produces erroneous output, the computation function should return a value of 0. A coroutine module should not call **AVScorout_output** if such an error occurs because the flow executive does not execute downstream modules that depend on output from the module that encounters the error.

If a module encounters an error likely to be permanently fatal, such as a failure to allocate memory, it usually should not terminate its process by calling *exit*(2). Instead, it should call **AVSmessage** with a severity of **AVS_Fatal**. A subroutine computation function should then return a value of 0. A coroutine module should call **AVScorout_wait** and should not call **AVScorout_input** or **AVScorout_output** again.

If a module exits or dies unexpectedly and AVS tries to communicate with that module, AVS automatically generates a fatal error message. The user is

usually given an option to restart the module using the original parameters or the default parameters specified by the module description function.

AVS provides simple interfaces to **AVSmessage** for reporting errors of a given severity. These routines are called **AVSinformation**, **AVSdebug**, **AVSwarning**, **AVSerror**, and **AVSfatal**.

Selective Computation

When a module has more than one input port or parameter, it is possible that when the module computation function executes some ports or parameters have not changed since the previous execution of the computation function. By determining what has and what has not changed, the computation function may be able to avoid some computation on ports or parameters that have not changed.

AVS provides two routines, **AVSinput_changed** and **AVSparameter_changed**, to determine whether a given input port or parameter has changed since the previous invocation of the computation function. These routines return 1 if the input or parameter has changed and 0 if it has not. For a coroutine module, these routines determine whether the input or parameter has changed since the previous call to **AVScorout_input**.

When a module has more than one output port, it is possible that after the module computation function executes, some ports have not changed since the previous execution of the computation function. By default AVS assumes that all output ports have changed after each invocation of a module computation function. This can cause AVS to invoke downstream modules whose input depends on the output of the current module, even if some output ports have not changed.

AVS provides a routine, **AVSmark_output_unchanged**, to declare that a given output port has not changed since the previous invocation of the computation function. For a coroutine module, this routine declares that the output port has not changed since the previous call to **AVScorout_output**.

Building and Linking Modules

Each AVS module is a program that resides in a single executable file. The programmer can write the source code in either C or FORTRAN. The routines that the programmer provides depend on the source language and whether the module is a subroutine or a coroutine. For more information on subroutines and coroutines, see the "Subroutines and Coroutines" section in this chapter.

Writing Subroutines

A basic subroutine module as written by a programmer consists of a description function and a computation function, with optional initialization and destruction functions. The programmer does **not** supply a main program; instead, the AVS library supplies the main program for a module's executable file.

An executable file may contain more than one module, including description and computation functions for each module, but it has only one main program. In addition to the description and computation functions, the programmer supplies a function called **AVSinit_modules** to invoke the description functions for all modules in the file.

A FORTRAN file which contains only one module does not have to have a separate **AVSINIT_MODULES** function; instead, its description function can itself be named **AVSINIT_MODULES**.

If an executable has more than one module in it, by default, AVS creates a separate process for each instance of a module. See Chapter 4 for information on how to share a single process from multiple subroutine modules.

Writing Coroutines

A basic coroutine module as written by a programmer consists of a main program and a description function, with an optional initialization function. Each executable file can contain only one module. The description function can have any name.

Include Files

AVS supplies a number of include files for both C and FORTRAN programs. Some include files are needed for all modules, while others are needed only if the module is using data of a particular type. The routine descriptions in Appendix A lists any additional include files required by the particular AVS routine.

The AVS include files are located in the directory */usr/avs/include*. The file, */usr/include/avs*, is a link to this directory, so that both C and FORTRAN programs can refer to an include file using the following syntax:

```
#include <avs/filename>
```

For greater portability, FORTRAN modules should use the INCLUDE statement (no #) that requires an absolute pathname, such as:

```
INCLUDE '/usr/avs/include/avs.inc'
```

The example Makefile shows how to create a link to */usr/avs/include* called *avs* allowing the include statement to reference 'avs/avs.inc'.

C Language Include Files

All C language modules should include at least one header file:

avs.h
Data constants and primitive data types needed by all AVS modules.

The following files are needed when a module uses data of specific types:

avs_pixdata.h
Definitions for pixel maps.

colormap.h
Definitions for colormaps.

field.h
Definitions for fields.

geom.h
Definitions for geometries.

ucd_defs.h
Definitions for Unstructured Cell Data (UCD).

chemistry/CHEM.h*
Multiple definition files for Molecule Data Type (MDT).

udata.h
Definitions for user-specified data types used for upstream data flow.

Modules which call math functions such as *sqrt* or *sin* should not use the standard C header file */usr/include/math.h*. The file */usr/include/avs/avs_math.h* should be used instead, as it contains optimizations and declarations appropriate for the local hardware. See the following *avs_math.h* Include File section.

FORTRAN Include Files

All FORTRAN modules should include at least the following header file:

avs.inc
Data constants and data types needed by all AVS modules.

FORTRAN modules that use geometries should also include the following file:

geom.inc
Definitions for geometries.

The file *avs.inc* should be included after each subroutine declaration for subroutines that utilize AVS library calls. The file *geom.inc* should be included in each subroutine that utilizes Geometry Library calls. Refer to programs in */usr/avs/examples* and */usr/avs/filter* for examples of proper usage.

avs_math.h Include File

C programs using math functions, such as *sqrt*, normally include *math.h* which contains definitions resembling

```
extern double sqrt();
```

Without this, the compiler assumes *sqrt* returns *int*.

Some systems have special include files that allow the compiler to generate faster (perhaps inline) code. For each platform, it is desirable to use the best version of a math function that is available. In order to make code portable to any system supporting AVS, it is desirable to use a single include file with a name common across all systems.

The include file *avs_math.h* provides a way to write portable code that optimizes for each local platform on which it is compiled and linked. *avs_math.h* is therefore useful in computation intensive situations where it is desirable to use math routines optimized for the particular machine that will execute the module. *avs_math.h* contains system dependencies so that you should not have to modify source code as modules migrate to different systems. It should be used instead of the standard header file *math.h*.

Compiling and Linking Modules

AVS supplies four basic module libraries in the directory */usr/avs/lib*. Each module must be linked with one of these libraries. The library to use depends on the source language and whether the module is a subroutine or a coroutine:

Table 3-1 Archive Libraries for Modules

Module Type	Source Language	Library
Subroutine	C	*libflow_c.a*
Subroutine	FORTRAN	*libflow_f.a*
Coroutine	C	*libsim_c.a*
Coroutine	FORTRAN	*libsim_f.a*

A module might need to be linked with other libraries, depending on what data types it uses and what operations it performs. For example, a module that uses geometries needs the *geom* library, which requires linking with a number of library files. For details see the "Geometry Library" appendix.

An example ordering of libraries for a module that utilizes AVS and geom library calls might be the following:

-lflow_c -lgeom -lutil

Different platforms may require additional libraries, in a specific order.

Compilers may also differ from platform to platform. In general, you can use *cc* for C modules, and *f77* for FORTRAN modules. However, check the release notes that accompany your platform for specific compiler information.

Note: To avoid confusion about compilers, compiler options, libraries, and library linking order, you should use the example Makefile in /usr/avs/examples as a template for creating your own makefiles to compile and link modules.

Each of the template compile/link commands in the example Makefile contains a series of macro symbols that expand out to include the correct compiler options and libraries (in the correct order) for your platform. In the C case, the symbol FLOWLIBS expands out to a symbol BASELIBS, which includes the base AVS libraries necessary for a compile; plus a symbol LASTLIBS. LASTLIBS contains platform-specific libraries to link with. LASTLIBS itself is defined explicitly in the file */usr/avs/include/Makeinclude*, which */usr/avs/examples/Makefile* includes as its first step. The Makeinclude file's purpose is to define platform-specific options. The compiler flags are defined by the macro symbol CFLAGS, which expands to ACFLAGS and AVS_INC. ACFLAGS is defined in */usr/avs/include/Makeinclude*; AVS_INC is defined within the Makefile. The FORTRAN examples show the use of similar definitions for FORTRAN such as AFFLAGS and LASTFLIBS.

Using the example Makefile ensures that your modules will compile and link correctly on your platform. Pick a template compile/link command that corresponds to your module's type (subroutine or coroutine) and source language (C or FORTRAN). If you need to modify or supplement the compiler options or libraries, be sure to study the expansion of the macros and insert your modifications in the correct place without leaving any of the necessary base or platform-specific options or libraries out.

Converting an Existing Application to a Module

You can convert many existing simulations, batch data converters, and other scientific applications to AVS modules with little difficulty. Often such applications are most easily converted to coroutine modules. Following are some of the essential steps in the conversion process:

1. Determine what data the application needs to obtain from AVS as inputs or parameters and what data it needs to send to AVS as outputs.
2. Choose the AVS data type that is most appropriate for each input, output, and parameter.
3. Write a description function to declare the module and its inputs, outputs, and parameters.

4. In the application's main program, insert a call to **AVScorout_init** and calls to other AVS coroutine functions like **AVScorout_input**, **AVScorout_output**, and **AVScorout_wait** as appropriate.
5. Convert the program's data structures to the corresponding AVS data types for inputs, outputs, and parameters. **AVSbuild_field** is particularly useful in converting arrays to fields.
6. Ensure that the program allocates and frees memory for AVS outputs where necessary. The **AVSinitialize_output** and **AVSautofree_output** routines make this task easier.
7. Use **AVSmessage** or its variants to handle errors in the program.
8. Ensure that the program uses appropriate AVS include files. C language programs should include <*avs/avs.h*> and any files needed for particular data types. FORTRAN programs should include <*avs/avs.inc*>.
9. Compile and link the program with the AVS coroutine module archive library that is appropriate for the program's source language.

Converting an existing application to a subroutine module is similar, with these differences:

- Convert the application's main program to a computation function. A subroutine module does not supply its own main program.
- Ensure that the computation function returns 1 if successful and 0 if unsuccessful.
- Do not insert calls to AVS coroutine functions. Instead, ensure that the arguments to the computation function are the module inputs, outputs, and parameters.
- For a C language subroutine module, supply an **AVSinit_modules** routine. For a FORTRAN subroutine module, name the description function **AVSINIT_MODULES**.
- Compile and link the module with the AVS subroutine module archive library that is appropriate for the module's source language. AVS has different archive libraries for subroutine and coroutine modules.

Debugging Modules

AVS provides a facility for debugging a module during the execution of an AVS network. The file */usr/bin/avs_dbx* is a shell script that arranges for a module to run under the native debugger. On many UNIX systems, the native debugger is *dbx*. You can also specify an alternate debugger using the **-debug** option to **avs_dbx**.

Syntax of avs_dbx

The syntax of **avs_dbx** is as follows:

avs_dbx [**-id**] [**-debug** *com*] [**-mod** *mod_name*] [*debug_opts*] *file*

The *file* argument is the name of the executable file that contains the module. You can specify the following options:

-id

If you specify this option, the module runs under the debugger during an invocation of the module for *identification* (e.g. when identified by the **Read Module** command) as well as during an invocation of the module for *instantiation* (e.g. when moving the module from the module palette into the workspace). This option is useful if the module does not appear on the module palette when you use the **Read Module** Network Editor command (thereby implying the module's description function is not working properly).

When you invoke the **Read Module** command, AVS calls the module description function, which conveys the module declarations to AVS. The module's process then exits. When you specify the **-id** option, the execution of the module's description function caused by the invocation of the **Read Module** command runs under the debugger; by default it does not run under the debugger.

Note that when you create an instance of the module by moving the module icon from the module palette to the workspace, AVS invokes the module again, and this invocation is run under the debugger whether or not the **-id** option is present. During this invocation, AVS calls the description function again. The description function then runs under the debugger even if the **-id** option is not present.

-mod *mod_name*

Some executable files may contain more than one module. If you specify the **-mod** option, only the module named *mod_name* is run under the debugger. By default all modules in the file are run under the debugger.

If the **-id** option is also present, the description functions for all modules in the executable file are run under the debugger when the executable is invoked for identification. The description functions for all modules in the executable file are always run under the debugger when the module is invoked for instantiation.

debug_opts

These options are passed to the debugger. For more information, see your debugger's manual page.

-debug *debug_com*

This causes the **avs_dbx** command to run a debugger other than your native system debugger.

Using avs_dbx

The following describes how to use **avs_dbx** to run your debugger:

1. Compile the module using the **-g** option to *cc*(1) or *f77*(1). This option instructs the compiler to generate information needed by the debugger.

2. In a separate **xterm** window, issue the **avs_dbx** command. The **avs_dbx** command invokes the debugger in this **xterm** window and passes any options specified in the *debug_opts* argument to your debugger. When the debugger starts, you can set breakpoints and issue other debugger commands. Do not run your program yet.
3. Identify the module to AVS. In the Network Editor, you identify the module by invoking the **Read Module** command. This installs the module icon in the module palette.
4. Create an instance of the module. In the Network Editor, move the module icon from the module palette to the workspace.
5. In the **xterm** window, AVS prints the message:

   file `instance waiting, fire when ready...`

   Instruct your debugger to run the executable file that contains the module. The command for the *dbx* and *dbg* debuggers is **run**; consult your debugger manual if you are using another debugger.
6. In the Network Editor, you can now make connections to other modules and you can adjust parameters by manipulating widgets for the module. When the flow executive causes the module computation function to run, it runs under the debugger. All interaction with debugger takes place in the **xterm** window.

The following are some notes on using **avs_dbx**:

- You can run more than one module under the debugger by invoking the debugger in multiple **xterm** windows. However, if you want to run a module under the debugger, you cannot make more than one instance of the same module without copying the executable and using the **Read Module** command after starting the first instance. Note, however, that this creates two executable with the same name, the first of which is ignored if they are in the same module library.
- To run a module under the debugger, you must invoke the debugger before you make the instance of the module, (e.g. before moving the module icon from the Network Editor module palette to the workspace). You can use the **Read Module** Network Editor command to identify the module to AVS before invoking **avs_dbx**. However, in this case the **-id** option has no effect.
- Do not run your program under the debugger until after AVS has printed its "fire when ready" message. This message appears after you make an instance of the module.
- After you have made an instance of a module that is running under the debugger, you cannot manipulate any widgets for that module or make any Network Editor connections to or from that module until after you have run your program under the debugger and it has successfully completed the description function.
- Avoid commands that might disrupt the synchronization of the module's execution with AVS execution. For example, do not rerun your module unless you have destroyed its current instance and created a new one.

- If you recompile and relink a module after it has been identified to AVS, you do not have to re-execute the **Read Module** command. However, you should destroy all previous instances of the module before you make any instances of the recompiled module. If you want to run the module under the debugger, you must reinvoke **avs_dbx** before making an instance of the recompiled module.
- The **avs_dbx** command renames your executable by appending ".real" to its name while it is running. This can present a problem if the original file name is already the maximum allowed length. When you quit out of **avs_dbx**, it will restore your executable unless the executable has been modified since running the debugger. In either case, it will inform you of whether or not the executable was restored.
- You can use the **avs_dbx** command with both subroutine and coroutine modules.

Module Examples

Appendix C contains example source code for several AVS modules. Source code for these and other examples is also available in the directory */usr/avs/examples*.

CHAPTER 4

ADVANCED TOPICS

Introduction

This chapter discusses some of the more advanced topics in AVS module writing. These topics cover the general areas of memory allocation debugging, module process groups, coroutine synchronization, using upstream data, automatic connection to ports, and running multiple modules in a single UNIX process.

Memory Allocation Debugging

Dynamic memory allocation is always a problem area; there are many ways to make mistakes:

- Write beyond end of memory allocated.
- Use memory after it has been freed.
- Assume that *malloc* clears memory.
- Not checking for NULL returned from *malloc*.
- Specify single argument to *calloc*.
- Failure to free memory ("a memory leak").

Tracking these problems down can be a real challenge.

The task can be easier by having a layer of functions between the application and the system memory allocation routines. For example, instead of calling *malloc* directly, a new routine such as *MEMmalloc* can be used. This can do some error checking, gather statistics, and produce output as a debugging aid. The AVS library memory allocation calls such as **AVSdata_alloc** and **AVSauto_free** are using these functions.

Rather than changing all occurrences of memory allocation function calls, macros have been defined to replace the UNIX standard function calls. These are defined in the header file */usr/avs/include/mem_defs.h*. You do not need to reference this explicitly; it is included by the major AVS include files.

Part of the file resembles:

```
#ifdef MEM_DEFS_ENABLE
#ifndef _MEM_DEFS_defined
#define _MEM_DEFS_defined 1

extern char *MEMmalloc();
extern void MEMfree();
extern char *MEMrealloc();
extern char *MEMcalloc();
extern char *MEMstrdup();

#define malloc(size) MEMmalloc((size),__FILE__,__LINE__)
#define free(ptr) { MEMfree((ptr),__FILE__,__LINE__); (ptr) = NULL; }
#define realloc(ptr,size) MEMrealloc((ptr),(size),__FILE__,__LINE__)
#define calloc(nelem,elsize) MEMcalloc((nelem),(elsize),__FILE__,__LINE__)
#define strdup(ptr) MEMstrdup((ptr),__FILE__,__LINE__)
#endif
#endif
```

To use this facility, you must define the **MEM_DEFS_ENABLE** preprocessor symbol. This can be done by placing the following line of code at the top of your module's source file.

```
#define MEM_DEFS_ENABLE 1
```

Another approach for defining **MEM_DEFS_ENABLE** is to add *-DMEM_DEFS_ENABLE* to your list of compile time options (e.g., cc -I. -DMEM_DEFS_ENABLE foo.c).

When using the memory allocation debugging facility, there are some things to be cautious about:

- Casting the argument of *free* will result in a compiler error, (e.g., "free((char*)ptr);").
- Explicitly declaring *malloc* will result in a compiler error, (e.g., "extern char *malloc();").
- If you include *string.h*, do it before any of the AVS include files.
- Every source file using *malloc*, *calloc*, etc., should include one of the modified include files such as *avs.h*, *port.h*, *field.h*, or *geom.h* in order to include these macro definitions.

Run Time Environment Variables

To activate the memory allocation debugging routines, the following environment variables need to be defined.

AVS_MEM_CHECK = 1

When a block of memory is allocated, it will be filled with some non-zero value. This is to help catch cases that "just happened" to work most of the time on most machines because the memory usually contained zeros.

It will overwrite memory when it is freed to disappoint any routine trying to access it later. If **AVS_MEM_HISTORY** is not specified, only one byte is clobbered because the size of the block is not known.

This will also print the total number of allocates and frees just before exiting.

AVS_MEM_HISTORY = 1

This option keeps track of all allocations.

When a block of memory is freed, a search is made through the list of memory blocks that have been allocated. If the address is not found, an error message is printed.

When memory is allocated, a little extra memory is also allocated and filled with a particular pattern. When the memory is freed, the option looks for this pattern. If it is not there, it means that some routine wrote beyond the end of the block (e.g., allocated 1000 bytes and stored 1001 bytes there.)

AVS_MEM_VERBOSE = 1, 2, or 3

This option prints a message for every allocate or free as follows:

1 means within a module compute function only.
2 means outside of module compute functions only.
3 means everywhere.

Each message begins with:

- The executable filename.
- Module name if compute function is active.
- Source filename and line number.

Each allocation is assigned a sequence number to make matching the allocates and frees easier.

If **AVS_MEM_HISTORY** is active, the *free* will also print out information about the corresponding allocate.

The AVS libraries, modules and kernel are all built with **MEM_DEFS_ENABLE** defined. Thus, when you define these environment variables, you will see messages from AVS's own memory allocation/deallocation activities, as well as those of your own modules.

Below is an example of the use of these debugging facilities. First, we see some sample module source.

```
char *p, *q, *r;

p = malloc (8);
q = malloc (100);
r = malloc (99);          /* Not freed. */
strcpy (p, "too long");
free (q);               /* OK. Note that q is clobbered. */
free (p);               /* Error - wrote beyond end. */
free (q);               /* Error - was already freed. */
```

With the following environment variables defined,

```
setenv AVS_MEM_VERBOSE 1
setenv AVS_MEM_HISTORY 1
setenv AVS_MEN_CHECK 1
```

We get the resulting debugging output.

```
thing2(thing2.user.4) thing2.c:60: allocate 8 bytes, #26 result 0x1002ce24
thing2(thing2.user.4) thing2.c:61: allocate 100 bytes, #27 result 0x1002ce34
thing2(thing2.user.4) thing2.c:62: allocate 99 bytes, #28 result 0x1002ced0
thing2(thing2.user.4) thing2.c:64: free 0x1002ce34, 100 bytes #27 allocated at thing2.c:61
thing2(thing2.user.4) thing2.c:65: free 0x1002ce24, 8 bytes #26 allocated at thing2.c:60
thing2(thing2.user.4) thing2.c:65: free ERROR - Wrote beyond end of block (8 bytes #26 ...
thing2(thing2.user.4) thing2.c:66: free 0x55555555 - ERROR no record of corresponding malloc.
```

The module would die trying to free 0x55555555. With the final free removed, debug output concludes when the module is hammered.

```
thing2: Total of 29 allocates and 6 frees.
thing2: Memory allocated and not freed: current seq: 29

     :
     :

seq:   28; size:  99; ptr:1002ced0; line:   62; file:thing2.c

     :
     :
```

Coroutine Synchronization

Coroutine modules have the ability to execute asynchronously from the AVS flow executive. This means that at any time, coroutine modules can call **AVScorout_input** to acquire their input values and **AVScorout_output** to send output. This provides coroutine modules with more flexibility than subroutine modules. For example, they can manage their own input sources (such as X events or keyboard input), they can run in parallel with other AVS modules, and they can schedule execution themselves rather than waiting for user interaction.

Some example coroutine modules in the standard module set are the **animated integer**, **animated float**, and **particle advector** modules. These modules need be coroutine modules because, in certain states, they execute continuously. In contrast, subroutine modules can only execute in response to user input or input from upstream modules.

Coroutine modules can either execute continuously or can "wait" for upstream input or for the user to change a particular parameter.

In order to perform these tasks effectively, coroutine modules must be able to communicate with AVS to determine when they should run. For example, if a coroutine module is in a tight loop calling **AVScorout_output**, the AVS kernel

reads the data as fast as possible. If the coroutine module executes quickly enough (or the network takes long enough), some of the data produced by the coroutine may not processed by the network.

To avoid this problem, more synchronization with the flow executive is necessary. You can use the routine, **AVScorout_wait**, to facilitate flow executive scheduling of the module. This routine allows the flow executive to execute the module when it is the next "changed" module in the run queue. A "changed" module is one whose inputs or parameters have been modified or one that has been marked as changed via the **AVScorout_mark_changed** routine.

The flow executive executes the module when the following conditions are true:

- The flow executive is enabled.
- The module is the next "changed" module in the run queue.

The **AVScorout_mark_changed** routine is useful if you want to implement a continuously running module. When a module calls this routine, the flow executive marks the module as being in a "changed" state. AVS continues to consider this module as "changed" until the next call to **AVScorout_input** (or **AVScorout_output** if the module has neither inputs nor parameters). This causes the **AVScorout_wait** to return immediately rather than wait for input or parameter changes.

When using **AVScorout_mark_changed**, you should call **AVScorout_input** to determine when inputs and parameters change as **AVScorout_wait** is not responding to these events.

Another way to schedule the execution of a coroutine module with the flow executive is with the **AVScorout_exec** routine. Calling this routine causes module execution to wait until the flow executive stops running before returning. This is useful when you want to delay module execution until the network has completed its processing. See Appendix A for more information on these routines.

Coroutine Scheduling with X

The **AVScorout_wait** routine does not allow you to schedule module execution with X events. To do this, use the **AVScorout_X_wait** routine. When called, this routine waits for both input/parameter changes and X events/errors. **AVScorout_X_wait** also allows you to set a time interval that determine how long the routine waits before returning. The ability to set a "timeout" interval is useful when implementing features like "double-click" mouse action, for example. Setting the timeout interval to 0 makes this routine useful for polling the X server and to determine the status of module input/parameter changes. Remember to include <*sys/time.h*> when using this routine. See Appendix A for more information.

Coroutine Scheduling with Other Devices

AVS provides a more general mechanism by which a coroutine module can schedule with other file descriptor type devices. You can use the **AVScorout_event_wait** routine to wait for data from one of the specified file descriptor devices or to determine if module inputs or parameters have changed. If no descriptors are of interest, you can still use this routine to wait for input or parameter changes within the confines of the specified time interval by specifying the descriptor arguments as zero pointers.

On most systems, this routine uses the *select* system call. It allows the module to wait for all of the same events that *select* waits for and returns the same values that *select* returns. See the *select* man page for a complete description of the functionality, including error conditions, etc. The only difference between this routine and the *select* routine is that it takes an additional parameter that specifies the coroutine events to wait for. Currently only one coroutine event is supported: **COROUT_WAIT**. Remember to include <*sys/time.h*> when using this routine. See Appendix A for more information.

Synchronous Execution

By default, coroutine modules run in parallel with other modules. This means that AVS does not wait for coroutine modules to "finish" before starting other modules.

When running modules in parallel (i.e. asynchronously), the flow executive does not wait before scheduling any other modules that are ready to run. If there are no other modules that are ready to execute, this behavior does not cause problems. However, if there are other modules waiting to execute, problems may ensue because the two processes may share data. If this is the case, modules may executed twice or in an unpredictable way.

To prevent this, you can run your module synchronously with the flow executive. When a coroutine module runs synchronously, AVS assumes that as long as it is not waiting in **AVScorout_wait**, **AVScorout_event_wait** or **AVScorout_X_wait**, then it is running. Therefore, when the coroutine module is executing, no other modules are allowed to run.

By default, coroutine modules are run "asynchronous". To make your coroutine run synchronously, use the **AVScorout_set_sync** routine:

```
AVScorout_set_sync(value)
int value;
```

where value is "0" or "1". A value of 1 makes the coroutine run synchronously, a value of 0 makes it asynchronously. The coroutine can toggle its synchronous state during execution. The affects take place during the next call to **AVScorout_wait**.

Upstream Data

This section discusses the use of upstream data. The term upstream data refers to the process of sending information from one module to another module that precedes it in the AVS network. AVS uses the upstream data mechanism to communicate information about direct user manipulation of geometry (such as rotating an object with the mouse) to modules in the network that need this information to recalculate their contribution to the displayed image.

While AVS defines special data types to facilitate the use of upstream data, module developers can define their own data types for this purpose (see the "User-Defined Data" section of this chapter for information on defining your own data types). The following sections describe the upstream data facilities that already exist in AVS, as well as how to build these capabilities into modules you are developing.

Overview of Upstream Data Feedback Mechanism

In certain situations, modules need to receive information about events that occur after their own execution has completed. Examples of this feedback are the following:

- The arbitrary slice module that produces a geometry object and needs to get information when the user transforms the slice plane with the Geometry Viewer so that it can regenerate the slice at the new location.
- A module defining a molecule that needs information about which bond the user has selected so the bond can be highlighted in the image.
- The **probe** module that has a mode where each time the user "picks" an object, it snaps the probe to a vertex on the object selected.

The module developer implements this feedback through the data flow mechanism. A downstream module must have an additional output port from which to send the data and the upstream module must have an additional input port on which to receive the data. In addition, both of theses input and output ports must be defined as using the same data type.

The following is an example of using upstream data:

1. The upstream module **molecule** executes and outputs geometry data to the **geometry viewer** module.
2. The **geometry viewer** module executes, marking its upstream output port as unchanged. This system is now "idle".
3. The user selects a chemical bond. This causes the **geometry viewer** module to pass data on its upstream output port to the **molecule** module’s input port.
4. The **molecule** module executes in response to the upstream data change and highlights the selected bond. It outputs new geometry.

5. The **geometry viewer** module executes again, this time marking its upstream port as unchanged.

Note that you have created a loop in the network.

You must be careful when constructing loops. For example, if the downstream module outputs data on its upstream connection each time it executes and this always causes the upstream module to execute and output data to the downstream module, then you have constructed an infinite loop in the network.

AVS assumes all output data changes after each invocation of a module. In the example just discussed, the geometry module marks its upstream output port as unchanged (using the **AVSmark_output_unchanged** routine) when it executes in response to input from *upstream* in the network. However, when user interaction requires a change to the display (that is, changes occur *downstream* of the geometry module), the geometry module activates its upstream output port.

Implementing Upstream Data

This section describes how to use the two types of upstream data currently supported by AVS and how to implement your own type of upstream data in modules you develop.

In AVS, the only standard modules that can send upstream data are the **geometry viewer** module and the **display tracker** module. They support the following operations:

- Sending transformation information upstream (i.e. rotation, translation, and scaling information).
- Sending information on the selection of a particular object (that is, "picking" the object).

These two modules handle each of these cases using separate data types.

Transformation Information

The **geometry viewer** and **display tracker** modules can send upstream transformation data any time AVS transforms an object. The modules transmit the following data structure upstream:

```
typedef struct _upstream_transform {
   int flags;           /* Button state              */
   float msxform[4][4]; /* Modeling space transform */
   char object_name[256]; /* Current object          */
   int camera_index;    /* View transformation is in */
   int x, y;            /* Button x, y               */
   int width, height;   /* window width,height       */
} upstream_transform;
```

flags
The current button state that existed when the transformation occurred. In this case, the transformation is generated by the user rotating the object with the mouse. Possible values are: **BUTTON_DOWN**, **BUTTON_UP** or **BUTTON_MOVING.**

msxform
The object's current transformation matrix. This matrix transforms the vertices of the object from the modeling coordinate system (in which they are defined) to the coordinate system of the object's parent.

object_name
The name of the object whose transformation information is being passed upstream. Note that the object name may have a suffix appended to it (e.g. "arbitrary slice" may appear as "arbitrary slice.1").

camera_index
The view number of the window in which the transformation was generated.

x, y
The x, y position of the cursor in pixels (or -1 if the transformation was not generated by the mouse).

width, height
The width and height of the window in which the transformation was generated, if it was generated by the mouse.

Keep in mind that this data structure is transmitted from the downstream module to an upstream module. Once received, this data structure is available to the upstream module.

A geometry object can have two modes associated with it that determine how this mechanism operates: "notify" and "redirect". In both cases, AVS passes the same information upstream. The two modes differ with respect to how the geometry viewer treats the object.

In notify mode, the object is treated as any other object in the scene. When the transformation matrix is changed, the geometry for that object, and all child objects, are transformed and rendered. The downstream module then transmits the *upstream_transform* structure to the upstream module.

Notify mode is useful for cases where you want the upstream module to be informed of changes in the object's position/orientation but do not want it to regenerate the geometry (it is being regenerated anyway by the dowstream module). If the upstream module modifies the geometry, its output causes the downstream module to refresh the scene again.

AVS uses notify mode to implement the **probe** module. As the probe is transformed, the module obtains the data probe's transformation matrix. The module then has the opportunity to update the values displayed by the probe

according to the new position of the probe. Since the module uses notify mode, it does not have to update the position of the probe itself because this is handled by the geometry viewer.

In redirect mode, the transformation matrix is maintained like it normally is for the object, but this transformation matrix is not used to render the object or any children of the object. However, the downstream module still transmits *upstream_tramsform* to the upstream module (which may send new output to the downstream module, causing it to render the object).

Redirect mode is useful when you know that the upstream module needs to regenerate the geometry in order for the display to be correct. The output of the upstream module then causes the downstream module to execute. If the upstream module is going to regenerate the geometry, using redirect mode provides better performance because the scene is regenerated only once.

When a user defined upstream module receives upstream data from either the **geometry viewer** or the **display tracker** modules, the designer of the upstream module has the flexibility to use or ignore any of the data received. See the example, */usr/avs/example/pick_cube.c*, for sample code that uses upstream data.

The following is an example of how to use upstream data to send transformation information from the **geometry viewer** module to your upstream module:

- Add an input port to your module having the data type *struct upstream_transform*. This is a user-defined data type that is defined in *avs/udata.h*. See the "User-Defined Data" section for more details on user-defined data types.
- Your module should have a geom type output that is connected to the **geometry viewer** module.
- When constructing your edit list, you should use the routine, **GEOMedit_transform_mode**:

```
GEOMedit_transform_mode(edit_list,object_name,mode,flags)
GEOMedit_list  edit_list;
char *object_name;
char *mode;
int flags;
```

mode should have one of the following values:

- *notify* (to enable *notify* mode)
- *redirect* (to enable *redirect* mode)
- *normal* (to restore normal operation)
- *parent* (transform my parent object instead of me — not relevant to this section)

The *flags* argument should contain one or more of the following flags "OR"'d together: **BUTTON_DOWN**, **BUTTON_UP** and **BUTTON_MOVING**. The flags indicate for a given mouse button state when transforms should be sent

to the upstream module. For normal usage, you should specify **(BUTTON_-MOVING | BUTTON_UP)**. Since **BUTTON_DOWN** does not cause a change in the transformation matrix (but simply starts off a transform), specifying it causes a needless execution of the module.

The object name can be either the name of an object that the upstream module produced, the name of an object that another module produced, or one that was read in by the user directly from the geometry viewer. See Appendix G for more details on object names.

Setting the transform mode of an object to either *notify* or *redirect* causes the **geometry viewer** module and the **display tracker** module to output data from the "Transform Info" output port. If there is a connection from this port to the input port of type *struct upstream_transform* on an upstream module, that module executes when the object is changed. See the "Automatic Connection of Ports" section of this chapter to see how to instruct AVS to make this connection automatically. If you do not add automatic connection support in the module description function, you must make the connection by hand. See the example, */usr/avs/examples/pick_cube* for information on invisible ports.

Any subsequent call to GEOMedit_transform_mode for a particular object overrides the previous call. This means there can be only one requestor for the transformations of a particular object.

Selection Information

You can cause a module to executed by sending "selection information" to an input port any time a user picks an object. Selection information includes the name of the selected object, the nearest vertex selected and user-defined data associated with the nearest vertex, any user-defined data associated with the particular primitive selected (line, polygon, or sphere) and the coordinates of the 3D point that was selected.

The 3D point selected by AVS is the intersection between a ray projected from the pick point directly into the screen and the first encountered piece of geometry. The coordinates of the selected point is provided in several different coordinate systems, as well as the transformation matrices for the selected object and the view that contains the selected object.

AVS defines the following structure to contain this information:

```
typedef struct _upstream_geom {
   int flags;                  /* Button state, and selection mode */
   char current_obj[256];      /* Object whose selection mode set (coords
                                  are in the coordinate system of this object
                                 but vertices are defined for "picked obj" */
   float mscoord[3];           /* Modeling space coordinates  */
   float wscoord[3];           /* World space coordinates      */
   float sscoord[3];           /* Screen space coordinates     */
   float objxform[4][4];       /* Object's coordinate matrix */
   float worldxform[4][4];     /* From modelling space to world space */
   float viewxform[4][4];      /* From world space to screen space */
```

```
    int x, y;                   /* Button x, y              */
    int width, height;          /* window width,height      */
    int camera_index;           /* Index of the view selection was made */

    char picked_obj[256];       /* Name of object whose vertex was picked */
    float vertex[3];            /* nearest vertex to selection */
    int vdata;                  /* per-vertex data -- user specified */
    int odata;                  /* per-object (line,poly,sphere) data */
} upstream_geom;
```

flags
: One of **BUTTON_DOWN**, **BUTTON_UP** or **BUTTON_MOVING**.

current_obj
: The name of the object selected by the user.

mscoord
: The coordinates of the 3D selection point, in modeling coordinates (the coordinate system in which the object's vertices are defined).

wscoord
: The coordinates of the 3D selected point in world coordinates (lighting is performed in the world coordinate system).

sscoord
: The coordinates of the 3D selected point in screen coordinates (the screen coordinate system ranges from -1 to 1 with -1,-1,-1 being the lower left hand furthest corner of the window, 1,1,1 being the upper right hand closest corner).

objxform
: The current transformation matrix of the selected object.

worldxform
: The matrix that transforms the current object from modeling coordinates to world coordinates.

viewxform
: The matrix that transforms the current object from modeling coordinates to screen coordinates.

x, y
: The x and y coordinates, in pixels, of the selection point

width, height
: The width and height, in pixels, of the view in which the selection was made.

picked_obj
: The name of the object whose direct geometry was selected. This can either be *current_obj* or a descendant of *current_obj*.

vertex
The X,Y, and Z values of the selected vertex. This is a vertex contained in the geometry of *picked_obj*.

vdata
If *picked_obj* had any vertex data associated with the selected vertex, this 32 bit integer is stored in this member (Appendix G describes how to associate user-defined data with an object). If there is no vertex data, this member is -1.

odata
If *picked_obj* has any primitive or object data associated with the primitive that was selected, this member contains that data. Otherwise this field has the value -1.

Rules for Picking Objects

When the user selects (or picks) an object, the **geometry viewer** module receives a list of selected objects. The first item in the list is the object that contained the picked geometry. We'll call this the "picked object". The next item in the list is the parent of the "picked object", then the parent of that object, etc. The last item in the list is the "top" object.

Note that the "top" object is always picked regardless of where the user makes the pick (it is even put in the list when there is no "picked object").

There maybe multiple objects in the list that are requesting selection (that is, that need to be highlighted by the module owning the object). However, only a single object is reported as picked. The algorithm for choosing which object is picked is as follows:

1. If any object in the list is the current object, it has priority over all other objects and the selection information is sent to the module that requested a pick on this object.
2. If the current object is not in the list, the first object in the list that has been selected is picked and this object then becomes the current object.

Note that if you press the shift key when the you make a selection, the **geometry viewer** module changes only the current object and does not process any selections.

Picking the Top Level Object

If the user does not pick any geometry, the top level object becomes the current selection. If you pick the top level object, AVS returns valid data only in the following members of the *upstream_geom* data structure: current_obj, x, y, width, height, objxform, wsxform, viewxform, and picked_obj.

In this case, *picked_obj* is a zero-length string.

The following is an example of how to use upstream data to receive selection information at your upstream module.

- Add an input port to your module using the data type *struct upstream_-geom*. This is a "user-defined" data type that is defined in *avs/udata.h*. See the "User-Defined Data" section of this chapter for details on user-defined data types.
- Your module should have a geom type output that is connected to the **geometry viewer** module.
- When constructing your edit list, you should use the routine, **GEOMedit_selection_mode**:

```
GEOMedit_selection_mode(edit_list,object_name,mode,flags)
GEOMedit_list  edit_list;
char *object_name;
char *mode;
int flags;
```

mode should have one of the following values:

- *notify* (to enable *notify* mode)
- *normal* (to restore normal operation)
- *ignore* (to disable any picking of the object — not relevant to this section)

The *flags* argument should contain one or more of the following flags "OR"'d together: **BUTTON_DOWN**, **BUTTON_UP** and **BUTTON_MOVING**. The flags indicate for a given mouse button state when picks should be sent to your module. For example, if you specify only **BUTTON_DOWN**, you only get picks when the button is pressed. If you specify **BUTTON_DOWN** and **BUTTON_MOVING**, you get picks each time the button is pressed and subsequently each time that the cursor moves until the button is released.

The Geometry Viewer does not process **BUTTON_MOVING** and **BUTTON_RELEASE** selections if the object picked on the **BUTTON_DOWN** did not have any module requesting it.

The object name can be either the name of an object that you produced, the name of an object that another module produced, or one that was read in by the user directly from the geometry viewer. See Appendix G for more details on object names.

Setting the selection mode of an object to *notify* causes the **geometry viewer** module to output data from the "Geometric Info" output port. If there is a connection from this port to the input port of type *struct upstream_geom* on your module, it executes when the object is selected. See the "Automatic Connection of Ports" section of this chapter for information on making this connection automatically. If you do not add the support for your description function, you must make this connection by hand. See the example, */usr/avs/examples/pick_cube.c* for information on invisible ports.

User-Defined Upstream Data

While AVS supports only two modules capable of outputing upstream data, you can develop you own modules with this capability. You are not limited to passing information on geometric transformations and on picking. In fact, as long as you specify the input and output ports correctly and ensure that both upstream and downstream modules recognize the data structures that you are passing between them, you can build upstream data capabilities into any module you are designing.

You can define input and output ports to use any AVS data type (see Chapter 2), and you can also configure ports to use any data type that you can define using AVS' user-defined data capability. See the "User-Defined Data" section of this chapter for more information.

Once you've defined the desired data type, you can setup the port connections in the upstream and downstream modules description functions so that both ports accept your user-defined data type. If you want AVS to automatically connect the upstream ports when you make the module's downstream port connections, see the "Automatic Connection of Ports" section in this chapter.

As described earlier in the discussion on upstream data, you must be careful to avoid creating infinite loops in the AVS network when upstream data. See */usr/avs/examples/pick_cube.c* for an example of a module that use upstream data.

Automatic Connections of Ports

To simplify the network building process, AVS can hide certain types of connections from the user. This section describes a mechanism by which you can instruct the flow executive to automatically make an upstream connection when the user makes a downstream connection. Alternately, you can make ports optionally visible when the port is not required for module execution.

Port Classes

Data in a network can flow both downstream (e.g., **probe** outputs geometry to **geometry viewer**) and upstream (**geometry viewer** outputs pick information to **probe**). It is often the case that every upstream connection is associated with a particular downstream connection, usually, in order to feed back data about a user's action to the module that produced the particular object.

We associate a "class" attribute with both input and output ports. A class attribute is a character string name that contains two fields. The first (optional) field is a port name, the second (required) field is a port type specification. The port type specification is an arbitrary string that is meaningful to both the upstream and downstream modules

When the flow executive makes a connection between two modules, it looks for a match between the input ports of the upstream module and the output ports of the downstream module. If it finds a match, it makes this upstream connection.

A successful match occurs when the type of the input port class matches the type of the output port class. If a module has optionally specified a port name for the class, the match is made only if the port name specified is the name of the port being connected.

For example, the **probe** module defines a port class of type *upstream_transform* for its input port. The **geometry viewer** module defines a port class of type *upstream_transform* for its output port. When the **probe** module is connected to the **geometry viewer** module, the input port of the **probe** module is automatically connected to the output port of the **geometry viewer** module because the class matches. For simplicity, this particular case omits the optional *port name* field of the class attribute. This is almost always the correct thing to do.

Here is the code fragment that implements the appropriate part of the probe module's description function. Note that the data type of the input port is a user-defined data structure:

```
int port;
port = AVScreate_input_port("Transform Info",
                        "struct upstream_transform", OPTIONAL | INVISIBLE);
AVSset_input_class(port,"upstream_transform");
```

The purpose of the INVISIBLE flag is discussed in the next section.

Here is the code fragment that creates a compatible port for the output port on the **geometry viewer** module:

```
port =AVScreate_output_port("Transform Info",
                                "struct upstream_transform");
AVSset_output_flags(port,INVISIBLE);
AVSset_output_class(port,"upstream_transform");
```

Note that in the above example the correspondence between the class type ("upstream_transform") and the data type of the ports ("struct upstream_transform") is a convention, not a requirement.

In a more complicated and rarer example, we might have a module that has multiple outputs or a module that has multiple inputs. It might be the case that, for such a module, an automatic connection only makes sense when a particular input or output is connected. In this case, you can specify the optional port name in the port class to restrict the automatic connection mechanism to only take affect when that particular port is connected.

In this case, the output class contains two fields, the port name and the port type. The port name precedes the port type and the two are separated by a ":". Here is an example of such a case:

```
/* Description of our upstream module */
foo_desc()
{
   ...
   oport2 = AVScreate_output_port("First Output","integer");
   iport = AVScreate_input_port("First Input","boolean",OPTIONAL);
   AVSset_input_class(oport,"First Output:bizarretype");
   ....
}

/* Description of our downstream module */
bar_desc()
{
   ...
   iport1 = AVScreate_input_port("Input 1","integer",OPTIONAL);
   iport2 = AVScreate_input_port("Input 2","integer",OPTIONAL);
   oport = AVScreate_output_port("Output 1","boolean");
   AVSset_output_class(oport,"Input 1:bizarretype");
   ...
}
```

In the above example, if the module described by "bar_desc" is connected such that "Input 1" is connected to "First Output," an automatic connection is made from "Output 1" to "First Input". But, if a connection is made from "Input 2" to "First Output," there is no automatic connection made. This is because the class for the output port "Output 1" specifies that the port should only be connected if "Input 1" is the port that is connected.

Automatic connections are automatically disconnected when the connection that caused their creation is broken. There can only be a single class for a port. Automatic connections are not saved in a network but are recreated when the network is read in, if the classes defined in the modules haven't changed.

Port Visibility

You can make a module's input and output ports "invisible" so that colored boxes do not appear on the module icon. The default visibility of a port is assigned through the module description function by setting the port flag "INVISIBLE". For input ports this flag is specified with the routine **AVScreate_input_port** in the module description function. An example of this call is:

```
port = AVScreate_input_port("obscure port","integer",INVISIBLE | OPTIONAL);
```

Unlike **AVScreate_input_port**, the routine **AVScreate_output_port** does not have a "flags" field. To make an output port invisible, you must use the routine **AVSset_output_flags**:

```
int port;
port = AVScreate_output_port("obscure out port","integer");
AVSset_output_flags(port,INVISIBLE);
```

There are two situations in which you can effectively use the port visibility feature. The simplest is where your module contains an optional input or output port that is tangential to the module's execution. It may be the case that the input confuses the intended use of the module to naive users.

The second case is where you have two modules or a class of modules that are intended to be connected to each other. These modules may have a standard downstream connection and a standard upstream connection. Using the mechanism of port classes, you can arrange for an upstream connection to be made automatically when the downstream connection is made. This feature combined with the port visibility feature, allows upstream data to be hidden from the naive user and makes upstream networks much simpler to understand visually.

AVS saves the port visibility attribute when a network is written out and restores it when the network is read in.

User-Defined Data

AVS allows users to define their own data types and to use these data types for inter-module communication. In fact, the two standard AVS data types used for upstream data (*upstream_transform* and *upstream_geom*) are defined using this mechanism. This section is of interest to both users implementing their own data types and to those using the upstream data types.

Defining User-Defined Data

User-defined data is implemented as a "class" of data that can have an extensible number of subclasses. The class name for the user-defined data type is "struct".

You can define your own subclasses. For example, a complete user-defined description might be, "struct foo" where "foo" is the name of your subclass.

The user-defined data mechanism resembles the C *structure* definition, although Fortran users can also access these data types. AVS supports a subset of the mechanism for defining a **typedef** of a structure in C. For example, the declaration

```
typedef struct _foobar {
   int hop;
   int hog;
} foo;
```

defines a data type called "struct foo" that contains two integers, one named "hop", the other named "hog". Elements in this structure are restricted to int, char, float, double and arbitrary dimensional arrays of int, char, float, double. Elements CANNOT be pointers, unions, structures, enums, bitfields, etc.

AVS parses the header file containing these definitions with a parser that has limited understanding of valid C constructs. It is ignores all C pre-processor directives and comments. Therefore the following example is NOT VALID:

```
#define ARRAY 5

typedef struct _bar {
   int hop[ARRAY];
} bar;
```

An example of a valid declaration is:

```
/* These are foo flags -- we don't complain but
   don't look at them either */
#define FOO_FLAG 1
#define BAR_FLAG 2

typedef struct _dec1 {
   int flag;
   float matrix1[4][4], matrix2[4][4];
   char matrix_name[256];
} foobar;
```

AVS' user data definition capabilities are not designed to parse an arbitrary include file, but rather to provide the user with the capability to design a header file that can be parsed by AVS and also included in a C program.

Using a User-Defined Data Type On an Input Port

Inputing a user-defined data type is straightforward:

- For a C module, include the header file defining your data type
- Declare an input port of type "struct <classname>", where classname is the name of the typedef in your header file. For our example, it is the following:

```
port=AVScreate_input_port("my port name","struct foobar",<flags>);
```

- For a C module, the argument to your compute function is a pointer to a structure of the type you've specified. A declaration for the compute function defined in our example is the following:

```
#include "foo.h"

foo_compute(fooptr)
foobar *fooptr;
{
   if (fooptr->flag == ...)
}
```

- For a Fortran module, there are two ways to access the data depending on whether or not the SINGLE_ARG_DATA flag is set using **AVSset_module_flags**. By default, the arguments to your compute function are ex-

panded so that you have a separate parameter for each field in the structure of the data type. Our example has four arguments for its input port.

If you select SINGLE_ARG_DATA, then AVS passes a single integer value for each user-defined data input or output defined. This integer value is then passed to a number of accessor functions which copy data to or from the user-defined data structure into a local array or scalar variable. For example, you can use **AVSudata_get_int** to retrieve integer arrays or scalars from a user-defined data structure. The module must call **AVSload_user_data_types** before using the accessor functions so AVS has access to a description of the user-defined data structure. See Appendix A for descriptions of the accessor functions for user-defined data types. Also see the example, */usr/avs/example/user_data_f.f*.

Using a User-Defined Data Type On an Output Port

Outputing user-defined data requires slightly more effort. In the description function, you declare the output port in the same way that you declare the input port. In addition, you must specify the file name that contains the data types to load using the **AVSload_user_data_types** routine as follows:

```
AVSload_user_data_types(filename)
char *filename;
```

For example:

```
AVScreate_output_port("my out port name","struct foobar");
AVSload_user_data_types("/mydir/foo.h");
```

If you provide a relative pathname to the routine, AVSload_user_data_types, the file should be located relative to the directory */usr/avs/include*.

For a C module, you must declare the data type in your module as a pointer to a pointer to the structure you declared, and then use the routine, **AVSdata_alloc**, to allocate the data as follows:

```
foo_output(foopp)
foobar **foopp;
{
   if (*foopp == NULL) *foopp = AVSdata_alloc("struct
foobar",0);
   (*foopp)->flags = ...
}
```

AVS does not use the second argument to the **AVSdata_alloc** routine. To accommodate future enhancements to user-defined data, you should pass a 0 as the argument. In FORTRAN, the output user data field is an integer; call **AVSdata_alloc** to allocate the field as in the C example.

Image Picking Data Type

The user-defined data type mechanism has been used to implement an image-picking data structure. The **image viewer** module outputs this data structure through its leftmost, normally invisible, output port. Downstream modules can use this data structure to take action based upon where a user has clicked the left mouse button in the **image viewer** scene window. For example, a module could use the picking information to display the original field data values present at that location before it was converted to an ARGB image. At present, none of the supplied AVS modules make use of this data type.

The data structure is:

```
typedef struct _iv_pick {
  int  view_x, view_y;   /* picked point in viewspace (always valid)   */
  int  view_window_id;   /* id of iv window (always valid)             */
  char image_name[256];  /* name of picked image ("" if none picked)   */
  int  image_x, image_y; /* picked point in imagespace (-1,-1 if none) */
  int  image_window_id;  /* id of image window (-1 if none picked)     */
  char label_name[256];  /* name of picked label ("" if none picked)   */
                         /* label_name with no image indicates title   */
} iv_pick;
```

View space is the X, Y field of the entire image viewer scene window. (0, 0) is at the upper left corner. The *view_window_id* is the X Window System window ID of the scene window. The *image_name* is the name of the image displayed by the **image viewer**'s Current Image Browser. It is also the image name required by the CLI. *image_x* and *image_y* are the X, Y coordinates of the point picked within an individual image, with (0, 0) at the upper left corner. X and Y coordinates are preseved across image scaling operations. *image_window_id* is the X window id of the individual image. If the user happens to have picked an Image Viewer label, then *label_name* is its identifying string, in the format required by the CLI.

To use this data type, you must include the file */usr/include/avs/udata.h* as follows:

```
#include <avs/udata.h>
```

As image picking is implemented with user-defined data, see the previous "User-Defined Data" section for more information on its use as input and output from both C and FORTRAN.

Multiple Modules in a Single Process

In AVS, it is possible to run multiple AVS modules from the same UNIX process. This has several advantages:

- There are fewer UNIX processes running, which in turn, uses less of the system resources (sockets, process table slots, etc.)

- Module startup for the second and subsequent modules does not require the creation of a new process and is therefore faster.
- When two modules that are in the same process are connected, AVS can avoid some of the communication overhead of passing data between the two modules.
- It can reduce memory requirements.

There is, however, a disadvantage to running many modules in one process; one module could, conceivably, kill the process and thereby kill all the modules running in that process.

Restrictions

There are some restrictions on the conditions under which you can run multiple modules in the same process:

- The two modules must not interfere with each other's data allocation. For example, two modules in the same executable cannot use the same static memory locations to store read / write information. Here is an example of two modules that you cannot run in the same process:

```
int globalvar;

module1_compute(foo,output)
int foo, *output;
{
   globalvar++;
   *output = foo + globalvar;
}

module2_compute(bar,output)
int foo, *output;
{
  globalvar--;
  *output = bar - globalvar;
}
```

- Two instances of the same module cannot run in the same executable unless they do not rely on any read / write static data. Modules that do not use any read / write static data are usually called "re-entrant" modules. Modules that are not re-entrant cannot be executed in the same process.
- You cannot run coroutine modules in the same process with any other module.
- You cannot mark an input port with the flag, **MODIFY_IN** and run multiple modules in a single process. This flag is used in situations when you want the module to be able to modify the data on its input port. Since all of the modules in a single executable can share the same data, modules that rely on the **MODIFY_IN** flag are not suitable to run with other modules in a process.
- In general, modules that do not free allocated static data in their destruction function should not be run with other modules in a single process.

This is because, if this memory is not freed, it is not available to other modules in the process that are still active.

Implementing Multiple Modules Processes

When you compile multiple modules into a single executable, AVS starts a new process each time a module is executed. To run a module cooperatively with other modules in the same executable, you must set the **COOPERATIVE** module flag. You do this using the **AVSset_module_flags** routine in the description function of each module that you want to run cooperatively. Set the flag as follows:

```
AVSset_module_flags(COOPERATIVE);
```

In order to run a module cooperatively, the module must be able to run cooperatively with **ALL** the modules in the executable file.

If the module is "reentrant", (i.e. multiple instances of a particular module can be run from the same process), you must explicitly mark it as such by setting the **REENTRANT** module flag:

```
AVSset_module_flags(COOPERATIVE | REENTRANT);
```

When AVS is about to instance a module, either from reading in a network or as a user moves a module to the workspace, it searches the list of currently active modules for an existing process that matches all of the following conditions:

- The process is an executable that contains the same version of the module that AVS is going to execute.
- The module to instance and all active modules in the process are marked as **COOPERATIVE** modules.
- Either the module is marked as **REENTRANT** or there is not an existing instance of the particular module in the executable.
- No modules in the process are currently executing. If AVS determines there is a module executing, it starts another process for the module. This means that you cannot necessarily rely on a module being run in an existing process.
- AVS was not started with the **-separate** option.

If you change the module flags or any other option in the module description function, you must regenerate any module library that contains that module.

If a module dies, **AVSmessage** offers the user a choice to restart the module. If the user chooses to restart the module, all modules running in that process are restarted sequentially so that AVS can again run these modules from a single process.

You can also restart a module by pressing the module tools button while in the network editor. This brings up a choice box that contains a "restart modules" button.

Implementing Reentrant Modules

Modules that require the use of static data can use the **AVSstatic** feature of the module programmers interface. This is an external variable of type "char *" that AVS retains on a per-module basis. It is defined in the header file */usr/avs/include/flow.h*. AVS saves the values assigned to **AVSstatic** after executing the module and restores it before executing the next module. The value is also saved and restored for the initialize and destroy functions that your module might define.

You can use this variable to store a pointer to information that you want to keep available from one module invocation to another. **AVSstatic** is available only in C modules.

Modifying Modules that Share Processes

When a module is instanced, it is possible that it will attach to an existing process rather than starting a new process. It is also possible that the module's executable could have been modified since the previous process was started. You could, therefore, end up running a stale copy of the module.

There are two ways to avoid this:

- Each time you change the module, do a **Read Module** on the executable. This causes AVS to mark the current process as running a different version of the module. Subsequent attempts to instance a module in this executable starts a new process.
- Start AVS with the command line option: **-separate**. This causes AVS to run each module in a separate process regardless of how you set their module flags.

Linking Multiple Modules Together

When modules are linked into one executable, there should not be a separate **AVSinit_modules** function for each module. Instead the initialization of each module to be compiled into one executable is specified in a separate file. The names of the description functions for each module (e.g., your_mod1) to be linked into the single executable file are listed in this file. The following is an example of such a file.

```
#include <avs/avs.h>
#include <avs/flow.h>
```

```
/* module initialization */

extern int
  your_mod1(),
  your_mod2(),
  your_mod3();

/* Build Module List */

static int ((*mod_list[])()) = {
  your_mod1,
  your_mod2,
  your_mod3};

#define NMODS sizeof(mod_list) / sizeof(char*)

AVSinit_modules ()
 {
   AVSinit_from_module_list(mod_list, NMODS);
 }
```

Note that the **AVSinit_modules** function calls **AVSinit_from_module_list** instead of the **AVSmodule_from_desc** function. This and all the modules listed in this file are linked into one executable.

Since the **AVSinit_modules** function is now located in this file, it should not occur in the individual modules since this would cause multiple definitions of the same function.

If your modules are written in FORTRAN, you would use the same format as shown for C, except that:

- You do not need the "static int ((*mod_list[])())" block, or the #define that follows that counts the number of modules in the list.
- You would use a series of **AVSmodule_from_desc** calls, one for each module, in place of the **AVSinit_from_module_list** call.

It is also possible to link the standard AVS module set in with your modules as one binary so that they will execute in one process. The file *avs/examples/multi_hog.c* is an example of file that performs this task. Use *multi_hog.c* as a template, adding your own module function description names to the AVS modules initialized in this file.

The *avs/examples/Makefile* make file provides a useful template for creating the appropriate make file to compile your modules. The line for *multi_hog.c* further specifies the $(MODLIBS) option. At the top of the *Makefile*, this is expanded to include the set of six libraries that represent the supported AVS module set compiled in "library" form: *libmdata.a*, *libmfilt.a*, *libmmapp.a*, *libmrend.a*, *libmucd.a*, and *librf.a*. The MODLIBS symbol can be removed if you are just linking your own modules together.

Module developers may find it useful to use a preprocessor symbol that determines whether the module has its own **AVSinit_modules** function. This al-

lows you to develop and test the module as a separate executable. Later it can be linked with other modules without changing the source. Simply recompile without the symbol defined.

For example, the *your_isosurface* module might have its own AVSinit_modules function which is used only when the **SEPARATE_MODULES** flag is defined:

```
#ifdef SEPARATE_MODULES
/*****************************************************/
AVSinit_modules()
{
    int your_isosurface();

    AVSmodule_from_desc(your_isosurface);
}

/*****************************************************/
#endif
```

You can define this symbol on the command line (or in a make file) when the compilation command is given using the -D option.

In order for a module to be eligible to run in the same process as another module, the **COOPERATIVE** flag must be set. The **COOPERATIVE** flag is set within a module using the **AVSset_module_flags** routine. This routine is described in Appendix A of the *AVS Developer's Guide*.

All modules supplied with this product have the **COOPERATIVE** flag set, EXCEPT the following:

tube
wireframe
flip normal
offset
pixmap to image
image to pixmap
animate lines
extract vector
isosurface
particle advector
field legend
tracer
display tracker
write image
write volume
ucd legend
ucd advect

Coroutine modules cannot run cooperatively and consequently always run in their own process.

Module Groups

Modules have an attribute called the "module group" that can be used to control whether two modules are placed in the same process or not. This feature is useful for advanced network editing operations to:

- Increase the parallelism of a network by ensuring that parallel fragments of a network are not placed in the same process. Two modules that are in the same process cannot be run in parallel.
- Increase the efficiency of executing a particular network by ensuring that modules that are connected are placed in the same process so that they can share data if inter-process shared memory is not available for some reason.

The module group attribute is an ASCII string that the user can define for a particular module by typing into the Module Editor panel of a particular module. When this string is the same for two modules, they share the same module group and can be placed in the same process when the module is created.

The module group can only be used to inhibit two modules from being placed in the same process but by inhibiting a module from sharing another module's process, it opens up an opportunity to share the process with a third module.

Here is an example. We have three modules "module A", "module B" and "module C". All three modules are contained in the same executable, all are marked **REENTRANT** and **COOPERATIVE**. We want module A and module B to be run in parallel so they cannot be in the same process. But, we want module A and module C to be in the same process to reduce the startup time of the network. We can do this most concisely by defining a module group for module B when the module is still in the module Pallete. Now, when we instance module A and module C, they will be placed in the same process because neither module has a defined module group string. When we instance module B, however, its module group won't match the non-existant group of module A, and so it will start a new process. Module A and module B can now be run in parallel if the network is constructed in a way that allows parallel execution.

There are other situations where defining the module group may make sense. Some systems do not support shared memory communication of data between two processes. On these systems, when two modules are connected it is necessary for AVS to make an extra copy of the data set in order for the downstream module to have access to it—unless the two modules are in the same process. A module that is **COOPERATIVE** but not **REENTRANT** can only have one module instance in a particular process. We can therefore come up with a situation in which the system will make an arbitrary decision when grouping modules into processes. The user can use the module group to impose an order on this grouping.

The module group can be modified for either an inactive module in the Palette or for an active module that has been instanced in the Network Editor Workspace. The module group attribute is only referenced, though, when you are creating a new module (dragging it into the Workspace). Changing the module group of an active module does not rearrange modules in processes but will potentially affect how a new module is started.

The module group attribute is properly saved and retrieved when you save away a network.

Note: There is a CLI **net_group** command that supports a different, simpler function: the lassoing of modules to be moved as a group. The *net_group* CLI command is irrelevant to module grouping as just described. Module process grouping is controlled by a **-group** option to the **module** CLI command.

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 sub-

nets, 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 set-

ting) 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**. Hit-

ting 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 individ-

ual 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 re-

quests 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>} }
```

APPENDIX A

AVS LIBRARY ROUTINES

Introduction

The routines described in this section are designed for use from within AVS modules. These routines allow module developers to implement the following functions:

- Initializing and describing modules to AVS
- Parameter handling
- Accessing data
- Error handling
- Coroutine event handling

Include File

AVS supplies a number of header files that you should include in each module. All modules written in C should include */usr/avs/include/avs.h* and all modules written in FORTRAN should include */usr/avs/include/avs.inc*. Although the **#include** compiler directive is shown in most of the FORTRAN declarations, the **include** statement is preferred for portability reasons. In addition, other header files are required by some routines. These files are specified with the individual routine description.

Type Declarations

Many AVS routines can input and/or output different types of data, depending on, for example, the data type the port is designed to handle. In these situations, the routine's parameter declaration is specified as:

`<type>`

AVS provides a variety of data access routines that facilitate access to complex data types, such as AVS fields, AVS colormaps, and user-defined data. These routines are described in this appendix. Using the AVS user-defined data facility is described in Chapter 4.

Routine Summary

The following list of AVS routines is organized by functional category. Complete descriptions follow in the remainder of the chapter. Note that some routines are functions that return values. See the main routine descriptions.

Routines for Module Initialization

AVSinit_modules()
AVSinit_from_module_list(*AVSmodule_list, count***)**
AVSmodule_from_desc(*desc***)**

Routines for Module Description Functions

AVSadd_parameter(*name, type, init, minval, maxval***)**
AVSadd_float_parameter(*name, init, minval, maxval***)**
AVSadd_parameter_prop(*param_num, prop_name, prop_type,*
AVSautofree_output(*out_port***)**
AVSconnect_widget(*param_num, widget_type***)**
AVScreate_input_port(*name, type, flags***)**
AVScreate_output_port(*name, type***)**
AVSinitialize_output(*in_port, out_port***)**
AVSload_user_data_types(*filename***)**
AVSset_compute_proc(*comp_func***)**
AVSset_destroy_proc(*destroy_func***)**
AVSset_init_proc(*init_func***)**
AVSset_input_class(*port, class***)**
AVSset_module_flags (*flag***)**
AVSset_module_name(*name, type***)**
AVSset_output_class(*port, class***)**
AVSset_output_flags(*port, flags***)**
AVSset_parameter_class(*port, class***)**

Routines for Modifying and Interpreting Parameters

AVSchoice_number(*name, string***)**
AVSmodify_float_parameter(*name, flags, init, minval, maxval***)**
AVSmodify_parameter(*name, flags, init, minval, maxval***)**
AVSmodify_parameter_prop(*name, prop_name, prop_type, prop_value***)**
AVSparameter_visible(*name, stat***)**

Routines for Coroutine Modules

AVScorout_event_wait(*nfds,readfds,writefds,exceptfds,timeout,mask***)**
AVScorout_exec()
AVScorout_init(*argc, argv, desc***)**
AVScorout_input(*input1, input2, ..., param1, param2, ...***)**
AVScorout_mark_changed()
AVScorout_output(*output1, output2, ...***)**
AVScorout_set_sync(*value***)**
AVScorout_wait()
AVScorout_X_wait(*dpy, timeout, mask***)**

Status Monitoring Routines

AVSmodule_status(*comment, percent***)**

AVS Command Language Interpreter Routine

AVScommand(*destination, command_buffer, output_buffer, error_buffer***)**

Routines for Selective Computation

AVSinput_changed(*port_name, i***)**
AVSmark_output_unchanged(*port_name***)**
AVSparameter_changed(*param_name***)**

Routines for Creating Fields

AVSPORT_FIELD(*PORT_NAME***)**
AVSdata_alloc(*string, dims***)**
AVSdata_free(*type, data_ptr***)**
AVSfield_alloc(*template, dims***)**
AVSfield_copy_points(*field_in, field_out***)**
AVSfield_free(*field***)**
AVSfield_make_template(*field_in, template***)**
AVSbuild_field(*ndim, veclen, uniform, ncoord, type, dim1, dim2, ..., data, coords***)**
AVSbuild_2d_field(*data, dim1, dim2***)**
AVSbuild_3d_field(*data, dim1, dim2, dim3***)**

Field Accessor Routines

AVSFIELD_DATA_OFFSET(*FIELD, BASEVEC, OFFSET***)**
AVSfield_data_ptr(*field***)**
AVSfield_get_dimensions(*field, dimensions***)**
AVSfield_get_extent(*field, min_extent, max_extent***)**
AVSfield_get_int(*field, selector***)**
AVSfield_get_label(*field, number, label***)**
AVSfield_get_labels(*field, labels, delimiter***)**
AVSfield_get_minmax(*field, min, max***)**
AVSfield_get_unit(*field, number, unit***)**
AVSfield_get_units(*field, units, delimiter***)**
AVSfield_invalid_minmax(*field***)**
AVSFIELD_POINTS_OFFSET(*FIELD, BASEVEC, OFFSET***)**
AVSfield_points_ptr(*field***)**
AVSfield_reset_minmax(*field***)**
AVSfield_set_extent(*field, min_extent, max_extent***)**
AVSfield_set_int(*field, selector, value***)**
AVSfield_set_labels(*field, labels, delimiter***)**
AVSfield_set_minmax(*field, min, max***)**
AVSfield_set_units(*field, units, delimiter***)**
AVSfield_get_mesh_id(*field, mesh_id***)**
AVSfield_set_mesh_id(*field, mesh_id***)**
AVSget_unique_id()

Colormap Accessor Routines

AVScolormap_get(*cmap, max_size, size, lower, upper, hue, saturation, value, alpha***)**
AVScolormap_set(*cmap, size, lower, upper, hue, saturation, value, alpha***)**

User Data Accessor Routines

AVSudata_get_double(*ptr, name, value, value_elements***)**
AVSudata_get_int(*ptr, name, value, value_elements***)**
AVSudata_get_real(*ptr, name, value, value_elements***)**
AVSudata_get_string(*ptr, name, value, value_elements***)**
AVSudata_set_double(*ptr, name, value, value_elements***)**
AVSudata_set_int(*ptr, name, value, value_elements***)**
AVSudata_set_real(*ptr, name, value, value_elements***)**
AVSudata_set_string(*ptr, name, value, value_elements***)**

FORTRAN Array Accessor Routines

AVSPTR_ALLOC(*NAME, NELEM, ELSIZE, CLEAN, BASEVEC, ADDR, OFFSET***)**
AVSPTR_OFFSET(*NAME, ELSIZE, BASEVEC, ADDR, OFFSET***)**

FORTRAN Single Byte Accessor Routines

AVSLOAD_BYTE(*BASE, OFFSET***)**
AVSSTORE_BYTE(*BASE, OFFSET, VALUE***)**

Routines for Handling Errors

AVSdebug(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
AVSerror(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
AVSfatal(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
AVSinformation(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
AVSmessage(*version, severity, module, function_name, choices, message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
AVSMESSAGE_SUB(*ANSWER, VERSION, SEVERITY, MODULE, FUNCTION_NAME, CHOICES, MESSAGE***)**
AVSwarning(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**

Routines for Module Initialization

Modules can use routines in this section only in the module description function. See Chapter 3 for general information on module description functions.

AVSinit_modules

C:
AVSinit_modules()

FORTRAN:
AVSINIT_MODULES()

The AVS programmer defines this routine. AVS invokes this routine when it loads the modules defined in a file. Each executable file that defines subroutine modules should have one and only one definition for **AVSinit_modules**. Use **AVSinit_modules** as follows:

- Each source file can define more than one module. **AVSinit_modules** should contain one call to **AVSmodule_from_desc** to initialize each module defined in the file. Alternately, **AVSinit_modules** can call **AVSinit_from_module_list** to initialize a list of modules defined in the file.
- In FORTRAN, if the source file defines only one subroutine module, the module description function itself can simply be called **AVSINIT_MODULES**.

A file that defines a coroutine module should not have a definition for **AVSinit_modules**; a coroutine calls **AVScorout_init** from its main program instead.

AVSinit_from_module_list

C:
AVSinit_from_module_list(*AVSmodule_list, count***)**
int *(**AVSmodule_list)();*
int *count;*

AVSinit_from_module_list initializes a list of modules from their description functions. The *AVSmodule_list* argument is a list of pointers, one to each module description function defined in the file. The *count* argument is the number of pointers in the list.

Source files can define more than one module to be built into a single executable. The programmer-supplied routine **AVSinit_modules** can call **AVSinit_from_module_list** to initialize a list of modules defined in the file.

There is no FORTRAN equivalent for this routine. FORTRAN programmers must use the routine **AVSmodule_from_desc**.

AVSmodule_from_desc

C:
AVSmodule_from_desc(*desc***)**
int *(*desc)();*

FORTRAN:
AVSMODULE_FROM_DESC(*DESC* **)**
EXTERNAL *DESC*

AVSmodule_from_desc initializes a module from its description function. The *desc* argument is a pointer to the description function.

Source files can define more than one module to be built into a single executable. The programmer-supplied routine **AVSinit_modules** must contain one call to **AVSmodule_from_desc** to initialize each module defined in the file. Alternately, **AVSinit_modules** can call **AVSinit_from_module_list** to initialize a list of modules defined in the file.

In FORTRAN, if the source file defines only one subroutine module, the module description function itself can simply be called **AVSINIT_MODULES**.

Routines for Module Description Functions

AVSadd_parameter

C:
#include *<avs/avs.h>*
int AVSadd_parameter(*name, type, init, minval, maxval***)**
char **name, *type;*
<type> init, minval, maxval;

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSADD_PARAMETER(*NAME, TYPE, INIT,*
*MINVAL, MAXVAL***)**
CHARACTER*n *NAME, TYPE*
<type> INIT, MINVAL, MAXVAL

This routine declares a parameter for the module being defined in the current description function. Each parameter is usually connected to a widget in the module control panel to allow the user to modify the value of the parameter.

This routine returns an integer parameter identifier that other AVS routines, such as **AVSconnect_widget**, use as an argument.

The *name* argument is a string that appears as the name of the widget associated with the parameter.

The *init*, *minval*, and *maxval* arguments are cast as **int**s in C and **integer**s in FORTRAN, but their storage type actually depends on the parameter type. For any type of parameter, *init*, *minval*, and *maxval* all have the same storage type. For example, if the parameter is of type "string", all three values must be char* values (or CHARACTER*(*) in FORTRAN). Each value must fit into an integer-size memory slot or must be a pointer to a larger memory allocation. Values representing **float**s in C must be pointers to allocated memory. The routine **AVSadd_float_parameter** handles this allocation automatically. FORTRAN does not have this special requirement.

For many parameter types, *init* is the initial or default value of the parameter, and *minval* and *maxval* are the inclusive bounds for the acceptable range of values. When this range is specified, AVS ensures that values passed to the computation routine are inside this range.

The *type* argument is a string that represents the parameter type. The following table lists the possible values for *type*. For each type, it lists the C and FORTRAN data types for *init*, *minval*, and *maxval*. These are also the data types for parameters passed as arguments to module computation routines.

Table A-1 Parameter Types and C/FORTRAN Data Type Declarations

type **String**	**C Data Type**	**FORTRAN Data Type**
"integer"	**int**	**INTEGER**
"boolean"	**int**	**INTEGER**
"tristate"	**int**	**INTEGER**
"oneshot"	**int**	**INTEGER**
"real"	**float ***	**REAL**
"string"	**char ***	**CHARACTER*n**
"string_block"	**char ***	**CHARACTER*n**
"choice"	**char ***	**CHARACTER*n**
"colormap"	**AVScolormap ***	**INTEGER**
"field"	**AVSfield ***	**INTEGER**
"delta_matrix_4x4"	**AVSfield ***	**INTEGER**

AVS passes most parameters to the compute function as a single argument. However, AVS passes fields to FORTRAN computation functions as multiple arguments by default. It may be desirable to call **AVSset_module_flags** to instruct AVS to pass a single argument instead, which can then be used with language independent field access routines; see the "AVS Data Types" chapter. The "INTEGER" declarations listed above for colormaps and fields are for use when passing these data structures as a single argument.

Following are notes on some of the parameter data types:

integer

The *minval* argument is the minimum value; the *maxval* argument is the maximum value. To specify an unlimited range of possible values, set both *minval* and *maxval* to the constant **INT_UNBOUND**. Both *minval* and *maxval* must be either bounded or unbounded.

boolean
Possible values are 0 and 1. The *minval* and *maxval* arguments are ignored.

tristate
Possible values are 0, 1, and 2. The *minval* and *maxval* arguments are ignored.

oneshot
This is a command-style signal counter. The current value is incremented by 1 each time the value is set, often by means of a mouse click on a widget. This allows the module to determine how many times the user set the value since the last module compute invocation. The value is automatically cleared to zero after the module is invoked. The *minval* and *maxval* arguments are ignored.

real
To specify an unlimited range of possible values, set both *minval* and *maxval* to the constant **FLOAT_UNBOUND**. Both *minval* and *maxval* must be either bounded or unbounded.

string
This is used for both simple strings and file pathnames. The value may be **NULL** in C, an empty string ("" or ' ' (single space) in FORTRAN), or an allocated string. FORTRAN must pass a valid string at least one character in length for the value to be recognized properly. Since trailing spaces are stripped off, a single space works as an empty string and is also handled properly when being used as a delimiter value. Widgets often present **NULL** values as "$NULL". For a text browser, *minval* is a comment character used to suppress display of text lines that begin with that character. For a file browser, *maxval* is a list of acceptable file types, separated by periods. For example, if *maxval* is ".x.image", only pathnames ending with *.x* or *.image* appear in the file browser attached to this parameter.

string_block
The value is a string that may contain embedded newline characters to delimit separate lines in a block of text. The entire value is displayed through several types of widgets for more extensive text output. In all other respects, it is equivalent to the string data type.

choice
The value is one of an enumerated set of strings. The *minval* argument is the set of possible choices separated by a delimiter character, such as "Alpha!Beta!Gamma". The *maxval* argument is the delimiter character, in this case "!". This delimiter should not be a newline "\n".

colormap
The *minval* and *maxval* arguments are ignored.

field
The only supported field type is "field 2D scalar real". This is used for handling 4 x 4 transformation matrices. The *minval* and *maxval* arguments are ignored.

delta_matrix_4x4
This is a synonym for "field 2D scalar real". This is used for handling the 4 x 4 delta matrices used by the Spaceball. The *minval* and *maxval* arguments are ignored.

AVSadd_float_parameter

C:
#include *<avs/avs.h>*
int AVSadd_float_parameter(*name, init, minval, maxval***)**
char **name;*
double *init, minval, maxval;*

This routine declares a parameter of type "real" for the module being defined in the current description function. The routine is an interface to the **AVSadd_parameter** routine; it allocates space for the *init, minval,* and *maxval* arguments automatically. The calling routine should declare these arguments as **float**. In C, when a float is passed as an argument, it is converted to a **double**.

There is no FORTRAN equivalent for this routine; use **AVSADD_PARAMETER** instead.

AVSadd_parameter_prop

C:
AVSadd_parameter_prop(*param_num, prop_name, prop_type, prop_value***)**
int *param_num;*
char **prop_name, *prop_type;*
<type> *prop_value;*

FORTRAN:
AVSADD_PARAMETER_PROP(*PARAM_NUM, PROP_NAME, PROP_TYPE, PROP_VALUE***)**
INTEGER *PARAM_NUM*
CHARACTER*n *PROP_NAME, PROP_TYPE*
INTEGER *PROP_VALUE*

This routine adds a property to a parameter for the module being defined in the current description function. A property usually determines some aspect of how the user interface presents the parameter. By calling this routine, a module can customize how the user interface handles the parameter.

The *param_num* argument is a parameter identifier returned by **AVSadd_parameter** or **AVSadd_float_parameter**. The *prop_name* argument is a string specifying the name of the property, and *prop_type* is a string specifying the type of property value being provided. The property type must be one of the parameter types. Each property has only one permissible property type, and AVS verifies that the *prop_type* is permissible for the *prop_name* supplied.

The *prop_value* argument is the value of the property. The storage type of *prop_value* is the storage type that corresponds to the property type. For a floating-point value, *prop_type* is a **float** rather than a **float *** in C.

As an example of using **AVSadd_parameter_prop**, assume that an integer parameter is attached to a dial widget. By default, when the user manipulates the widget, AVS reinvokes the module only when the user releases the mouse button. To cause AVS to reinvoke the module continually as the user manipulates the widget, the description function can use **AVSadd_parameter_prop** to attach an "immediate" property to the parameter. This property has a boolean value; a value of 1 causes continuous reinvocation as the mouse moves.

Some properties are not meaningful with all possible widgets. For example, the "immediate" property is not meaningful with a typein widget, since the module should be reinvoked only when the user has finished typing in the new value. If a call to **AVSadd_parameter_prop** requests a property or property value that a widget does not support, AVS ignores the request when it creates that widget. The property remains attached to the parameter, and AVS uses the property if the user attaches an appropriate widget at a later time.

Some widgets may allow the user to change properties interactively. When the user saves a network after making such a change, the property settings are saved as the user has modified them. When the saved network is subsequently read, the user's property settings override the values set by the call to **AVSadd_parameter_prop**.

The following table lists each available property name along with its property type, the C and FORTRAN data types of the property value, and the widget types that support the property:

Table A-2 Property Name, Data Type, and Widget Type Correspondence

Property Name	Property Type	C Data Type	FORTRAN Data Type	Widget Types
title	string	**char ***	**character*n**	dial, idial, slider, islider, toggle, tristate, oneshot, radio_buttons, browsers
immediate	boolean	**int**	**integer**	dial, idial, slider, islider
accumulator	boolean	**int**	**integer**	dial, idial
editable	boolean	**int**	**integer**	text
local_range	real	**float**	**real**	dial, idial

Table A-2 Property Name, Data Type, and Widget Type Correspondence

Property Name	Property Type	C Data Type	FORTRAN Data Type	Widget Types
width	integer	**int**	**integer**	toggle, tristate, oneshot, typein, text, browser, text_browser, radio_buttons, text_block_browser, textblock choice_browser
height	integer	**int**	**integer**	toggle, tristate, oneshot, typein, browser, text_browser, radio_buttons, text_block_browser, textblock choice_browser
columns	integer	**int**	**integer**	radio_buttons
layout	string_block	**char ***	**character*n**	any

The following are notes on some of these types:

title
This property specifies a title label for the widget. The default title is the parameter name.

immediate
A value of 0 means that AVS should reinvoke the module when the user has finished manipulating the widget (for example, by releasing the mouse button for a dial or slider). This is the default. A value of 1 means that AVS should continually reinvoke the module as the user manipulates the widget.

accumulator
This property is used with dial widgets. When the parameter bounds are fixed, a value of 0 means that the parameter range should map to one complete rotation of the dial. This is the default. A value of 1 means that the parameter range may extend over multiple rotations of the dial. When the parameter is unbounded, multiple dial rotations are always allowed.

editable
This property determines whether or not a text widget is editable in the Layout Editor. A value of 1, the default, specifies that the string is editable. A value of 0 specifies that the string is not editable. Text widgets are not editable outside the Layout Editor.

local_range
This property is used with dial widgets when the parameter is unbounded or when the "accumulator" property has a value of 1, allowing the parameter range to extend over multiple rotations of the dial. The value of the "local_range" property is the range that maps to one complete dial rotation. The default is 200.0.

width
This property specifies the width of the widget. The value is an integer between 1 and 20 inclusive and is interpreted as a multiple of the standard button width, which is approximately 60 pixels. (The application panel is just over 4 units wide.)

height
This property specifies the height of the widget. The value is an integer between 1 and 100 inclusive and is interpreted as a multiple of the height of a text line.

columns
This property specifies the number of columns of buttons in the widget. The default is 1.

layout
This property is used to specify user interface layout information in the form of CLI commands. See the CLI chapter for more information.

AVSautofree_output

C:
AVSautofree_output(*out_port***)**
int *out_port;*

FORTRAN:
AVSAUTOFREE_OUTPUT(*OUT_PORT***)**
INTEGER *OUT_PORT*

This routine tells AVS to free output data from the previous invocation before invoking the module being defined in the current description function. If neither this routine nor **AVSinitialize_output** is called, AVS does not free output data from the previous invocation when it invokes a module. The *out_port* argument is a port identifier returned by **AVScreate_output_port**.

AVSconnect_widget

C:
int AVSconnect_widget(*param_num, widget_type***)**
int *param_num;*
char **widget_type;*

FORTRAN:
INTEGER AVSCONNECT_WIDGET(*PARAM_NUM, WIDGET_TYPE***)**
INTEGER *PARAM_NUM*
CHARACTER*n *WIDGET_TYPE*

This routine allows you to select the kind of widget you want a parameter to connect to. However, AVS can connect a parameter only to a widget that is compatible with the parameter's data type. If a module calls this routine and selects an incompatible parameter type / widget combination, AVS ignores the call, issues a warning, and uses the default widget for that type of parameter. If a module makes no call to this routine, AVS uses the default widget for that parameter type.

AVSconnect_widget is a function, though its return value is not often used. (It returns 0 if successful, and -1 for an error.) Thus, it is usually called in examples as a subroutine. Note that some compilers may require that it be called as a function with a return value, i.e.:

```
i = AVSconnect_widget(...)
```

The *param_num* argument is a parameter identifier returned by **AVSadd_parameter** or **AVSadd_float_parameter**.

The *widget_type* argument is a string that indicates the type of widget to be connected to the parameter. If *widget_type* is "none", no widget is connected to the parameter. The following table lists the available widgets for each parameter type. If a parameter type has more than one possible widget, the widget type that appears first is the default. For more information on parameter types, see the documentation for **AVSadd_parameter**. You can use the **AVSadd_parameter_prop** and **AVSmodify_parameter_prop** routines to change a widget's appearance.

Table A-3 Parameter to Widget Correspondence

Parameter Type	Widget Type	Widget Description
[any]	none	[No widget]
integer	idial	Round dial with pointer; may be unbounded.
	islider	Fixed-length, left-to-right slider; must be bounded.
	typein_integer	Direct type-in with title.
boolean	toggle	On / off toggle switch.
tristate	tristate	Variant of toggle switch with 3 highlight states.
oneshot	oneshot	Button to request single actions
real	dial	Round dial with pointer; may be unbounded.
	slider	Fixed-length left-to-right slider; must be bounded.
	typein_real	Direct typein with title.
string	typein	Direct typein with title.
	text	String button, useful for titling; editable only in the Layout Editor.

Table A-3 Parameter to Widget Correspondence

Parameter Type	Widget Type	Widget Description
	browser	File browser. If the string is a pathname, the initial directory is set to the directory portion of the pathname.
	text_browser	ASCII file browser that displays the file specified by the string. Skips comment lines and filters out embedded *nroff* directives.
string_block	text_block_browser	Multiple line text browser. Scrolling text display window to display arbitrarily large amounts of text output. By default, the display is about twice the width of the AVS control panel and four lines high. Parameter properties can be used to create a browser of a specific height and width.
	text_block	Multiple line string block, useful for titling, or desriptions; not editable. By default it is the width of the AVS control panel and four lines high.
choice	radio_buttons	Set of radio buttons, one for each choice. The value is a copy of the selected string or **NULL** if no string is selected.
	choice_browser	Scrolling list of choices of arbitrary length similar to other browser widgets.
colormap	color_editor	Colormap editor.
field	track	Cursor-tracking virtual trackball.
delta_matrix_4x4	spaceball client	spaceball
delta_matrix_4x4	dials_matrix_client	dialbox

AVScreate_input_port

C:
#include *<avs/avs.h>*
int AVScreate_input_port(*name, type, flags***)**
char **name, *type;*
int *flags;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSCREATE_INPUT_PORT(*NAME, TYPE, FLAGS***)**
CHARACTER*n *NAME, TYPE*
INTEGER *FLAGS*

This routine declares an input port for the module being defined in the current description function. The name of the port is set to the string *name*. The *type* argument is a string that defines the data type of the port, as follows:

Table A-4 Data Port Type to String Correspondence

Data Type	*type* **String**
byte	"byte"
integer	"integer"
real	"real"
string	"string"
field	"field"
ucd	"ucd"
struct	"struct xxx"
molecule	"molecule"
colormap	"colormap"
geometry	"geom"
pixel map	"pixmap"

The "field" string can contain further specializing words; see the section "Declaring Fields" in the "AVS Data Types" chapter.

The *flags* argument consists of an OR'd combination of bitfields indicating optional properties of the input port. Currently supported values are: **REQUIRED**, meaning that a connection is required, **INVISIBLE** meaning that the port is invisible at start up time, and **MODIFY_IN** meaning that the module wishes to modify its input data.

If a port is **REQUIRED**, the module will not be invoked until a connection is made and there is data on the output port of the other end of the connection. The constant **OPTIONAL** can also be used with this argument but it is the default and is unnecessary.

The **MODIFY_IN** flag is used when the module needs to modify its input data. **MODIFY_IN** causes the module to receive a *copy* of the input data that it can change.

Note: Normally, modules should not directly modify an input, since the use of shared memory implies that all other modules sharing that port will see the modified data rather than the original data (usually an undesirable side effect). Also, it's generally considered bad practice to modify an input to a routine. However, there are some circumstances where it may be necessary. In this instance, do not use **MODIFY_IN**. Rather, just code the module to modify its input data. For the module to work, AVS will have to be run as either **-noshm** or **ReadOnlySharedMemory 0.**

This routine returns an integer identifier for the port that is used as an argument to some other AVS routines, such as **AVSinitialize_output**.

AVScreate_output_port

C:
int AVScreate_output_port(*name, type***)**
char **name, *type;*

FORTRAN:
INTEGER AVSCREATE_OUTPUT_PORT(*NAME, TYPE***)**
CHARACTER*n *NAME, TYPE*

This routine declares an output port for the module being defined in the current description function. The name of the port is set to the string *name*. The *type* argument is a string that defines the data type of the port. For possible values of the *type* argument, see the documentation for **AVScreate_input_port**.

This routine returns an integer identifier for the port that is used as an argument to some other AVS routines, such as **AVSset_output_flags**.

AVSinitialize_output

C:
AVSinitialize_output(*in_port, out_port***)**
int *in_port, out_port;*

FORTRAN:
AVSINITIALIZE_OUTPUT(*IN_PORT, OUT_PORT***)**
INTEGER *IN_PORT, OUT_PORT*

This routine tells AVS to preallocate memory for output data before invoking the module being defined in the current description function. Before each invocation of the module, AVS frees output data from the previous invocation and then allocates space for an output data structure of the same size and dimensions as those of the specified input data structure. AVS does not copy the input data to the output data. This is useful for modules that transform fields, producing an output field of the same type and dimensions as the input field. The *in_port* argument is a port identifier returned by **AVScreate_input_port**. The *out_port* argument is a port identifier returned by **AVScreate_output_port**.

AVSload_user_data_types

C:
AVSload_user_data_types(*filename***)**
char **filename;*

FORTRAN:

AVSLOAD_USER_DATA_TYPES(*FILENAME***)**
CHARACTER*n *FILENAME*

This routine specifies a filename containing a description of one or more user-defined data types. It is usually called during the description function of the module. The filename is either an absolute pathname of the file or, if a relative pathname, the path is interpreted as relative to the directory */usr/avs/include*. See the section on "User-Defined Data Types" in Chapter 4 for more information on the contents of this file and how modules use these data types. Also, see the program */usr/avs/examples/pick_cube.c* for an example of using a user-defined data type for passing upstream data. See the */usr/avs/examples/user_data.c* and */usr/avs/examples/user_data.f* programs for more general examples of using user-defined data.

AVSset_compute_proc

C:
AVSset_compute_proc(*comp_func***)**
int *(*comp_func)();*

FORTRAN:
AVSSET_COMPUTE_PROC(*COMP_FUNC***)**
EXTERNAL *COMP_FUNC*

This routine declares the computation function for the module being defined in the current description function.

AVSset_destroy_proc

C:
AVSset_destroy_proc(*destroy_func***)**
int *(*destroy_func)();*

FORTRAN:
AVSSET_DESTROY_PROC(*DESTROY_FUNC***)**
EXTERNAL *DESTROY_FUNC*

This routine declares an optional destruction function for the module being defined in the current description function. AVS invokes the destruction function when the module is destroyed, usually when the user moves the module icon from the workspace to the "hammer" icon. A destruction function might take actions such as freeing memory or destroying a window.

AVSset_init_proc

C:
AVSset_init_proc(*init_func***)**

int *(*init_func)();*

FORTRAN:
AVSSET_INIT_PROC(*INIT_FUNC***)**
EXTERNAL *INIT_FUNC*

This routine declares an optional initialization function for the module being defined in the current description function. AVS invokes the initialization function when the module is instantiated (usually when the user moves the module icon from the module palette into the workspace). An initialization function might take actions such as allocating memory or creating a window. Since this routine is called during the creation of the module, it cannot use calls to the AVS kernel that depend on the existance of a fully initialized module. Some examples of such calls are: **AVSmessage**, **AVScommand**, and **AVSmodify_parameter**.

AVSset_input_class, AVSset_output_class, AVSset_parameter_class

C:
AVSset_input_class(*port*, *class***)**
int *port;*
char **class;*

AVSset_output_class(*port*, *class***)**
int *port;*
char **class;*

AVSset_parameter_class(*port*, *class***)**
int *port;*
char **class;*

FORTRAN:
AVSSET_INPUT_CLASS(*PORT*, *CLASS***)**
INTEGER *PORT*
CHARACTER*n *CLASS*

AVSSET_OUTPUT_CLASS(*PORT*, *CLASS***)**
INTEGER *PORT*
CHARACTER*n *CLASS*

AVSSET_PARAMETER_CLASS(*PORT*, *CLASS***)**
INTEGER *PORT*
CHARACTER*n *CLASS*

These routines set the port class for an input, output or parameter port. The "class" for a port or parameter is used to make automatic upstream connections when particular downstream connections are made.

The *port* argument should be the integer value returned from: **AVScreate_input_port**, **AVScreate_output_port**, or **AVSadd_parameter** for **AVSset_input_class**, **AVSset_output_class**, and **AVSset_parameter_class** respectively.

The *class* argument is a character string that contains a class name and an optional port name to associate it with. The class name is determined by convention between the upstream and downstream module. This name is often the name of the data type of the downstream connection.

An optional port name can be specified as part of the "class" character string. If so, a ":" character separates the port name from the class name. If the port name is specified, it indicates that the upstream connection should only be made if the downstream port is being connected.

See the section on automatic connection of ports in the chapter "Advanced Topics" for more information and examples on how to use port classes to cause automatic connections.

AVSset_module_flags

C:
#include <*avs/avs.h*>
AVSset_module_flags (*flag***)**
  unsigned int *flags;*

FORTRAN:
#INCLUDE <*avs/avs.inc*>
AVSSET_MODULE_FLAGS(*FLAG***)**
  INTEGER *FLAGS*

This routine specifies a number of special options for how the module is to be handled or how it receives data during computation. The *flags* argument consists of an OR'd combination of bitfields indicating optional properties of the module. The following flags are defined:

Table A-5 Module Flags and Meaning

Flag	Meaning
COOPERATIVE	Module can run with others in the executable
REENTRANT	Module can run with itself in the executable
SINGLE_ARG_DATA	Module expects data to be passed as a single argument
SINGLE_ARG_FIELD	Module expects field to be passed as a single argument
COROUT_UNPACK_ARGS	Used to request that strings and reals be copied into local variables directly for FORTRAN coroutines.

The SINGLE_ARG_DATA flag requests that input and output fields be passed as single arguments instead of multiple arguments to compute functions written in FORTRAN. Colormaps and user-defined data types are also affected by this flag.

The SINGLE_ARG_FIELD flag is similar to the SINGLE_ARG_DATA flag except it does not affect colormaps or user-defined data types.

For FORTRAN coroutines, the COROUT_UNPACK_ARGS flag asks that all strings and reals be copied into local variables directly instead of being passed as pointers to strings and reals. See the description of **AVScorout_input**.

AVSset_module_name

C:
#include *<avs/avs.h>*
AVSset_module_name(*name, type***)**
char **name;*
int *type;*

FORTRAN:
AVSSET_MODULE_NAME(*NAME, TYPE***)**
CHARACTER*n *NAME, TYPE*

This routine declares the name and type of the module being defined in the current description function. The module name is set to the string *name* and the type to *type*, where *type* is one of the following:

Table A-6 Module Types and C/FORTRAN Descriptions

Module Type	C Constant	FORTRAN string
Data Input	**MODULE_DATA**	'data'
Filter	**MODULE_FILTER**	'filter'
Mapper	**MODULE_MAPPER**	'mapper'
Renderer	**MODULE_RENDERER**	'renderer'

The module name appears in the module icon and other portions of the Network Editor. The module type determines the category in the Network Editor module palette in which the module icon appears.

AVSset_output_class

See the description at **AVSset_input_class**.

AVSset_output_flags

C:
#include *<avs/avs.h>*
int AVSset_output_flags(*port, flags***)**
int *port, flags;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSSET_OUTPUT_FLAGS(*PORT, FLAGS***)**
INTEGER *PORT, FLAGS*

This is used by a module in the description function to set some optional properties of an output port. Currently the only flag that is supported is the flag **INVISIBLE**. This flag causes the output port to be invisible by default.

AVSset_parameter_class

See the description at **AVSset_input_class**.

Routines for Modifying and Interpreting Parameters

You can use the routines in this section only during compute functions.

AVSchoice_number

C:
#include *<avs/avs.h>*
int AVSchoice_number(*name, string***)**
char **name, *string;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSCHOICE_NUMBER(*NAME, STRING***)**
CHARACTER*n *NAME, STRING*

This routine is called to interpret a value for a parameter of type "choice" passed to a module computation routine. The *name* argument is the name of the parameter as declared in the call to **AVSadd_parameter** in the module description function. The *string* argument must be a valid value for the choices allowed or a NULL string.

This routine returns an integer that represents the position of the given choice in the list of choices provided in the call to **AVSadd_parameter** in the module description function. If the choice is the first in the list, this routine returns 1; if the choice is the second in the list, this routine returns 2; and so on. If the choice is not in the list of choices, this routine returns 0.

A module computation function can also interpret choices by means of direct string comparisons of the parameter argument with expected literal strings.

AVSmodify_float_parameter

C:
#include *<avs/avs.h>*
AVSmodify_float_parameter(*name, flags, init, minval, maxval***)**
char **name;*
int *flags;*
double *init, minval, maxval;*

This routine is called from a module computation routine to change the value or bounds of a parameter of type "real". The routine is an interface to the **AVSmodify_parameter** routine; it allocates space for the *init, minval,* and *maxval* arguments automatically. The calling routine should declare these arguments as **float**. In C, when a float is passed as an argument it is converted to a **double**.

See the **WARNING** in the ***AVSmodify_parameter*** routine descripton.

There is no FORTRAN equivalent for this routine; use **AVSMODIFY_PARAMETER** instead.

AVSmodify_parameter

C:
#include *<avs/avs.h>*
AVSmodify_parameter(*name, flags, init, minval, maxval***)**
char **name;*
int *flags*
<type> init, minval, maxval;

FORTRAN:
#include *<avs/avs.inc>*
AVSMODIFY_PARAMETER(*NAME, FLAGS, INIT, MINVAL, MAXVAL***)**
CHARACTER*n *NAME*
INTEGER *FLAGS*
<type> INIT, MINVAL, MAXVAL

This routine is called from a module computation routine to change the value or bounds of a parameter. AVS first updates the parameter bounds and then checks the new or existing value for validity against the new bounds. If a widget is connected to the parameter, the widget is then updated to reflect the new parameter bounds and value.

The *name* argument is the name of the parameter as declared in the call to **AVSadd_parameter** or **AVSadd_float_parameter** in the module description function.

The *flags* argument is a bit mask indicating which combination of value, upper bound, and lower bound is to be changed. AVS defines the following constants corresponding to the three items to be changed:

AVS_VALUE
The *init* argument contains a new value for the parameter.

AVS_MINVAL
The *minval* argument contains a new minimum value for the parameter.

AVS_MAXVAL
The *maxval* argument contains a new maximum value for the parameter.

AVS_RECORD_VALUE
The value change provided in the **init** argument should be recorded if a CLI script is being generated. This is useful for modules with their own non-AVS user interface widgets as a way to record activity.

These constants can be combined using a bitwise OR operation to change more than one item at a time. For example, to change the value and upper bound but not the lower bound:

```
      /* C language */
      flags = AVS_VALUE | AVS_MAXVAL;
C     FORTRAN
      INTEGER FLAGS
      FLAGS = IOR(AVS_VALUE, AVS_MAXVAL)
```

AVS changes the value or a bound of a parameter only if the corresponding bit in the *flags* argument is on, or if a change in the bounds requires changing the current value of the parameter to be within the new bounds.

The *init*, *minval*, and *maxval* arguments are interpreted in the same way as the corresponding arguments to **AVSadd_parameter**. Note that the meaning and type of these arguments depend on the parameter type.

The *init, minval,* and *maxval* arguments are cast as **int**s in C and **integer**s in FORTRAN, but their storage type actually depends on the parameter type. For any type of parameter, *init, minval,* and *maxval* all have the same storage type. For example, if the parameter is of type "string", all three values must be char* values (or CHARACTER*(*) in FORTRAN). Each value must fit into an integer-size memory slot or must be a pointer to a larger memory allocation. Values representing **float**s in C must be pointers to allocated memory. The routine **AVSadd_float_parameter** handles this allocation automatically. FORTRAN does not have this special requirement.

If the call to **AVSmodify_parameter** does not change the value, lower bound, or upper bound, the corresponding *init*, *minval*, or *maxval* argument should be NULL in C (0 in FORTRAN).

WARNING

The arguments to the module computation routine are essentially a "snapshot" of the parameter values at the time the computation routine is called. This means that **AVSmodify_parameter** affects the value and range of the parameter the next time the computation routine is called; it does not necessarily affect the corresponding argument value within the current invocation of the routine. (It may in some cases, particularly floats and strings.)

If you intend to perform further computations on an argument whose corresponding parameter you change with **AVSmodify_parameter**, make a local copy of the argument before calling **AVSmodify_parameter**, apply the same changes to the copy argument, and then perform further computations with the copy, not the original.

AVSmodify_parameter_prop

C:
#include *<avs/avs.h>*
AVSmodify_parameter_prop(*name, prop_name, prop_type, prop_value***)**
char **name, *prop_name, *prop_type;*
int *prop_value;*

FORTRAN:
#include *<avs/avs.inc>*
AVSMODIFY_PARAMETER_PROP(*NAME, PROP_NAME, PROP_TYPE, PROP_VALUE***)**
CHARACTER*n *NAME, PROP_NAME, PROP_TYPE*
INTEGER *PROP_VALUE*

This routine is used to modify a parameter property during computation. The *prop_value* argument is treated exactly like the same argument in AVSadd_parameter_prop. Unlike that function, this one takes the parameter name since it can be used only by the compute function. The widget attached to the parameter may change immediately as in the case of changing the title property of a parameter. In other cases it has no apparent effect but causes no damage either. The property does not have to have been created by AVSadd_parameter_prop first.

AVSparameter_visible

C:
#include *<avs/avs.h>*
AVSparameter_visible(*name, stat***)**
**char ***param;*
int *stat;*

FORTRAN:
#include *<avs/avs.inc>*

AVSPARAMETER_VISIBLE(*NAME, STAT* **)**
CHARACTER*n *PARAM*
INTEGER *STAT*

This routine controls the visibility of the widget attached to a parameter. The *name* argument is the name of the parameter and the *stat* value is 0 for invisible and 1 for visible. This routine should be used sparingly in those situations where certain parameters are not meaningful or valid to modify until other parameters are set to reasonable values. The best way to use this function is make undesired widgets invisible in the initial compute function call, once they have been allocated space in the control panel and then make them visible when appropriate.

Routines for Coroutine Modules

AVScorout_event_wait

C:
AVScorout_event_wait(*nfds,readfds,writefds,exceptfds,timeout,mask***)**
int *nfds;*
fd_set *readfds,writefds,exceptfds;*
struct timeval **timeout;*
int **mask;*

This routine is used by a coroutine module that needs to simultaneously wait for data on one or more file descriptors or for its inputs and/or parameters to change. It can also be used by a module that does not have any file descriptors, but wants to wait for inputs and parameters to change with a timeout value.

On most systems, this routine uses the "select" system call. It was designed to mimic the functionality of this utility as much as possible. See the "select" man page for a complete description of the functionality, including error conditions, etc. The only difference between this routine and the "select" routine is that it takes an additional parameter which is the "mask" of coroutine events to wait for. Currently only one coroutine event is supported: **COROUT_WAIT**.

The *mask* argument, therefore, should be set to **COROUT_WAIT** before the call. If a coroutine event occurred, the corresponding bit will be set to indicate this in the "mask" parameter after the routine returns. The return value indicates the number of events that are ready, including the coroutine event.

The "timeout" parameter behaves just like select. If the value is 0, the routine blocks until either input or an error occurs. If the value points to a structure, the structure specifies the time in seconds and microsecond in which to wait. The **timval** structure is defined in the include file: **<sys/time.h>**.

There is no FORTRAN equivalent for this routine.

AVScorout_exec

C:
AVScorout_exec()

FORTRAN:
AVSCOROUT_EXEC()

This routine waits until the flow executive has stopped running. It then returns. The routine is useful for delaying output until the network has completely processed the output of the previous computation.

AVScorout_init

C:
AVScorout_init(*argc, argv, desc***)**
int *argc;*
char **argv[];*
int *(*desc)();*

FORTRAN:
AVSCOROUT_INIT(*DESC***)**
EXTERNAL *DESC*

This routine causes AVS to recognize and initialize the coroutine as a module and sets up the connection between the coroutine and AVS. The coroutine must call **AVScorout_init** before calling any other AVS routines. If this routine is invoked during the module identification pass, it exits; if the routine is invoked during module instantiation, it returns.

For a C coroutine, the *argc* and *argv* arguments are the corresponding arguments to the coroutine main program. The *desc* argument is a pointer to the module description function. For a FORTRAN coroutine, the only argument is the module description function; AVS automatically picks up the program arguments.

AVScorout_input

C:
int AVScorout_input(*input1, input2, ..., param1, param2, ...***)**
*<type> **input1, **input2, ...;*
*<type> *param1, *param2, ...;*

FORTRAN:
INTEGER AVSCOROUT_INPUT(*INPUT1, INPUT2, ...,*

*PARAM1, PARAM2, ...***)**
<*type*> *INPUT1, INPUT2,...*
<*type*> *PARAM1, PARAM2, ...*

A coroutine calls this routine to obtain inputs and parameters from AVS. There is one argument for each input port and one argument for each parameter declared in the module description function. All the input arguments appear first in the arglist, followed by all the parameter arguments. For most data types, the argument is a pointer to a pointer to a data item of the appropriate type for the input or parameter declared. For some data types, such as integers, the argument is a pointer to the data item itself. When the function returns, each argument location contains a pointer to the corresponding input or parameter value (or the value itself, for data types like integers).

In FORTRAN, most arguments are handled as integers. Simple integer data types are handled directly as values and complex data types, such as fields and colormaps, are handled using accessor functions (See the routines in this appendix in the following sections: Field Accessor Routines, Colormap Accessor Routines, User Data Accessor Routines, FORTRAN Array Accessor Routines, and FORTRAN Single Byte Accessor Routines.) Character strings are handled by passing in a buffer large enough for the expected value; the character data is copied into the provided buffer. The real data type is handled directly; the argument value is copied into the provided real.

The routine returns 0 if a required input or parameter is missing. Otherwise, it returns the number of inputs and parameters supplied.

Under AVS, strings and reals are passed as C pointers which are awkward to handle in FORTRAN. This is the default mode for backward compatibility. To have strings copied into buffers and reals copied in directly, you MUST set the module flag, COROUT_UNPACK_ARGS, using the **AVSset_module_flags** routine.

AVScorout_mark_changed

C:
AVScorout_mark_changed()

FORTRAN:
AVSCOROUT_MARK_CHANGED()

This routine marks the module as having changed since the last call to **AVScorout_input**. The module will continue to be considered "changed" until the next call to **AVScorout_input** (or **AVScorout_output** for modules that have no inputs and parameters).

This routine can be used by coroutine modules that want to run continuously as it will cause the routine **AVScorout_wait** to return rather than wait for the next input or parameter to change.

AVScorout_output

C:
AVScorout_output(*output1, output2, ...***)**
<*type*> **output1, *output2, ...;*

FORTRAN:
AVSCOROUT_OUTPUT(*OUTPUT1, OUTPUT2, ...***)**
<*type*> *OUTPUT1, OUTPUT2, ...*

A coroutine calls this routine to send output data to AVS. There is one argument for each output port declared in the module description function. For most data types, the argument is a pointer to a data item of the appropriate type for the output declared. For some data types, such as integers, the argument is the data item itself.

In FORTRAN, integers and complex data types are passed in as integers (where necessary, use the **AVSdata_alloc** routine to allocate a complex data type). Character strings are passed using local string buffers; the string data is copied out of the buffer into the port data structure automatically. Reals are passed directly; their values are copied into the port data structure automatically.

If the user has disabled the module or the flow executive, this routine may hang for an arbitrary time before returning.

Under AVS, strings and reals are passed as C pointers which are awkward to handle in FORTRAN. This is the default mode for backward compatibility. To have strings copied into buffers and reals copied in directly, you MUST set the module flag, COROUT_UNPACK_ARGS, using the **AVSset_module_flags** routine.

AVScorout_set_sync

C:
AVScorout_set_sync(*value***)**
int *value;*

FORTRAN:
AVSCOROUT_SET_SYNC(*VALUE***)**
INTEGER *VALUE*

This routine allows you to force a coroutine module execute synchronously and thereby under the control of the flow-executive. Set the *value* parameter as follows: 0 Execute asynchronously 1 Execute synchronously

A coroutine module can call this routine any time after it calls **AVScorout_init**. Typically, the module calls this routine only once.

By default, coroutine modules run asynchronously. This means that coroutine modules can run in parallel with other coroutine modules or other subroutine modules. Sometimes, this may be the desired behavior. However, in certain situations, modules that execute in parallel can cause the AVS network to behave unpredictably and / or might cause the network to execute downstream modules more than once.

Well behaved coroutine modules can be run synchronously. To run synchronously means that except when the coroutine module is waiting in **AVScorout_wait, AVScorout_X_wait**, or **AVScorout_event_wait**, the flow-executive executes no other AVS module. This results in the network having a predictable order of execution.

AVScorout_wait

C:
AVScorout_wait()

FORTRAN:
AVSCOROUT_WAIT()

This routine waits until the module is "changed" and has been "scheduled" by the flow executive. A module is defined as "changed" when an input or parameter has been modified or it is marked as changed by the module with the **AVScorout_mark_changed** routine.

The module is scheduled by the flow executive when the module is the next changed module in the run queue. The run queue is only processed when the flow executive is enabled.

This routine will continue to return until the routine **AVScorout_input** has been called. If the routine has no inputs or parameters the **AVScorout_output** routine will mark the module as "unchanged".

AVScorout_X_wait

C:
AVScorout_X_wait(*dpy, timeout, mask***)**
Display **dpy;*
struct timeval **timeout;*
int **mask;*

This routine is used by a coroutine module that needs to simultaneously wait for inputs and / or parameters to change and for X events / errors.

The *dpy* argument is the X display on which X events / errors are expected.

The *timeout* argument is a pointer to a "timeval" structure. This structure is defined in the include file: **<sys/time.h>**. and has two fields: **tv_sec** and **tv_usec** which describe the number of seconds and microseconds to wait respectively. If the pointer to the timeval structure is **NULL**, the routine will wait indefinitely, otherwise the timeval structure contains a number of seconds and a number of microseconds to wait before timing out. A structure containing 0 seconds and 0 microseconds can be used to poll the X socket and the state of AVS inputs / parameters.

The *mask* parameter is a pointer to an integer containing the coroutine events to wait for. Currently there is only one coroutine event: **COROUT_WAIT**. The *mask* argument, therefore, should always be set to the value: **COROUT_-WAIT**.

This routine will return a "1" if there are X events / errors waiting to be processed. The flags set in *mask* will indicate the state of the coroutine events that were waited for. Since **COROUT_WAIT** is the only supported event, the value will either be 0, the module was not scheduled to be executed or the value: **COROUT_WAIT** (the inputs / parameters did change for this module).

If **AVScorout_X_wait** returns 0 and the value of mask returned is 0, then the routine timed out with the timeout value specified.

There is no FORTRAN equivalent for this routine.

Status Monitoring Routine

AVSmodule_status

C:
AVSmodule_status(*comment, percent***)**
**char ****comment;*
int *percent;*

FORTRAN:
AVSMODULE_STATUS(*COMMENT, PERCENT***)**
CHARACTER*n *COMMENT*
INTEGER *PERCENT*

This routine sends the kernel status updates when a long operation is in progress and is of predictable length. The information may be displayed in the status bar on the main control panel to inform the user of incremental progress. A module's status is considered to be broken into input transmission (0 - 10%), module operation (10-90%) and output processing (90-100%). The status *percent* argument is given in terms of 0-100% of the module operation and thus shows up as changes between 10 and 90% of the overall operation. The *comment* is shown in the status bar for particularly long operations to show the intermediate operation in progress. For shorter operations, only

the module name might actually show up. If no status calls are made, the status bar does not show any intermediate progress between the 10 and 90% mark.

AVS Command Language Interpreter Routine

The AVS Command Language Interpreter (CLI) allows you write ASCII scripts that can control most AVS systems. With the CLI, you can save AVS networks, widget layouts, parameter settings, and can record a sequence of user interactions. Individual modules can also send CLI commands to AVS. Allowing modules to issue CLI commands provides opportunities for AVS application modules to manage AVS network execution in response to changes in their own parameters. By preprogramming a set of instructions that change the relevant parameters, you can create animations using AVS. For more information on using CLI, see the "Command Language Interpreter" chapter.

AVScommand

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, ERROR_BUFFER*

Use this routine to send Command Language Interpreter (CLI) commands to the AVS kernel. See the CLI chapter for a list of available CLI commands.

The *destination* argument can have only the value "kernel".

The *command_buffer* argument specifies a buffer that contains one or more CLI commands. You can include multiple commands in the same command buffer by separating these commands with newline characters.

The *output_buffer* and *error_buffer* arguments are used to receive output from commands and from errors, respectively. In C, each of these two arguments are provided as the address of a char pointer (char *) which will be changed to point to the actual buffer by the routine. (i.e. declare a char * variable ("buf") and pass its address ("&buf").) Memory management for the buffers is provided 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. Select a <*maxsize*> dimension for these buffers that is adequate to hold the expected output. Output that exceeds the specified size is lost. The output buffers contain the accumulated output and error messages resulting from issuing all the commands in the command buffer.

Multiple commands can be included in the same command buffer and should be separated by newline characters. The accumulated output and 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 needed during an **AVScommand** call.

The CLI command, **debug**, 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".

Routines for Selective Computation

AVSinput_changed

C:
int AVSinput_changed(*port_name, i***)**
char **port_name;*
int *i;*

FORTRAN:
INTEGER AVSINPUT_CHANGED(*PORT_NAME, I***)**
CHARACTER*n *PORT_NAME*
INTEGER *I*

This routine determines whether or not input data has changed since the previous invocation of the module. The *port_name* argument is the name of the input port as declared in the module description function. The second argument is the number of a connection to that port; the first connection is 0 for the C routine and 1 for the FORTRAN routine. **AVSinput_changed** returns 1 if the input data has changed for the specified port and connection. It returns 0 if the input has not changed or if the specified connection does not exist.

AVSmark_output_unchanged

C:
AVSmark_output_unchanged(*port_name***)**
char **port_name;*

FORTRAN:
AVSMARK_OUTPUT_UNCHANGED(*PORT_NAME***)**
CHARACTER*n *PORT_NAME*

By default, AVS assumes that all output data has changed after each invocation of a module. This can cause AVS to invoke downstream modules. **AVSmark_output_unchanged** tells AVS that output data for a port has not changed. The *port_name* argument is the name of the output port as declared in the module description function.

AVSparameter_changed

C:
int AVSparameter_changed(*param_name***)**
char **param_name;*

FORTRAN:
AVSPARAMETER_CHANGED(*PARAM_NAME***)**
CHARACTER*n *PARAM_NAME*

This routine determines whether or not a parameter value has changed since the previous invocation of the module. The *param_name* argument is the name of the parameter as declared in the module description function. **AVSparameter_changed** returns 1 if the parameter value has changed. It returns 0 if the parameter value has not changed.

Routines for Creating Fields

Use the routines described in this section to construct the AVS field data type. Refer to the "AVS Data Types" chapter for more information on the field data type. You should not develop new modules that use the **AVSbuild_field, AVSbuild_2d_field,** and **AVSbuild_3d_field** routines. These routines cannot use shared memory and may be removed from future releases of AVS. Instead, use the **AVSdata_alloc** and **AVSfield_alloc** routines described in this section.

AVSport_field

FORTRAN:
#include <*avs/avs.inc*>

INTEGER AVSPORT_FIELD(*PORT_NAME***)**
CHARACTER*n *PORT_NAME*

This routine is used by FORTRAN module writers using the old approach of passing fields to the computation routine as multiple arguments. It returns the field pointer required by the new field accessor functions. The integer value returned by the function is the associated field pointer or 0 if there is no valid field data associated with that port.

When fields are passed as single arguments, the field pointer is passed directly as an argument to the computation function. See the documentation for **AVSset_module_flags** for a description of how to request single argument passing.

AVSdata_alloc

C:
#include *<avs/avs.h>*
char * AVSdata_alloc(*string, dims***)**
char **string;*
int **dims;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSDATA_ALLOC(*STRING, DIMS***)**
CHARACTER*n *STRING*
INTEGER *DIMS(*ndim*)*

This routine is similar to **AVSfield_alloc**, except it takes a character string describing the field, rather than a field template structure. In C, it returns a pointer to a **char**, which should be cast to a pointer to an **AVSfield** of the correct type (**AVSfield_char**, **AVSfield_float**, etc.). In FORTRAN, it returns an integer that you can use with the field accessor routines.

You can also use this routine to allocate other complex data types, such as colormaps and user data types.

The *dims* argument is an array of integers specifying the desired dimensions of the data; it is used to preallocate storage.

C example:

```
field = (AVSfield_char *)
        AVSdata_alloc("field 2D 4-vector byte", dims);
```

FORTRAN example:

```
ifield = AVSDATA_ALLOC('field 2D 4-vector byte', idims)
```

AVSdata_alloc returns a NULL value if the field allocation fails. It could fail, for example, allocating memory for the data. Programs should check for this

eventuality. The following C fragment contains a typo in the field description that will cause the allocation to fail.

```
AVSfield_char **yourfield;                    /*declare return value variable*/
        .
        .
        .
if (*yourfield) AVSfield_free(*yourfield);  /*free data from previous invocation*/
dims0[0] = 100;
dims0[1] = 200;
*yourfield = (AVSfield_char *)                        /*allocate new data area*/
    AVSdata_alloc("field 2D 2-space 4-vector uniform bite"), dims0);
if  (*yourfield == NULL) {
    AVSerror("Allocation of output field failed.");
    return(0);
    }
```

AVSdata_free

C:
#include *<avs/avs.h>*
void AVSdata_free(*type, data_ptr***)**
  char **type, *data_ptr;*

FORTRAN:
#include *<avs/avs.inc>*
AVSDATA_FREE(*TYPE, DATA_PTR***)**
  CHARACTER*n *TYPE*
  INTEGER *DATA_PTR*

This routine frees all memory associated with a data *type*. This *type* is the same string that was usen in the **AVSdata_alloc** call to create the data. When **AVSdata_alloc** is used to create a field, the string includes the word "field" plus various field descriptors such as "2D uniform". When you are freeing the data, you should include only the string "field" without the other descriptors. The *data_ptr* is the pointer that the **AVSdata_alloc** call returned when the data structure was created. The following FORTRAN example would free the field created in the example under **AVSdata_alloc** above:

```
AVSDATA_FREE ('field', ifield)
```

AVSfield_alloc

C:
#include *<avs/field.h>*
char * AVSfield_alloc(*template, dims***)**
  AVSfield **template;*
  int **dims;*

FORTRAN:

#include <*avs/avs.inc*>
INTEGER AVSFIELD_ALLOC(*TEMPLATE, DIMS***)**
INTEGER *TEMPLATE*
INTEGER *DIMS(*ndim*)*

This routine creates and allocates memory for a field. In C, it returns a pointer to a **char**, which should be cast to a pointer to an **AVSfield**.

In FORTRAN, use the integer this routines returns with the Field Accessor Routines.

The *template* argument is a pointer to a field to be used as a template for creating the new field. The *dims* argument is an array of integers to be used as the dimensions of the new field in computational space. The length of the array must be the same as the number of dimensions in the template field. The *dims* argument can also be 0; in this case, the dimensions of the template field are used to create the new field.

This routine copies the *nspace*, *veclen*, *type*, *size*, and *uniform* members of the template field to the new field. If the *dims* argument is 0, it copies the *dimensions* array of the template field to the new field; otherwise, it copies the *dims* argument to the *dimensions* array of the new field. This routine allocates memory for the *points* array of the new field. If the template field is rectilinear or irregular and if the template field has a *points* array, this routine copies the *points* array of the template field to the new field. This routine allocates memory for the *data* array of the new field but does not copy the *data* array of the template field to the new field.

The template field can be an existing field, such as an input argument to a module computation routine, or a template created from an existing field by **AVSfield_make_template**. A template created by **AVSfield_make_template** is useful when the *points* array of the template field is not to be copied to the new field.

Note: AVSfield_alloc does not copy the *labels*, *units*, *min*, *max*, *min_ext*, or *max_ext* from the template field. It only creates pointers to zero length strings or null values. These values should be set on the new field with **AVSfield_set_labels/units**/**minmax/extent**. If you wish the new values to equal those of the old template field, first use **AVSfield_get_labels/units/**...etc. to establish the values, then assign them with **AVSfield_set_labels/units/**...etc. To ensure the integrity of field memory allocation, you should always use these accessor functions rather than attempting to manipulate the field directly.

AVSfield_alloc returns a NULL value if the field allocation fails. Programs should check for this eventuality. See the example under **AVSdata_alloc.**

AVSfield_copy_points

C:
#include <*avs/field.h*>

int AVSfield_copy_points(*field_in, field_out***)**
AVSfield **field_in, *field_out;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_COPY_POINTS(*FIELD_IN, FIELD_OUT***)**
INTEGER *FIELD_IN, FIELD_OUT*

This routine copies the coordinates array from *field_in* to *field_out*. Memory must be allocated for the coordinates array in *field_out* before this routine is called. This routine is useful for passing the coordinates array from an input field to an output field in a module computation routine that operates only on the computational data of a field and ignores the coordinates. 1=success; 0=failure.

AVSfield_free

C:
#include <*avs/field.h*>
void AVSfield_free(*field***)**
AVSfield **field;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_FREE(*FIELD***)**
INTEGER *FIELD*

This routine frees all memory associated with a field. The *field* variable is whatever pointer variable was returned by the **AVSfield_alloc** routine that created the field.

AVSfield_make_template

C:
#include <*avs/field.h*>
AVSfield_make_template(*field_in, template***)**
AVSfield **field_in, *template;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_MAKE_TEMPLATE(*FIELD_IN, TEMPLATE***)**
INTEGER *FIELD_IN, TEMPLATE*

This routine copies the *ndim*, *nspace*, *veclen*, *type*, *size*, and *uniform* members of *field_in* to *template*. It allocates memory for the *dimensions* array of the template field and copies the *dimensions* array of *field_in* to the template field. This routine does not allocate memory for the *data* and *points* arrays of the template field; it sets the value of those members of the template field to NULL.

This routine is intended to use an existing field, such as an input argument to a module computation routine, to create a template for **AVSfield_alloc**. The *template* argument can be created as follows:

```
AVSfield *template;
template = (AVSfield *) malloc(sizeof(AVSfield));
```

The FORTRAN routine makes a template field that you can modify using the **AVSFIELD_SET_INT** routine and then use it in **AVSFIELD_ALLOC**. If the initial value of the template argument is 0, the template structure is allocated automatically.

NOTE:

As previously stated, you should not develop new modules that use the **AVSbuild_field, AVSbuild_2d_field,** and **AVSbuild_3d_field** routines. These routines cannot use shared memory and may be removed from future releases of AVS. Instead, use the **AVSdata_alloc** and **AVSfield_alloc** routines described in this section.

AVSbuild_field

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
AVSfield * AVSbuild_field(*ndim, veclen, uniform, ncoord, type, dim1, dim2, ..., data, coords***)**
int *ndim, veclen, uniform, ncoord, type;*
int *dim1, dim2, ...;*
unsigned char **data;*
float **coords;*

FORTRAN:
#include *<avs/avs.inc>*
AVSBUILD_FIELD(*NDIM, IVLEN, IFLAG, NCOORD, ITYPE, IDIM1, IDIM2, ..., DATA, COORDS***)**
INTEGER *NDIM, IVLEN, IFLAG, NCOORD, ITYPE*
INTEGER *IDIM1, IDIM2, ...*
BYTE *DATA(*)*
REAL *COORDS(*)*

NOTE:

This routine is provided for backward compatibility with AVS2 only. New modules should use the **AVSdata_alloc** and **AVSfield_alloc** calls described in this section and in Chapter 2 under the heading "Creating Fields" instead. Modules that use the **AVSbuild...** series of calls cannot use shared memory. These routines may be removed from future releases of AVS.

This routine is a utility that constructs a field from its components. The routine returns a pointer to an **AVSfield** structure. Following is a description of the arguments:

ndim
A positive integer specifying the number of dimensions in the computational space of the field.

veclen
A positive integer specifying the length of the data vector at each point. For a scalar field, the value is 1.

uniform
A constant specifying whether the field is uniform, rectilinear, or irregular. Possible values are **UNIFORM**, **RECTILINEAR**, and **IRREGULAR**.

ncoord
An integer specifying the number of dimensions in the coordinate space of nonuniform fields. For uniform fields, the value is 0. For rectilinear fields, the value is the same as *ndim*.

type
A constant specifying the type of data in the field. Possible values are **AVS_TYPE_BYTE**, **AVS_TYPE_INTEGER**, **AVS_TYPE_REAL**, and **AVS_TYPE_DOUBLE**.

dim1, dim2, ...
For each dimension, an integer specifying the size of the dimension.

data
The data array, in "FORTRAN" order. The subscript for vector element varies fastest, then the subscript for the first dimension, then the subscript for the second dimension, and so on. The storage type for each element depends on the data type of the field.

coords
For a nonuniform field, an array of floating-point values specifying the coordinates of the data points. For a rectilinear field, the length of the array is the sum of the dimensions of the field in computational space. For an irregular field, the length of the array is the product of the dimensions of the field in computational space and the number of dimensions in coordinate space. All the *X* coordinates are stored first, then all the *Y* coordinates, and so on. For an irregular field, the subscript for the first field dimension varies fastest. This argument is omitted for uniform fields.

AVSbuild_2d_field

C:
#include *<avs/field.h>*
AVSfield * AVSbuild_2d_field(*data, dim1, dim2***)**
float **data;*
int *dim1, dim2;*

FORTRAN:

AVSBUILD_2D_FIELD(*DATA, IDIM1, IDIM2***)**
REAL *DATA(IDIM1, IDIM2)*
INTEGER *IDIM1, IDIM2*

NOTE:
This routine is provided for backward compatibility with AVS2 only. New modules should use the **AVSdata_alloc** and **AVSfield_alloc** calls described in this section and in Chapter 2 under the heading "Creating Fields" instead. Modules that use the **AVSbuild...** series of calls cannot use shared memory. These routines may be removed from future releases of AVS.

This routine is a utility that builds a two-dimensional uniform scalar real field from its components. The routine returns a pointer to an **AVSfield** structure. The *data* argument is the data array, in "FORTRAN" order. The subscript for the first dimension varies fastest. The *dim1* and *dim2* arguments are integers specifying the size of the first and second dimensions, respectively.

AVSbuild_3d_field

C:
#include *<avs/field.h>*
AVSfield * AVSbuild_3d_field(*data, dim1, dim2, dim3***)**
float **data;*
int *dim1, dim2, dim3;*

FORTRAN:
AVSBUILD_3D_FIELD(*DATA, IDIM1, IDIM2, IDIM3***)**
REAL *DATA(IDIM1, IDIM2, IDIM3)*
INTEGER *IDIM1, IDIM2, IDIM3*

NOTE:
This routine is provided for backward compatibility with AVS 2 only. New modules should use the **AVSdata_alloc** and **AVSfield_alloc** calls described in this section and in Chapter 2 under the heading "Creating Fields" instead. Modules that use the **AVSbuild...** series of calls cannot use shared memory. These routines may be removed from future releases of AVS.

This routine is a utility that builds a three-dimensional uniform scalar real field from its components. The routine returns a pointer to an **AVSfield** structure. The *data* argument is the data array, in "FORTRAN" order. The subscript for the first dimension varies fastest, then the subscript for the second dimension. The *dim1*, *dim2*, and *dim3* arguments are integers specifying the size of the first, second, and third dimensions, respectively.

Field Accessor Routines

AVSfield_data_offset

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_DATA_OFFSET(*FIELD, BASEVEC, OFFSET***)**
  INTEGER *FIELD*
  <type> *BASEVEC(1)*
  INTEGER *OFFSET*

This routine allows the FORTRAN module writer to retrieve an offset index of the field data array relative to a given local reference array of <type>. The element *basevec*(*offset*+1) is the same as the first element of the data array. In order for FORTRAN to more conveniently handle this reference, pass this element to a second FORTRAN function which is expecting a variable size <type> array. The *basevec* array should actually be the same type as the field data array being retrieved (real to get real data, integer for integer data, etc.).

See the description of the **AVSPTR_ALLOC** routine in this appendix and the **AVSPTR_OFFSET** routine in Appendix F for information about related routines. Also, see the */usr/avs/examples/colorizer_f.f* example program.

AVSfield_data_ptr

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSfield * AVSfield_data_ptr(*field***)**
  AVSfield **field;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_DATA_PTR(*FIELD***)**
  INTEGER *FIELD*

This routine allows the module writer to retrieve the direct data pointer from the field structure and is intended primarily for the FORTRAN module writer. In order for a FORTRAN program to "dereference" the returned pointer, you should pass the the %VAL() (or %LOC() on some systems) of the pointer, along with the dimensions, to a second FORTRAN subroutine that declares the incoming argument as a variable size array.

WARNING:
This approach to getting the data array is not portable across all hardware platforms. Using the **AVSfield_data_offset** routine is a better approach.

Input

field
field to return data pointer for

Output

none

Returns

pointer
pointer to data array

AVSfield_get_dimensions

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
int AVSfield_get_dimensions(*field, dimensions***)**
AVSfield **field;*
int **dimensions;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSFIELD_GET_DIMENSIONS(*FIELD, DIMENSIONS***)**
INTEGER *FIELD*
INTEGER *DIMENSIONS(*ndim*)*

This routine allows the module writer to obtain the dimensions of the field's data space. It copies *field->ndims* elements into the *dimensions* array. It is up to the module writer to check that the array passed is at least large enough for the dimensions of the field.

Input

field
field to get dimensions array for

Output

dimensions
integer array to receive dimensions

Returns

1
valid data

0
invalid data

AVSfield_get_extent

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSfield_get_extent(*field, min_extent, max_extent***)**
AVSfield **float** **float**
**field;*
**min_extent;*
**max_extent;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_GET_EXTENT(*FIELD, MIN_EXTENT, MAX_EXTENT***)**
INTEGER *FIELD*
REAL *MIN_EXTENT*(nspace)
REAL *MAX_EXTENT*(nspace)

This routine allows the module writer to obtain the extent of the field in n-space. Please note that *min_extent* and *max_extent* are arrays of dimension *field->nspace* and must be allocated by the caller.

Input

field
field to get extents in

Outputs

min_extent
coordinates of minimum extent

max_extent
coordinates of maximum extent

Returns

1
valid data

0
invalid data

AVSfield_get_int

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSfield_get_int(*field, selector***)**

AVSfield **field;*
int *selector;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSFIELD_GET_INT(*FIELD, SELECTOR***)**
INTEGER *FIELD*
INTEGER *SELECTOR*

This routine allows the module writer to retrieve one of the integer fields in the field structure. The selector should be one of the following:

AVS_FIELD_NDIM
AVS_FIELD_NSPACE
AVS_FIELD_VECLEN
AVS_FIELD_TYPE
AVS_FIELD_SIZE
AVS_FIELD_UNIFORM
AVS_FIELD_FLAGS

Input

field
field to retrieve value from

selector
id of value to be retrieved

Output

none

Return

nonzero
value of integer field specified by *selector*

0
invalid selector specified

AVSfield_get_label

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
int AVSfield_get_label(*field, number, label***)**
AVSfield **field;*
int *number;*
char **label;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_GET_LABEL(*FIELD, NUMBER, LABEL***)**
INTEGER *FIELD*
INTEGER *NUMBER*
CHARACTER*length *LABEL*

This routine allows the module writer to query the label for an individual component in the field. It is up to the caller to make sure that the allocated array is large enough to hold the label string. The label string can have a maximum size of **AVS_FIELD_LABEL_LEN**.

Input

field
field to get label from

number
individual component number

Outputs

label
label string

Returns

1
valid data

0
invalid data

AVSfield_get_labels

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSfield_get_labels(*field, labels, delimiter***)**
AVSfield **field;*
char **labels;*
char **delimiter;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_GET_LABELS(*FIELD, LABELS, DELIMITER***)**
INTEGER *FIELD*
CHARACTER*length *LABELS*
CHARACTER*length *DELIMITER*

This routine allows the module writer to query the labels for each component in the field. For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to. It is up to the caller to make sure that the *labels* and *delimiter* arrays are long enough to contain the returned strings. These strings can be a maximum length of **AVS_FIELD_LABEL_LEN**.

Example

```
labels = "temp;density;mach number"
delimiter = ";"
```

Input

field
field to get labels in

Outputs

labels
string with labels and delimiters included

delimiter
delimiter between each individual string

Returns

1
valid data

0
invalid data

AVSfield_get_minmax

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSfield_get_minmax(*field, min, max***)**
AVSfield **field;*
char **min;*
char **max;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_GET_MINMAX(*FIELD, MIN, MAX***)**
INTEGER *FIELD*
<*type*> *MIN(*veclen*)*
<*type*> *MAX(*veclen*)*

This routine allows the module writer to obtain the range of the field data. Note that *min* and *max* are arrays of dimension *field->veclen* of the same type (BYTE, REAL, INTEGER, DOUBLE) as the computational data in the field. It is up to the caller to allocate enough space in these arrays to contain the returned information.

Input

field
field to get min / max from

Outputs

min
minimums of data

max
maximums of data

Returns

1
valid min / max

0
invalid min / max

AVSfield_get_unit

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
int AVSfield_get_unit(*field, number, unit***)**
AVSfield **field;*
int *number;*
char **unit;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSFIELD_GET_UNIT(*FIELD, NUMBER, UNIT***)**
INTEGER *FIELD*
INTEGER *NUMBER*
CHARACTER*length *UNIT*

This routine allows the module writer to query the unit string for an individual component in the field. It is up to the caller to allocate enough space in the *unit* array to contain the returned string. This string can be a maximum length of **AVS_FIELD_UNIT_LEN**.

Input

field
field to get units in

number
individual component number

Outputs

unit
unit string

Returns

1
valid data

0
invalid data

AVSfield_get_units

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
int AVSfield_get_units(*field, units, delimiter***)**
AVSfield **field;*
char **units;*
char **delimiter;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSFIELD_GET_UNITS(*FIELD, UNITS, DELIMITER***)**
INTEGER *FIELD*
CHARACTER*length *UNITS*
CHARACTER*length *DELIMITER*

This routine allows the module writer to query the units for each component in the field. The unit label is associated with each vector element in the array of computational data. It is a character array with a delimiter character as the first character in the array. The delimiter is followed by string / delimiter pairs, the number of which is equal to the vector length of the field. The unit labels are useful for defining measurement units for each variable in the array of data. For instance, in the case of a CFD dataset, the module writer might want to specify components of the field as temperature, density, mach number, etc.

It is up to the caller to allocate enough space in the *units* and *delimiter* arrays to contain the returned string. This string can be a maximum length of **AVS_-FIELD_UNIT_LEN**.

Example

```
units = "degrees C;g/cc;mach"
delimiter = ";"
```

Input

field
field to get units in

Outputs

units
string with units included

delimiter
delimiter between each individual string

Returns

1
valid data

0
invalid data

AVSfield_invalid_minmax

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
void AVSfield_invalid_minmax(*field***)**
AVSfield **field;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_INVALID_MINMAX(*FIELD***)**
INTEGER *FIELD*

This routine allows the module writer to set the min / max range of the field data to be invalid. This function should be used after the field data has been changed by the module and the module does not want to spend the time calling the routine **AVSfield_reset_minmax**.

Input

field
field to set min / max invalid

Outputs

none

Returns

none

AVSfield_points_offset

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSFIELD_POINTS_OFFSET(*FIELD, BASEVEC, OFFSET***)**
INTEGER *FIELD*
REAL *BASEVEC(*n*)*
INTEGER *OFFSET*

This routine allows the FORTRAN module writer to retrieve an offset index for the field's coordinates array relative to a given local reference array of type REAL. The element *BASEVEC*(*OFFSET*+1) is the same as the first element of the coordinates array in the field. In order for FORTRAN to more conveniently handle this reference, pass this element to a second FORTRAN function which declares its incoming argument as a variable size real array.

Returns

1
valid data

0
invalid data

AVSfield_points_ptr

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
int AVSfield_points_ptr(*field***)**
AVSfield **field;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSFIELD_POINTS_PTR(*FIELD***)**
INTEGER *FIELD*

This routine allows the module writer to retrieve the pointer to the coordinates array from the field structure and is intended primarily for the FORTRAN module writer. In order for the FORTRAN programmer to "dereference" the pointer, the %VAL() (%LOC() on some systems) of the pointer - along with the dimensions - should be passed to a second FORTRAN subroutine which declares its incoming argument as a variable size real array.

WARNING:
This approach to getting the points array is not portable across all hardware platforms. Using the **AVSfield_points_offset** routine is a better approach.

Input

field
field for which coordinates array pointer should be returned

Outputs

none

Returns

pointer
pointer to points array

AVSfield_reset_minmax

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
void AVSfield_reset_minmax(*field***)**
AVSfield **field;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_RESET_MINMAX(*FIELD***)**
INTEGER *FIELD*

This routine computes the min and max values for the field's computational data and stores them in the field's data structure.

Input

field
field to set min / max in

Outputs

none (use **AVSfield_get_minmax** to get the result)

Returns

none

AVSfield_set_extent

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
void AVSfield_set_extent(*field, min_extent, max_extent***)**
AVSfield **field;*
float **min_extent;*
float **max_extent;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_SET_EXTENT(*FIELD, MIN_EXTENT, MAX_EXTENT***)**
INTEGER *FIELD*
REAL *MIN_EXTENT(*nspace*)*
REAL *MAX_EXTENT(*nspace*)*

This routine allows the module writer to specify the extent of the field in n-space. It should be noted that *min_extent* and *max_extent* are arrays of dimension *field->nspace*.

Input

field
field to set extents in

min_extent
coordinates of minimum extent

max_extent
coordinates of maximum extent

Returns

none

AVSfield_set_int

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSfield_set_int(*field, selector, value***)**
AVSfield **field;*
int *selector;*
int *value;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSFIELD_GET_INT(*FIELD, SELECTOR, VALUE***)**

INTEGER *FIELD*
INTEGER *SELECTOR*
INTEGER *VALUE*

This routine allows the module writer to set one of the integer fields in the field structure. You can only perform this operation on a template. Otherwise, AVS issues an error. See also the **AVSfield_make_template** routine. See */usr/avs/examples/colorizer_f.f* for an example of a module that uses this call.

The following selectors determine which structure element to change:

AVS_FIELD_NDIM
AVS_FIELD_NSPACE
AVS_FIELD_VECLEN
AVS_FIELD_TYPE
AVS_FIELD_SIZE
AVS_FIELD_UNIFORM
AVS_FIELD_FLAGS

Input

field
field to in which to change value

selector
id of value to be changed

value
the new value

Output

none

Return

nonzero
value of integer field specified by *selector*

0
invalid selector specified

AVSfield_set_labels

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
void AVSfield_set_labels(*field, labels, delimiter***)**
AVSfield **field;*
char **labels;*

char **delimiter;*

FORTRAN:
#include *<avs/avs.inc>*
AVSFIELD_SET_LABELS(*FIELD, LABELS, DELIMITER***)**
INTEGER *FIELD*
CHARACTER*length *LABELS*
CHARACTER*length *DELIMITER*

This routine allows the module writer to set the labels for each component in the field. For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels appear on the dials so the user has a better understanding of which component attaches to each dial.

Example

```
labels = "temp;density;mach number"
delimiter = ";"
```

Input

field
field to set labels in

labels
string with labels included

delimiter
delimiter between each individual string

Outputs

none

Returns

none

AVSfield_set_minmax

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
void AVSfield_set_minmax(*field, min, max***)**
AVSfield **field;*
*<type> *min;*
*<type> *max;*

FORTRAN:
#include *<avs/avs.inc>*

AVSFIELD_SET_MINMAX(*FIELD, MIN, MAX***)**
INTEGER *FIELD*
<*type*> *MIN(*veclen*)*
<*type*> *MAX(*veclen*)*

This routine allows the module writer to set the range of the field data. It should be noted that *min* and *max* are arrays of dimension *field->veclen* of the same type (BYTE, REAL, INTEGER, DOUBLE) as the computational data in the field, although they are initially declared as byte (i.e., "char") arrays in **AVSfield_set_minmax** for generality.

Input

field
field to set min / max in

min
value of minimum data point

max
value of maximum data point

Outputs

none

Returns

none

AVSfield_set_units

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
void AVSfield_set_units(*field, units, delimiter***)**
AVSfield **field;*
char **units;*
char **delimiter;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_SET_UNITS(*FIELD, UNITS, DELIMITER***)**
INTEGER *FIELD*
CHARACTER*length *UNITS*
CHARACTER*length *DELIMITER*

This routine allows the module writer to set the unit for each component in the field. The unit label is associated with each vector element in the array of computational data. It is a character array with a delimiter character as the

first character in the array. The delimiter is followed by string / delimiter pairs, the number of which is equal to the vector length of the field. The unit labels are useful for defining measurement units for each variable in the array of data. For instance, in the case of a CFD dataset, the module writer might want to specify components of the field as temperature, density, mach number, etc.

Example

```
units = "degrees C;mg/cc;mach"
delimiter = ";"
```

Input

field
field to set units in

units
string with units included

delimiter
delimiter between each individual string

Outputs

none

Returns

none

AVSget_unique_id

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
int AVSget_unique_id()

FORTRAN:
#include <*avs/avs.inc*>
AVSGET_UNIQUE_ID()

The **AVSfield_alloc** call will assign mesh_ids to field structures when it is called. The number will be unique and different each time these calls are made during an AVS session. The allocation routines use the **AVSget_unique_id** function.

Modules (for example, filters) can signal downstream modules that the mesh has not changed by allocating their output data area, then copying the mesh_id from the input data area to the output data area. The downstream module can then check the mesh id against the previous number. If it is the same, then the mesh has not changed.

This *libflow* call is used with field structures to create unique mesh_ids. It returns an integer that is guaranteed to be unique throughout the AVS session.

Input

none

Outputs

none

Returns

unique ID for the field

AVSfield_set_mesh_id

C:
#include <*avs/avs.h*>
#include <*avs/field.h*>
void AVSfield_set_mesh_id(*field, mesh_id***)**
AVSfield **field;*
int *mesh_id;*

FORTRAN:
#include <*avs/avs.inc*>
AVSFIELD_SET_MESH_ID(*FIELD, MESH_ID***)**
INTEGER *FIELD*
INTEGER *MESH_ID*

This routine sets the mesh_id of the field structure. It can be used by modules to set the mesh_id to signal downstream modules that the input mesh (coordinates array) has been changed.

Input

field
field to set mesh_id in

mesh_id
number for new mesh_id

Outputs

none

Returns

none

AVSfield_get_mesh_id

C:
#include *<avs/avs.h>*
#include *<avs/field.h>*
void AVSfield_get_mesh_id(*field, mesh_id***)**
AVSfield **field;*
int **mesh_id;*

FORTRAN:
#include *<avs/avs.inc>*
AVSFIELD_GET_MESH_ID(*FIELD, MESH_ID***)**
INTEGER *FIELD*
INTEGER *MESH_ID*

This routine gets the mesh_id of the field structure. The mesh id can be used in field modules to determine if the input mesh (coordinates array) has changed since the last time the module received the structure from an up-stream module.

Input

field
field to get mesh_id from

Outputs

mesh_id
number for mesh_id

Returns

none

Colormap Accessor Routines

AVScolormap_get

C:
#include *<avs/avs.h>*
#include *<avs/colormap.h>*
int AVScolormap_get(*cmap, max_size, size, lower, upper, hue, saturation, value, alpha***)**
AVScolormap **cmap;*
int **size;*
int *max_size;*
float **lower, *upper, *hue, *saturation, *value, *alpha;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSCOLORMAP_GET(*CMAP, MAX_SIZE, SIZE,*
LOWER, UPPER, HUE, SATURATION,
*VALUE, ALPHA***)**
INTEGER *CMAP, MAX_SIZE, SIZE*
REAL *LOWER, UPPER*
REAL *HUE(n), SATURATION(n), VALUE(n), ALPHA(n)*

This routine must be used by FORTRAN module writers to access the contents of a colormap input or output when the SINGLE_ARG_DATA flag has been set using **AVSSET_MODULE_FLAGS**. For each colormap input or output, the module passes a single integer argument that is a pointer to a colormap. You can pass the pointer to this routine to get the contents of the colormap. The *HUE, SATURATION, VALUE, and ALPHA* data are copied into the arrays provided by the caller. It is up to the caller to ensure that these arrays are large enough to hold the returned information. AVS issues an error if the colormap size exceeds the max_size argument.

Input

cmap
pointer to a colormap

max_size
maximum size of the constituent arrays

Outputs

size
size of the constituent arrays

lower, upper
the range of the data values

hue, saturation, value, alpha
the colormap contents arrays (each of which is *size* elements each)

Returns

1
success

0
failure

AVScolormap_set

C:
#include *<avs/avs.h>*

#include *<avs/colormap.h>*
int AVScolormap_set(*cmap, size, lower, upper, hue, saturation, value, alpha***)**
AVScolormap **cmap;*
int **size;*
float **lower, *upper, *hue, *saturation, *value, *alpha;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSCOLORMAP_SET(*CMAP, SIZE, LOWER, UPPER, HUE, SATURATION, VALUE, ALPHA***)**
INTEGER *CMAP, SIZE*
REAL *LOWER, UPPER*
REAL *HUE(n), SATURATION(n), VALUE(n), ALPHA(n)*

This routine must be used by FORTRAN module writers to access the contents of a colormap input or output when the SINGLE_ARG_DATA flag has been set using **AVSSET_MODULE_FLAGS**. For each colormap input or output the module is passed a single integer argument that is a pointer to a colormap. You can pass the pointer to this routine to set the contents of the colormap. The four arrays for *HUE, SATURATION, VALUE, ALPHA* are copied from the provided arrays.

Input

cmap
pointer to a colormap

Outputs

size
size of the constituent arrays

lower, upper
the range of the data values

hue, saturation, value, alpha
the colormap contents

Returns

1
success

0
failure

User Data Accessor Routines

In C, programmers can directly access the elements in a user-defined structure by including the type file (see **AVSload_user_data_types**). In FORTRAN, when the **single_arg_data** flag is enabled, you must use the functions in this section to access the data in user-defined structures. See Chapter 4 for more information.

AVSudata_get_double

C:
#include <*avs/avs.h*>
#include <*avs/udata.h*>
int AVSudata_get_double(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
double **value;*
int *value_elements;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSUDATA_GET_DOUBLE(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
REAL*8 *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to retrieve a double precision value *name* from a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a double precision field, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure elements directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type. See */usr/avs/examples/user_data_f.f* for an example FORTRAN module.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Output

value
retrieved contents of the requested field

Returns

1
success

0
failure

AVSudata_get_int

C:
#include *<avs/avs.h>*
#include *<avs/udata.h>*
int AVSudata_get_int(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
int **value;*
int *value_elements;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSUDATA_GET_INT(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
INTEGER *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to retrieve an integer value named "name" from a user data type structure. For scalar values, pass 1 for the value_elements argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not an integer, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Output

value
retrieved contents of the requested field

Returns

1

success

0
failure

AVSudata_get_real

C:
#include <*avs/avs.h*>
#include <*avs/udata.h*>
int AVSudata_get_real(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
float **value;*
int *value_elements;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSUDATA_GET_REAL(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
REAL *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to retrieve a floating point value named "name" from a user data type structure. For scalar values, pass 1 for the value_elements argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not floating point, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Output

value
retrieved contents of the requested field

Returns

1
success

0
failure

AVSudata_get_string

C:
#include *<avs/avs.h>*
#include *<avs/udata.h>*
int AVSudata_get_string(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
char **value;*
int *value_elements;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSUDATA_GET_STRING(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
CHARACTER *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to retrieve a string value *name* from a user data type structure. For scalar values, pass 1 for the value_elements argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a string, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Output

value
retrieved contents of the requested field

Returns

1
success

0
failure

AVSudata_set_double

C:
#include <*avs/avs.h*>
#include <*avs/udata.h*>
int AVSudata_set_double(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
double **value;*
int *value_elements;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSUDATA_SET_DOUBLE(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER *NAME*(n)
REAL*8 *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to store a double precision value *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a double precision field, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_-data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Outputs

none

Returns

1
success

0
failure

AVSudata_set_int

C:
#include <*avs/avs.h*>
#include <*avs/udata.h*>
int AVSudata_set_int(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
int **value;*
int *value_elements;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSUDATA_SET_INT(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
INTEGER *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to store an integer value *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not an integer, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Outputs

none

Returns

1
success

0
failure

AVSudata_set_real

C:
#include *<avs/avs.h>*
#include *<avs/udata.h>*
int AVSudata_set_real(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
float **value;*
int *value_elements;*

FORTRAN:
#include *<avs/avs.inc>*
INTEGER AVSUDATA_SET_REAL(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
REAL *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to store a floating point value *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not floating point, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Outputs

none

Returns

1
success

0
failure

AVSudata_set_string

C:
#include <*avs/avs.h*>
#include <*avs/udata.h*>
int AVSudata_set_string(*ptr, name, value, value_elements***)**
char **ptr;*
char **name;*
char **value;*
int *value_elements;*

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSUDATA_SET_STRING(*PTR, NAME, VALUE, VALUE_ELEMENTS***)**
INTEGER *PTR, VALUE_ELEMENTS*
CHARACTER*(*) *NAME*
CHARACTER *VALUE*
DIMENSION *VALUE(VALUE_ELEMENTS)*

This routine allows the module writer to store a string value *name* into a user data type structure. For scalar values, pass 1 for the *value_elements* argument. For array values, pass the size of the array that you are providing. If there is no structure field with the given name or if the field is not a string, then the function returns an error. This function is intended primarily for FORTRAN module writers since C programmers can access the structure fields directly. You must call the function, **AVSload_user_data_types**, in the module description function to describe the user data type.

Inputs

ptr
pointer to a user written data structure

name
name of a field in the structure

value
variable that the value is to be copied into; from C, this must be a pointer to the variable

value_elements
number of array elements in the *value* argument being sent; should be 1 for scalar values

Outputs

none

Returns

1
success

0
failure

FORTRAN Array Accessor Routines

Since dynamic memory allocation is not a standard FORTRAN concept, relying on nonstandard extensions such as pointers makes modules less portable. The use of the SINGLE_ARG_DATA flag allows **AVSdata_alloc** to do most memory allocations automatically. When additional memory allocation services are required, there are two AVS interface functions that allow blocks of data passed into the compute routines to be referenced in a completely portable way. In particular, these functions avoid use of pointer variables and the POINTER statement. See */usr/avs/example/test_field_f.f* for an example using these techniques. These two calls are useful but they are largely superceeded by the use of **AVSfield_alloc**, etc.

AVSptr_alloc

FORTRAN:
INTEGER AVSPTR_ALLOC(*NAME, NELEM, ELSIZE, CLEAN, BASEVEC, ADDR, OFFSET***)**
INTEGER *NELEM, ELSIZE, CLEAN, OFFSET, ADDR*
DIMENSION *BASEVEC(1)*
CHARACTER*(*) *NAME*

AVSptr_alloc allocates a new data block for the given pointer. The parameters are as follows:

name
The name of the AVS output port that the pointer belongs to. Use a name consisting of a single SPACE character if the data block is not associated with an output port.

nelem
The number of array elements to allocate.

elsize
The element size in bytes (INTEGER*4 is 4, REAL*8 is 8, etc).

clean
If 1, the new elements are initialized to 0. Otherwise they are not initialized.

basevec
The start of the local array, used as a reference location. This array can be dimensioned with 1 element.

addr
The data block memory pointer. Initialize this to 0 if the data block is being used for a local array instead of for an output port.

offset
The offset index relative to the *basevec* array that corresponds to the first element of the data block pointed to by *addr*.

If *addr* points to an existing data block, that block is first freed and then a new block is allocated. If the requested memory cannot be allocated, a value of 0 is returned.

The sample program **colorizer_f.f** in */usr/avs/examples* provides an example of using this routine.

AVSptr_offset

FORTRAN:
INTEGER AVSPTR_OFFSET(*NAME, ELSIZE, BASEVEC, ADDR, OFFSET***)**
INTEGER *ELSIZE, OFFSET, ADDR*
DIMENSION *BASEVEC(1)*
CHARACTER*(*) *NAME*

AVSptr_offset gets the offset index for an existing data block without reallocating it. The parameters are the same as for **AVSptr_alloc**. If *addr* is 0 (no space allocation), a value of 0 is returned. Otherwise a value of 1 is returned.

Once the offset index is returned, there are several ways that it can be used, depending on the circumstances:

- For 1D arrays, add the offset value to all of the local reference array, as in *basevec(offset+i)*. The **read_image_f** example module in */usr/avs/examples* uses this approach.
- For multi-dimensional arrays, a statement function can be used to perform the index arithmetic. The **threshold_f** example module in */usr/avs/examples* uses this approach:

```
      integer function threshold(f, nx, ny, nz,
     *  gp, mx, my, mz, fmin, fmax)
      dimension f(nx, ny, nz)
      integer gp, goffset
      dimension g(1)
      real fmin, fmax

      gi(i,j,k) = goffset + j + (mx * ((j-1) + my * (k-1)))

      iresult = AVSptr_offset('output field', 4, g, gp, goffset)
      ...
      g(gi(i,j,k)) = 0.0
```

- A more convenient approach to handling arrays of any dimension is to pass the offset element of the local reference array to a second function that is expecting an array. This effectively "dereferences" the pointer and allows you to directly reference array elements. The **test_field_f** example module in */usr/avs/examples* uses this approach.

```
 integer function test_field_compute(pfield,ni,nj,nk,
*     coordflag,nspace,pcoords,
*     ires,spacing,gridtype)
 integer pfield, pcoords, ofield,ocoords
 integer coordflag,ires,iresult
 character*32 gridtype
 real field(1),coords(1)

 iresult = AVSptr_alloc('field', ires*ires*ires, 4, 0, field,
*     pfield, ofield)
 iresult = AVSptr_alloc('field', ires*ires*ires*3, 4, 0, coords,
*     pcoords, ocoords)
 test_field_compute=test_field_compute2(field(ofield+1),ni,nj,nk,
*     coordflag,nspace,coords(ocoords+1),ires,spacing,gridtype)
 return
 end

 integer function test_field_compute2(field,ni,nj,nk,
*     coordflag,nspace,coords,ires,spacing,gridtype)
 integer coordflag,ires
 character*32 gridtype
 real field(ires,ires,ires),coords(ires,ires,ires,3)
 ...
          field(i,j,k)=dist/dmax
```

FORTRAN Single Byte Accessor Routines

See */usr/avs/colorizer_f.f* for an example of a module that uses these calls.

AVSload_byte

FORTRAN:
#include <*avs/avs.inc*>
INTEGER AVSLOAD_BYTE(*BASE, OFFSET***)**
INTEGER *BASE, OFFSET*

This function loads a byte from memory. This is useful for FORTRAN's that do not have a BYTE data type and do not allow LOGICAL*1 to be used as a numeric value.

Inputs

BASE
base address

OFFSET
byte offset from the base address (the first byte is number 1)

Outputs

none

Returns

value of an unsigned byte

AVSstore_byte

FORTRAN:
#include <*avs/avs.inc*>
AVSSTORE_BYTE(*BASE, OFFSET, VALUE***)**
INTEGER *BASE, OFFSET, VALUE*

This subroutine stores a byte into memory. This is useful for FORTRAN's that do not have a BYTE data type and do not allow LOGICAL*1 to be used as a numeric value.

Inputs

BASE
base address

OFFSET
byte offset from the base address (the first byte is number 1)

VALUE
new value for byte (only the low order 8 bits are used)

Outputs

none

Returns

none

Routines for Handling Errors

These routines provide the module with access to dialog boxes that they can use to warn users or provide choices to users. See the **AVSmessage** routine for a description of the severity levels that you can assign. See */usr/avs/examples/widgets.c* and *usr/avs/examples/widgets_f.f* for examples of modules that use these calls.

AVSdebug

C:
AVSdebug(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
char **message_format;*
char **msg1, *msg2, *msg3, *msg4, *msg5, *msg6;*

FORTRAN:
AVSDEBUG(*MESSAGE***)**
CHARACTER*length *MESSAGE*

This routine is an interface to the **AVSmessage** routine. It presents a message of severity **AVS_Debug**.

C language: To produce the message to be presented to the user, AVS calls **sprintf**(3S) with *message_format* as the format string and *msg1* through *msg6* as the arguments. The *msg1* through *msg6* arguments can be of any type valid for **sprintf**. Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented to the user is the *message* argument.

This routine presents the user with only the default choice, "Ok". It returns no meaningful value.

AVSerror

C:
AVSerror(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
char **message_format;*
char **msg1, *msg2, *msg3, *msg4, *msg5, *msg6;*

FORTRAN:
AVSERROR(*MESSAGE***)**
CHARACTER*length *MESSAGE*

This routine is an interface to the **AVSmessage** routine. It presents a message of severity **AVS_Error**.

C language: To produce the message to be presented to the user, AVS calls **sprintf**(3S) with *message_format* as the format string and *msg1* through *msg6* as the arguments. The *msg1* through *msg6* arguments can be of any type valid for **sprintf**. Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented to the user is the *message* argument.

This routine presents the user with only the default choice, "Ok". It returns no meaningful value.

AVSfatal

C:
AVSfatal(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
char **message_format;*
char **msg1, *msg2, *msg3, *msg4, *msg5, *msg6;*

FORTRAN:
AVSFATAL(*MESSAGE***)**
CHARACTER*length *MESSAGE*

This routine is an interface to the **AVSmessage** routine. It presents a message of severity **AVS_Fatal**.

C language: To produce the message to be presented to the user, AVS calls **sprintf**(3S) with *message_format* as the format string and *msg1* through *msg6* as the arguments. The *msg1* through *msg6* arguments can be of any type valid for **sprintf**. Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented to the user is the *message* argument.

This routine presents the user with only the default choice, "Ok". It returns no meaningful value.

AVSinformation

C:
AVSinformation(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
char **message_format;*
char **msg1, *msg2, *msg3, *msg4, *msg5, *msg6;*

FORTRAN:
AVSINFORMATION(*MESSAGE***)**
CHARACTER*length *MESSAGE*

This routine is an interface to the **AVSmessage** routine. It presents a message of severity **AVS_Information**.

C language: To produce the message to be presented to the user, AVS calls **sprintf**(3S) with *message_format* as the format string and *msg1* through *msg6* as the arguments. The *msg1* through *msg6* arguments can be of any type valid for **sprintf**. Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented to the user is the *message* argument.

This routine presents the user with no choices and returns no meaningful value.

AVSmessage

C:
#include *<avs/avs.h>*
char * AVSmessage(*version, severity, module, function_name, choices, message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
char **version;*
AVS_MESSAGE_SEVERITY *severity;*
integer *module;*
char **function_name, *choices, *message_format;*
char **msg1, *msg2, *msg3, *msg4, *msg5, *msg6;*

FORTRAN:
#include *<avs/avs.inc>*
AVSMESSAGE(*VERSION, SEVERITY, MODULE, FUNCTION_NAME, CHOICES, MESSAGE***)**
CHARACTER*length *VERSION*
INTEGER *SEVERITY(*length*)*
INTEGER *MODULE*
CHARACTER*length *FUNCTION_NAME*
CHARACTER*length *CHOICES, MESSAGE*

This routine causes AVS to present the user with a message from a module computation routine, along with information about the module and function

sending the message. If the sender indicates that the message represents a warning or error, AVS also stops executing and presents the message in a dialog box, along with a set of choices. The user must acknowledge the message by selecting one of the choices before AVS can continue. The icon for the module that sends the message is highlighted in yellow in the Network Editor. The **AVSmessage** routine also records the message in a log file (*/tmp/avslog.<process ID>*) for later review.

Following is a description of the arguments:

version

A string indicating what version of the module is reporting the error. This can be any string, but it should be a meaningful identification for the code developer.

In some source code management systems, updating the version string can be handled automatically. In SCCS, for example, you can insert a line into a C source file declaring a global string variable that matches SCCS id keywords. The string is updated each time a delta is made. For example:

```
static char file_version[] = "%W% %E%";
```

severity

A value indicating the relative importance of the message being sent. This determines how AVS presents the message to the user and whether or not the user must acknowledge the message before AVS can continue. If the message appears in a dialog box, the border of the dialog box is color coded to indicate the severity. Following are the possible values:

AVS_Information

The message does not indicate an error. The message is written to *stderr*, and AVS continues executing. No choices are presented to the user.

AVS_Debug

The message does not indicate an error; it conveys information during module testing. The message is written to *stderr*, and AVS continues executing. No choices are presented to the user.

AVS_Warning

The message indicates a problem that is not fatal to module execution. The message and choices are presented in a dialog box with a yellow border. The user must make a choice before AVS can continue.

AVS_Error

The message indicates a serious problem that is not fatal to module execution. The message and choices are presented in a dialog box with a red border. The user must make a choice before AVS can continue.

AVS_Fatal

The message indicates a problem that is fatal to module execution. The message and choices are presented in a dialog box with a black border. The user must make a choice before AVS can continue. The

module is marked as dead, and the module icon in the Network Editor workspace turns black. The flow executive no longer executes the module.

module
The module sending the message. This value should always be NULL in C (0 in FORTRAN). AVS automatically identifies the module sending a message and highlights its icon in yellow.

function
The name of the function sending the message.

choices
A string containing the names of options to be presented to the user. The choices are separated by exclamation points (!). For example, "Ok!Kill Module!Exit" is presented as three choices: "Ok", "Kill Module", and "Exit". If the value is NULL in C (0 in FORTRAN) or the empty string, AVS presents a default choice, "Ok". AVS can add choices to those specified in the *choices* argument.

message_format, msg1, msg2, msg3, msg4, msg5, msg6
C language: To produce the message to be presented to the user, AVS calls **sprintf**(3S) with *message_format* as the format string and *msg1* through *msg6* as the arguments. The *msg1* through *msg6* arguments can be of any type valid for **sprintf**. Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented to the user is the *message* argument.

AVSmessage returns a string containing the choice the user made. A C language routine can use **strcmp**(3C) to identify the choice, as in this example:

```
char *answer;

answer = AVSmessage(...,"Ok!Reset!Exit", ...)
if (!strcmp(answer,"Reset")) { /* reset action */ }
else if (!strcmp(answer,"Exit") { exit(1); }
```

A FORTRAN routine should declare **AVSMESSAGE** to return **CHARACTER*** *n*, where *n* is the maximum length of the string to be returned. The string is padded on the right with spaces. The routine can use the **.EQ.** operator to identify the choice, as in this example:

```
      ...
      EXTERNAL AVSMESSAGE
      CHARACTER*32 AVSMESSAGE
      CHARACTER*32 RESPONSE
      RESPONSE = AVSMESSAGE('Version 1', AVS_Error, 0,
     +     'MY_ROUTINE', 'Ok!Reset!Exit',
     +     'Attempt to divide by zero.')
      IF (RESPONSE(1:2) .EQ. 'Ok') THEN
C     Process 'Ok' choice
      ELSE IF (RESPONSE(1:5) .EQ. 'Reset') THEN
C     Process 'Reset' choice
      ELSE IF (RESPONSE(1:4) .EQ. 'Exit') THEN
```

```
C     Process 'Exit' choice
      ELSE
C     Process other choices added by AVS
      END IF
      ...
```

Because AVS can add choices to those supplied in the *choices* argument, the returned value might not be one of the substrings in *choices*. For messages of severity **AVS_Information** and **AVS_Debug**, no choices are presented to the user, and the returned value is the empty string.

All messages sent through the AVS message mechanism are written to a log file named */tmp/avslog.<process ID>* in the current working directory. The log file may contain additional information beyond that presented in the dialog box, including the *version* string.

AVSmessage_sub

FORTRAN:
#include *<avs/avs.inc>*
AVSMESSAGE_SUB(*ANSWER, VERSION, SEVERITY, MODULE, FUNCTION_NAME, CHOICES, MESSAGE***)**
CHARACTER*length *ANSWER*
CHARACTER*length *VERSION*
INTEGER *SEVERITY*
INTEGER *MODULE*
CHARACTER*length *FUNCTION_NAME*
CHARACTER*length *CHOICES, MESSAGE*

This subroutine is a preferred alternative to **AVSMESSAGE** for FORTRAN module writers which modifies the *answer* argument rather than returning it as a function result. This approach is more portable between AVS implementations on different hardware platforms.

AVSwarning

C:
AVSwarning(*message_format, msg1, msg2, msg3, msg4, msg5, msg6***)**
char **message_format;*
char **msg1, *msg2, *msg3, *msg4, *msg5, *msg6;*

FORTRAN:
AVSWARNING(*MESSAGE***)**
CHARACTER*length *MESSAGE*

This routine is an interface to the **AVSmessage** routine. It presents a message of severity **AVS_Warning**.

C language: To produce the message to be presented to the user, AVS calls **sprintf**(3S) with *message_format* as the format string and *msg1* through *msg6* as the arguments. The *msg1* through *msg6* arguments can be of any type valid for **sprintf**. Only as many arguments as the format string requires need be supplied.

FORTRAN: The message to be presented to the user is the *message* argument.

This routine presents the user with only the default choice, "Ok". It returns no meaningful value.

APPENDIX B

AVS C LANGUAGE FIELD MACROS

Macros for Obtaining the Dimensions of a Field

MAXX

#include <*avs/field.h*>
MAXX(*field***)**
  AVSfield **field;*

MAXX provides the size of the first dimension of a field.

MAXY

#include <*avs/field.h*>
MAXY(*field***)**
  AVSfield **field;*

MAXY provides the size of the second dimension of a field.

MAXZ

#include <*avs/field.h*>
MAXZ(*field***)**
  AVSfield **field;*

MAXZ provides the size of the third dimension of a field.

Macros for Obtaining Elements of a Scalar Data Array

I2D

#include <*avs/field.h*>
I2D(*field, i, j***)**

AVSfield **field;*
int *i, j;*

For a two-dimensional field, **I2D** provides the element of the data array that corresponds to index *i* of the first dimension and index *j* of the second dimension. Note that the index "arguments" are in order of the field dimensions; if the indices were used directly as subscripts into the data array, they would be in reverse order.

I3D

#include <*avs/field.h*>
I3D(*field, i, j, k***)**
AVSfield **field;*
int *i, j, k;*

For a three-dimensional field, **I3D** provides the element of the data array that corresponds to index *i* of the first dimension, index *j* of the second dimension, and index *k* of the third dimension. Note that the index "arguments" are in order of the field dimensions; if the indices were used directly as subscripts into the data array, they would be in reverse order.

I4D

#include <*avs/field.h*>
I4D(*field, i, j, k, l***)**
AVSfield **field;*
int *i, j, k, l;*

For a four-dimensional field, **I4D** provides the element of the data array that corresponds to index *i* of the first dimension, index *j* of the second dimension, index *k* of the third dimension, and index *l* of the fourth dimension. Note that the index "arguments" are in order of the field dimensions; if the indices were used directly as subscripts into the data array, they would be in reverse order.

Macros for Obtaining Elements of a Vector Data Array

I1DV

#include <*avs/field.h*>
I1DV(*field, i***)**
AVSfield **field;*
int *i;*

For a one-dimensional field, **I1DV** provides a pointer to the first element of the vector in the data array that corresponds to index *i*.

I2DV

#include <*avs/field.h*>
I2DV(*field, i, j***)**
AVSfield **field;*
int *i, j;*

For a two-dimensional field, **I2DV** provides a pointer to the first element of the vector in the data array that corresponds to index *i* of the first dimension and index *j* of the second dimension. Note that the index "arguments" are in order of the field dimensions; if the indices were used directly as subscripts into the data array, they would be in reverse order, with the vector index as the last subscript.

I3DV

#include <*avs/field.h*>
I3DV(*field, i, j, k***)**
AVSfield **field;*
int *i, j, k;*

For a three-dimensional field, **I3DV** provides a pointer to the first element of the vector in the data array that corresponds to index *i* of the first dimension, index *j* of the second dimension, and index *k* of the third dimension. Note that the index "arguments" are in order of the field dimensions; if the indices were used directly as subscripts into the data array, they would be in reverse order, with the vector index as the last subscript.

I4DV

#include <*avs/field.h*>
I4DV(*field, i, j, k, l***)**
AVSfield **field;*
int *i, j, k, l;*

For a four-dimensional field, **I4DV** provides a pointer to the first element of the vector in the data array that corresponds to index *i* of the first dimension, index *j* of the second dimension, index *k* of the third dimension, and index *l* of the fourth dimension. Note that the index "arguments" are in order of the field dimensions; if the indices were used directly as subscripts into the data array, they would be in reverse order, with the vector index as the last subscript.

Macros for Obtaining Rectilinear Coordinate Arrays

RECT_X

```
#include <avs/field.h>
RECT_X(field)
    AVSfield    *field;
```

For a rectilinear field, **RECT_X** provides a pointer to the first element of the coordinate array that corresponds to the first dimension of computational space.

RECT_Y

```
#include <avs/field.h>
RECT_Y(field)
    AVSfield    *field;
```

For a rectilinear field, **RECT_Y** provides a pointer to the first element of the coordinate array that corresponds to the second dimension of computational space.

RECT_Z

```
#include <avs/field.h>
RECT_Z(field)
    AVSfield    *field;
```

For a rectilinear field, **RECT_Z** provides a pointer to the first element of the coordinate array that corresponds to the third dimension of computational space.

Macros for Obtaining Coordinates for 3D Data Elements

COORD_X_3D

```
#include <avs/field.h>
COORD_X_3D(field, i, j, k)
    AVSfield    *field;
    int     i, j, k;
```

For a three-dimensional uniform field, **COORD_X_3D** "returns" *i*. For a three-dimensional rectilinear or irregular field, **COORD_X_3D** provides the *X* co-

ordinate from the coordinate array that corresponds to the data element specified by the indices *i*, *j*, and *k*.

COORD_Y_3D

#include <*avs/field.h*>
COORD_Y_3D(*field, i, j, k***)**
AVSfield **field;*
int *i, j, k;*

For a three-dimensional uniform field, **COORD_Y_3D** "returns" *j*. For a three-dimensional rectilinear or irregular field, **COORD_Y_3D** provides the *Y* coordinate from the coordinate array that corresponds to the data element specified by the indices *i*, *j*, and *k*.

COORD_Z_3D

#include <*avs/field.h*>
COORD_Z_3D(*field, i, j, k***)**
AVSfield **field;*
int *i, j, k;*

For a three-dimensional uniform field, **COORD_Z_3D** "returns" *k*. For a three-dimensional rectilinear or irregular field, **COORD_Z_3D** provides the *Z* coordinate from the coordinate array that corresponds to the data element specified by the indices *i*, *j*, and *k*.

APPENDIX C

EXAMPLES OF AVS MODULES

Introduction

This appendix contains example source code for three AVS modules:

- A C language subroutine module that computes the threshold of a field of floating-point numbers.
- A FORTRAN version of the first example.
- A C language coroutine module that creates a geometry object.

For files that contain source code for these and other examples, see the directory */usr/avs/examples*.

AVS Example Modules

The following list describes the examples located in the */usr/avs/examples* directory:

Makefile
The examples Makefile is a good template for module makefiles for both FORTRAN and C modules. It sets up library and include file references so they can be redirected and it picks up definitions from the AVS Makeinclude file for the local hardware platform. For FORTRAN, it sets up a link to the include directory to allow the use of the FORTRAN include statement in FORTRAN modules.

avs_client.c
An example of an external process that can attach to AVS using the -server option and send it CLI commands to drive it remotely or for animations.

menus.c
This example shows how to control panels of widgets using a module. It centers on two techniques: setting up the panels using the **AVSadd_parameter_prop** function and controlling the panel visibility using the **AVScommand** function in the compute routine of the module.

camera.c
This example demonstrates the use of the routine: GEOMedit_projection to define the camera projection. This module allows you to specify a camera position.

pick_cube.c
This example demonstrates how to use upstream data from the render geometry module to pick geometric objects. Functionally, this module generates a "cube". The user selects a face of the cube and the module regenerates the cube with the selected face highlighted in red. It places a green sphere over the closest vertex to the selection, and a blue sphere over the selected point.

polygon.c
This example creates a geometric object. In this example, the original data is kept in an ascii description file. This module converts disjoint polygon information into geom format. It assumes that the polygons have no normals or colors (but you can easily modified it to include either or both). The vertices of the polygons can either be shared by all of the polygons (in which case they are be smooth shaded), or unshared (in which case they are flat shaded).

qix.c
This example draws disjoint lines in a random pattern. Instead of providing a compute module function that AVS calls whenver a parameter or input changes, this module determines when it wants to provide new data to the network. Many existing applications fit into this model more easily than the "compute function" model.

read_image.c
This example reads an image from a file ".x" format.

read_plot3d.c
This example reads a plot3d format file (this is the source for the unsupported module "Read Plot3D").

read_scans.c
This example reads a 3D scalar field that is organized as a set of different files.

read_ucd.c
This example reads a file (ascii or binary) that is in the ucd format. The example is slightly out of date, since it reads AVS 3 binary format ucd files, not the newer AVS 4 format. (The ASCII file format is unchanged.)

threshold.c
This example computes the threshold of a 3D scalar field of floating point numbers. The threshold function examines each element of a field to see whether it falls within the range specified by the minimum and maximum parameters (controlled by dials). Elements in the range are passed

unchanged to the output field, elements outside the range are set to zero in the output field.

This example module is not as general as the standard **threshold** module. It does not handle rectilinear and irregular data.

ucd_extract.c

This example extracts a single data component of a UCD structure. It creates an output UCD structure that contains all the cells and nodes of the input UCD sructure, but with only one scalar or vector data component. It illustrates both creating UCD structures and accessing the elements of existing structures.

ucd_thresh.c

This example computes the threshold of a UCD structure. It examines each cell of a UCD structure to see whether it falls within the range specified by minimum and maximum parameters supplied by the **ucd legend** module. It illustrates both creating UCD structures and accessing the elements of existing structures.

user_data.c

This example demonstrates using the user_data data type defined in the include file: ex_user_data.h.

widgets.c

This examples demonstrates the use of widgets. If you have questions about how to do something with widgets / parameters, you may find an example of it in this file.

colorizer_f.f

This example to takes a scalar volume and computes a colorized volume. In this example, the init (AVSint_modules) function calls two init routines to create two different modules. This is useful when modules share a substantial amount of code or when there is an advantage to letting them run in the same process (reduced data passing, etc).

gen_ucd.f

This example uses the ucd function calls. It creates a block of hexahedra using either scalar or vector data.

polygon_f.f

This example is a FORTRAN version of the polygon.c example discussed in the C Example Modules section.

qix_f.f

This example is a FORTRAN version of the qix.c example discussed in the C Example Modules section.

read_image_f.f

This example is a FORTRAN version of the read_image.c example discussed in the C Example Modules section.

read_vol_f.f
This example is a FORTRAN version of the read_vol.c example discussed in the C Example Modules section.

test_field_f.f
This example generates a dummy 3D field. You can specify irregular, rectilinear or uniform and it outputs a field of that type. The computational space is a layered sphere, ranging from 0.0 to 1.0. If you select either rectilinear or irregular field types, it is possible to adjust the spacing between grid elements, based on an exponential function. This is a good module for learning about what the parameters these data types can use. You can use this module with a volume bounds object to understand the field it producs.

test_fld2_f.f
This example is a modified version of test_field_f.f that uses the single argument FORTRAN field passing convention.

threshold_f.f
This example is a FORTRAN version of the threshold.c example discussed in the C Example Modules section.

user_data_f.f
Example of using the user_data data type defined in ex_user_data.h.

widgets_f.f
This example is a FORTRAN version of the widgets.c example discussed in the C Example Modules section.

chemistry
A subdirectory that contains examples of modules that manipulate the molecule data type.

imagenode
(Only with the Animator product.) A subdirectory containing sample source to do output from AVS to a VCR to perform single frame recording.

A C Language Subroutine Module

```
#include <avs/avs.h>
#include <avs/field.h>

/*****************************************************************************/

/*
 * This is a C example to compute the threshold of a 3D scalar field of
 * floating point numbers.
 */

/*
```

```
 * The threshold function examines each element of a field to see
 * whether it falls within the range specified by a minimum and maximum
 * parameter (controlled by dials).  Elements in the range are passed
 * unchanged to the output field, elements outside the range are
 * set to zero in the output field.
 */

/*
 * The function AVSinit_modules is called from the main() routine supplied
 * by AVS.  In it, we call AVSmodule_from_desc with the name of our
 * description routine.
 */

AVSinit_modules()
{
     void threshold();

     AVSmodule_from_desc(threshold);
}

/*  The routine "threshold" is the description routine.  */

threshold()
{
     int thresh_compute();     /* declare the compute function (below) */
     int in_port, out_port;     /* temporaries to hold the port numbers */

     /* Set the module name and type */
     AVSset_module_name("ex1-threshold", MODULE_FILTER);

     /* Create an input port for the required field input */
     in_port =
          AVScreate_input_port("Input Field",
                         "field 3D uniform scalar float", REQUIRED);

     /* Create an output port for the result */
     out_port = AVScreate_output_port("Output Field",
                          "field 3D uniform scalar float");

     /* Tell AVS to allocate space for the output data based on the size */
     /* of the input data - note that this only works when the output */
     /* port has the same type as the input port */

     AVSinitialize_output(in_port, out_port);

     /* Add two floating point parameters, both unbounded.  Min has */
     /* an initial value of zero, max of 255 */

     AVSadd_float_parameter("thresh_min", 0.0, FLOAT_UNBOUND, FLOAT_UNBOUND);
     AVSadd_float_parameter("thresh_max", 255.0, FLOAT_UNBOUND, FLOAT_UNBOUND);

     /* Tell avs what subroutine to call to do the compute */
     AVSset_compute_proc(thresh_compute);
}

/*
```

```
 * thresh_compute is the compute routine.  It is called whenever AVS wants to
 * compute new threshold results.  The arguments are: the value of the input
 * field, the new output field (doubly indirected), the minimum parameter
 * value and the maximum parameter value.  Note the order is always inputs,
 * outputs, parameters.  The min comes before the max because in the
 * description routine above, the min is declared before the max.
 */

thresh_compute(input, output, pmin, pmax)
AVSfield_float *input, **output;
float *pmin, *pmax;
{
    register int i, j, k;
    register float min = *pmin;
    register float max = *pmax;

    /*
     * We use a triply nested loop to traverse the field.  The macros MAXX,
     * MAXY, and MAXZ determine the maximum extent of the field in each of
     * the three dimensions.  We know this will be a 3-dimensional
     * field because of the declaration in the description routine, so
     * we don't need to check.  When we want to reference an element of the
     * field we use the I3D macro which picks an element of a 3D field.
     * Note that the first index (i) varies the fastest in memory, so we
     * make that the innermost loop.
     */

     for (k = 0; k < MAXZ(input); k++)
          for (j = 0; j < MAXY(input); j++)
               for (i = 0; i < MAXX(input); i++)
                   if (I3D(input, i, j, k) > max) {
                        I3D(*output, i, j, k) = 0.0;
                   } else if (I3D(input, i, j, k) < min) {
                        I3D(*output, i, j, k) = 0.0;
                   } else {
                        I3D(*output, i, j, k) = I3D(input, i, j, k);
                   }

    /* When we're done, we return 1 to indicate success */
    return(1);
}
```

A FORTRAN Subroutine Module

```
C This is a FORTRAN example to compute the threshold of a 3D scalar
C field of floating point numbers.



C The threshold function examines each element of a field to see
C whether it falls within the range specified by a minimum and maximum
C parameter (controlled by dials).  Elements in the range are passed
C unchanged to the output field, elements outside the range are
C set to zero in the output field.
```

```
C The AVS startup routines will call AVSinit_modules to initialize the
C modules.  This is the description routine for the module.

      subroutine AVSinit_modules
      include 'avs/avs.inc'
      integer iport, oport, iparm
      external threshold

C Set the module name and type
      call AVSset_module_name('threshold f', 'filter')

C Create an input port for the required field input

      iport = AVScreate_input_port('input field',
     $     'field 3D scalar uniform float', REQUIRED)

C Create an output port for the result

      oport = AVScreate_output_port('output field',
     $     'field 3D scalar uniform float')

C Tell AVS to allocate space for the output data based on the size
C of the input data - note that this only works when the output
C port has the same type as the input port

      call AVSinitialize_output(iport, oport)

C Add two floating point parameters, both unbounded.  Min has
C an initial value of zero, max of 255

      iparm = AVSadd_parameter('min', 'float', 0.0,
     $     FLOAT_UNBOUND, FLOAT_UNBOUND)
      iparm = AVSadd_parameter('max', 'float', 255.0,
     $     FLOAT_UNBOUND, FLOAT_UNBOUND)

C Tell AVS what function to call to do the compute

      call AVSset_compute_proc(threshold)

      return
      end


C Threshold is the compute function.  The first four arguments
C represent the input field: f, nx, ny, nz.  The second four arguments
C represent the output field: gp, mx, my, mz.  Since we used
C AVSinitialize_output in the description routine, gp, mx, my, and mz
C will already have the appropriate values.
C
C Note that for output field, we set up the an integer to receive a
C POINTER to the data field. We declare a local array g of size 1 and
C will use it as a base array in conjunction with an offset (goffset)
C that is obtained by calling AVSptr_offset to determine the distance
C between the base array G and the actual output data array pointed to
```

```
C by gp.
C
C The last two arguments are the minimum and the maximum, read from
C dials manipulated by the user.  Note that they are presented to the
C subroutine in the order they are declared in the description routine.

      integer function threshold(f, nx, ny, nz,
     $     gp, mx, my, mz, fmin, fmax)
      dimension f(nx, ny, nz)
      integer gp, goffset,gi
      real fmin, fmax, g
      dimension g(1)

C
C One option is to then define a statement function that will handle
C the index calculations, to simplify making references into the base
C array local array
C
      gi(i,j,k) = goffset + i + (mx * ((j-1) + my * (k-1)))

C call AVSPTR_OFFSET( NAME, ELSIZE, BASEVEC, ADDR, OFFSET )
C where:
C
C NAME    - the name of the port / parameter the data is associated
C           with
C ELSIZE  - the size of the elements = 4 bytes in our case
C BASEVEC - the an array name to use for indexing = G in our case
C ADDR    - the address of the actual data storage = GP
C OFFSET  - the offset to use in indexing = GOFFSET
C
      i = AVSptr_offset('output field', 4, g, gp, goffset)

      do k = 1, nz
         do j = 1, ny
            do i = 1, nx
               if (f(i, j, k) .gt. fmax) then

C For each reference to array G use the index calculation function
C GI(i,j,k)

                  g(gi(i,j,k)) = 0.0
               elseif (f(i, j, k) .lt. fmin) then
                  g(gi(i,j,k)) = 0.0
               else
                  g(gi(i,j,k)) = f(i, j, k)
               endif
            enddo
         enddo
      enddo

C When we're done, we return 1 to indicate success

      threshold = 1
      return
      end
```

A C Language Coroutine Module

```
#include <stdio.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/geom.h>

/*****************************************************************************/

/*
 * This is a C example to create a geometry object.  In this example,
 * the "simulation" program flow of control is used.  Instead of providing
 * a compute module function that is called whenever a parameter or input
 * has changed, this module can determine when it wants to provide new
 * data to the network.  Many existing applications will fit into this
 * model much more easily than the "compute function" model.
 */

/*
 * The routine "qix" is the description routine.  It provides
 * AVS some necessary information such as: name, input and output ports,
 * parameters etc.
 */
qix()
{
     int out_port;          /* temporary to hold the port number */
     int parm;          /* temporary to hold the parm number */

     /* Set the module name and type */
     AVSset_module_name("qix", MODULE_DATA);

     /* There are no input ports for this module */

     /* Create an output port for the result */
     out_port = AVScreate_output_port("Output Geometry", "geom");

     /* Add one parameter: an enable/disable toggle for the scope */
     (void) AVSadd_parameter("sleep", "boolean", 1, 0, 1);

     /* There is no compute function for this module */
}

#define MAXV 200
#define PERFRAME 6

typedef float FLOAT3[3];

main(argc,argv)
int     argc;
char     *argv[];
{
   int qix();
   int count = MAXV;
   FLOAT3 verts[2], move0, move1, colors[2], movec0, movec1;
   int sleep = 1;
```

```
    GEOMobj *obj = NULL;
    GEOMedit_list output = NULL;
    int i;

    AVScorout_init(argc,argv,qix);

    while(1) {
       /* If we are told to sleep, we'll just wait until a parameter changes */
       if (sleep) AVScorout_wait();

       /* Get input parameter (any inputs would be here as well) */
       AVScorout_input(&sleep);

       for (i = 0; i < PERFRAME; i++) {
       if (count >= MAXV) {
          start(verts,colors,move0,move1,movec0,movec1);
          count = 0;
          if (obj) GEOMdestroy_obj(obj);
          obj = GEOMcreate_obj(GEOM_POLYTRI,NULL);
       }
       else next(verts,colors,move0,move1,movec0,movec1);
       GEOMadd_disjoint_line(obj,verts,colors,2,GEOM_COPY_DATA);
       count++;
       }

       output = GEOMinit_edit_list(output);
       GEOMedit_geometry(output,"qix",obj);
       AVScorout_output(output);
    }
}

#define RA 5.0
#define DD 0.2
#define DC 0.05

start(verts,colors,move0,move1,movec0,movec1)
FLOAT3 *verts;
FLOAT3 *colors;
FLOAT3 move0, move1;
FLOAT3 movec0, movec1;
{
    float ran();

    verts[0][0] = ran(RA); verts[0][1] = ran(RA); verts[0][2] = ran(RA);
    verts[1][0] = ran(RA); verts[1][1] = ran(RA); verts[1][2] = ran(RA);

    move0[0] = ran(DD); move0[1] = ran(DD); move0[2] = ran(DD);
    move1[0] = ran(DD); move1[1] = ran(DD); move1[2] = ran(DD);

    colors[0][0] = ran(1.0); colors[0][1] = ran(1.0); colors[0][2] = ran(1.0);
    colors[1][0] = ran(1.0); colors[1][1] = ran(1.0); colors[1][2] = ran(1.0);

    movec0[0] = ran(DC); movec0[1] = ran(DC); movec0[2] = ran(DC);
    movec1[0] = ran(DC); movec1[1] = ran(DC); movec1[2] = ran(DC);
}
```

```
next(verts,colors,move0,move1,movec0,movec1)
FLOAT3 *verts;
FLOAT3 *colors;
FLOAT3 move0, move1;
FLOAT3 movec0, movec1;
{
   int i;
   for (i = 0; i < 3; i++) {
      verts[0][i] = verts[0][i] + move0[i];
      verts[1][i] = verts[1][i] + move1[i];
      colors[0][i] = colors[0][i] + movec0[i];
      colors[1][i] = colors[1][i] + movec1[i];
      if (verts[0][i] > RA && move0[i] > 0.0) {
      verts[0][i] = RA; move0[i] = -move0[i];
      }
      if (verts[0][i] < -RA && move0[i] < 0.0) {
      verts[0][i] = -RA; move0[i] = -move0[i];
      }
      if (verts[1][i] > RA && move1[i] > 0.0) {
      verts[1][i] = RA; move1[i] = -move1[i];
      }
      if (verts[1][i] < -RA && move1[i] < 0.0) {
      verts[1][i] = -RA; move1[i] = -move1[i];
      }

      if (colors[0][i] < 0.0 && movec0[0] < 0.0) {
      colors[0][i] = 0.0; movec0[i] = -movec0[i];
      }
      if (colors[0][i] > 1.0 && movec0[0] > 0.0) {
      colors[0][i] = 1.0; movec0[i] = -movec0[i];
      }
      if (colors[1][i] < 0.0 && movec1[1] < 0.0) {
      colors[1][i] = 0.0; movec1[i] = -movec1[i];
      }
      if (colors[1][i] > 1.0 && movec1[0] > 0.0) {
      colors[1][i] = 1.0; movec1[i] = -movec1[i];
      }
   }
}

float
ran(n)
float n;
{
   double drand48();
   return(n * drand48());
}
```

APPENDIX D

ON-LINE HELP FACILITY

Introduction

AVS makes it easy to supplement the on-line help facility with documentation for your own modules and / or networks. You can create a series of help files and have them accessible through the **Help** buttons and the **Show Module Documentation** button in the Module Editor window.

Help Files - Format and Naming Conventions

Each help screen in AVS is implemented as an ASCII text file, with a *.txt* filename suffix. In order to be portable the filename should be no more than 14 characters long including the .txt extension. The file is displayed in a Help Browser window using a monospace font (all characters have the same width). Thus, however you create the help file using a text editor is exactly how it appears in the browser. If you use TAB characters in help text files, be sure to set the tab stops in your text editor to every 8 columns. It may be safer to use SPACE characters to align columnar material instead.

You can include comment lines in your help files. Any line that begins with a period (**.**), pound-sign (**#**), or dash (**-**) character is suppressed when the file is displayed.

AVS looks for a help file based on either a *topic* string (module or network name) or a *filename* which is derived from the topic string. In order to generate the filename from a topic string, it takes the name of the module or network and replaces SPACE characters with underscores and appends ".txt". For example:

Module/Network Name	**Help Filename**
clarify edge	clarify_edge.txt
easy vu 2	easy_vu_2.txt

When possible, name your help file with a name that matches this convention. Obviously, with longer topic names, the filenames can become cumbersome and easily exceed the allowable length for filenames (14 characters on many systems).

A second and more convenient way is through the use of a *topics* file in the directory containing the help files. A list of all help topics, along with associated filenames, is stored in the ASCII file *.topics* which can be generated automatically by AVS or manually using a text editor. This file allows topic matching on longer names that may contain spaces and eliminates the need for the filename to follow the conventions mentioned above. There is one such file for each directory under */usr/avs/runtime/help*. For example, */usr/avs/runtime/help/.topics* contains the following:

```
avs_cmdopt.txt THE AVS COMMAND AND COMMAND-LINE OPTIONS
avs_dta.txt AVS DATA FILES
avs_envvar.txt AVS ENVIRONMENT VARIABLES
avs_mods.txt AVS MODULES
avs_start.txt AVS STARTUP FILE
avs_subsys.txt IMAGE VIEWER
fld_dtafmt.txt FIELD DATA FILE FORMAT
geo_dtafmt.txt GEOMETRY DATA FILE FORMAT
img_dtafmt.txt IMAGE DATA FILE FORMAT
vol_dtafmt.txt VOLUME DATA FILE FORMAT
```

Each line of the *.topics* file lists the name of one help file and the file's topic. This file can be written manually using any text editor or can be generated automatically by AVS using the -reindex option. For manual pages, the topic is automatically extracted from the line that follows the "NAME" heading. AVS looks for a pattern like the following:

```
AVS Modules                                        read image(6)

NAME
     read image     - read image file from disk into a field
```

AVS picks up "read image - read image from ..." as the topic line. It then searches for a match up to the first dash (-). For other help files, the topic line is just the first non-comment line. TAB characters are replaced by SPACE characters, and multiple SPACEs are compressed to a single SPACE.

When the Help Browser arrives in a particular directory of the help tree, it displays all the topics in that directory's *.topics* file, in place of the actual filenames. If a file in the directory does not have an entry in the *.topics* file, the filename itself is displayed. When the user clicks a particular topic, the Browser displays the corresponding file. When the **Show Module Documentation** operation is performed, it also uses the topics file to augment its search to match for module names based on topic name rather than filename.

Integrating Your Help Files into the Help System

There are two aspects to having your help files become part of the on-line help system. First, integrating the files into the AVS help facility. Second, integrating the files into the standard "man command" facility.

AVS Help

By default, AVS searches for help files in the directories under */usr/avs/runtime/help*. It is *not* advisable to store your help files in this location (in general, it is a bad idea to place user data in a "system" area). System areas may not be backed up, since they can be rebuilt from distribution tapes. Moreover, mixing user data and system data can cause problems when installing AVS releases in the future.

If you are writing your own modules for inclusion in the AVS system, create a help file for each module and place all the help files in a directory (e.g. */usr/derek/avs/help*). For portability, choose names with no more than 14 characters for the help files. Once this is done you need to create a .topics file for your directory and make that directory part of the search path that AVS uses when it is looking for help files.

You can set the AVS help search path using an environment variable (AVS_HELP_PATH) or an avsrc file option (HelpPath). A search path consists of a colon-separated list of complete pathnames of directories that contain help files or subdirectories of help files. The search path is used in addition to the standard */usr/avs/runtime/help* directory.

The -reindex Option and AVS_HELP_PATH

The command **avs -reindex** causes AVS to recreate the *.topics* file in each directory of the help tree. This command does not start an AVS session, but simply returns you to the UNIX prompt.

You can set the environment variable AVS_HELP_PATH to a colon-separated list of complete pathnames. If this variable is set, specifying the **-reindex** option (re)creates *.topics* files in the directory tree under each pathname in AVS_HELP_PATH. If this variable is not set, **-reindex** (re)creates *.topics* files in the standard directory tree, under */usr/avs/runtime/help*.

Another alternative: instead of using AVS_HELP_PATH to specify a colon-separated list of complete pathnames, you can place the list on the AVS command line:

avs -reindex *path*:*path* ...

An explicit argument overrides the current value of AVS_HELP_PATH, if any. Set your AVS_HELP_PATH environment variable:

```
setenv AVS_HELP_PATH /usr/derek/avs/help
```

Then, create a *.topics* file for the new help directory:

```
avs -reindex
```

The next time you run AVS, the new help files will be accessible through the Help Browser. You'll need to use the **New Dir** button to change to help directory */usr/derek/avs/help*. AVS automatically finds a module's help file when

you click **Show Module Documentation** in the Module Editor window, since AVS_HELP_PATH specifies the help file's directory tree.

AVS Help Search

The AVS_HELP_PATH variable is used by the Network Editor as follows:

- When you click the **Help** button in the Network Control Panel window (along the left edge of the screen), the name of the current network is converted to a filename by replacing SPACE characters with underscores and appending a *.txt* suffix. The help facility searches for either that filename or the unmodified topic string (module name or network) in the entire directory hierarchy under the first entry in AVS_HELP_PATH. If such a file exists or a matching topic entry in the .topics file is found, it is displayed in a Help Browser window. If not, the next entry in AVS_HELP_PATH is used, and so on.

  If no help file is found among all the AVS_HELP_PATH entries, a final search is made in the default help location, */usr/avs/runtime/help*. If this fails, an error message window pops up.
- The module icon for a user-written module includes the same small square as the AVS-supplied icons. You can click this square with the middle or right mouse button to bring up the Module Editor window. When you click the **Show Module Documentation** button, the help facility converts the module name to a filename and searches for the file, just as described in the preceding paragraph.

Man Command

You can use the *man*(1) command to view the AVS module help files. Installation of the help files into your hardware platform's *man* system is system dependent. Please consult your system documentation to find the procedure.

APPENDIX E

UNSTRUCTURED CELL DATA LIBRARY

Overview

The UCD package is a library containing the data structures and subroutines that allow users to write modules that handle Unstructured Cell Data (UCD). In particular, the UCD package can be used to create modules for Finite Element Analysis (FEA) and Computational Fluid Dynamics (CFD).

This chapter page is organized as follows:

SYNOPSIS section:

- Compiling and linking information
- Summary list of ucd routines, showing their parameters

DESCRIPTION section:

- An overview of how unstructured cell data is set up
- An overview of how ucd routines should be used
- A list of global variables
- Typedefs that are specific to unstructured cell data
- The file format for UCD data files

ROUTINES section:

- Structure Manipulation Routines
- Structure Query Routines
- Cell Manipulation Routines
- Cell Query Routines
- Node Manipulation Routines
- Node Query Routines

EXAMPLES section:

- C and FORTRAN examples of allocating and filling in a UCD structure

Synopsis

A C language module that uses ucd routines must use the following header file:

```
/usr/avs/include/ucd_defs.h
```

For example, in C:

```
#include<ucd_defs.h>
```

A FORTRAN language module that uses ucd routines must use the following header file:

```
/usr/avs/include/avs.inc
```

For example, in FORTRAN:

```
#include<avs.inc>
```

A module that uses ucd routines must be linked with the following libraries:

- C module

```
/usr/avs/lib/libflow_c.a
```

- C coroutine module

```
/usr/avs/lib/libsim_c.a
```

- FORTRAN module

```
/usr/avs/lib/libflow_f.a
```

- FORTRAN coroutine module

```
/usr/avs/lib/libsim_f.a
```

ucd Routine Summary

The following list of ucd routines is organized by functional category.

Structure Manipulation Routines

UCDstructure_alloc *(name, data_veclen, name_flag, ncells, cell_tsize, cell_veclen, nnodes, node_csize, node_veclen, util_flag*)
UCDstructure_free *(structure)*
UCDstructure_set_data *(structure, data)*
UCDstructure_set_data_labels *(structure, labels, delimiter)*
UCDstructure_set_data_units *(structure, labels, delimiter)*
UCDstructure_set_extent *(structure, min_extent, max_extent)*
UCDstructure_set_header_flag *(structure, util_flag)*
UCDstructure_set_mesh_id *(structure, mesh_id*)

Structure Query Routines

UCDstructure_get_data *(structure, data)*
UCDstructure_get_data_label *(structure, number, label)*
UCDstructure_get_data_labels *(structure, labels, delimiter)*
UCDstructure_get_data_unit *(structure, number, label)*
UCDstructure_get_data_units *(structure, labels, delimiter)*
UCDstructure_get_extent *(structure, min_extent, max_extent)*
UCDstructure_get_header *(structure, name, data_veclen, name_flag, ncells, cell_veclen, nnodes, node_veclen, util_flag)*
UCDstructure_get_mesh_id *(structure, mesh_id)*

Cell Manipulation Routines

UCDcell_set_information *(structure, cell, name, element_type, material_type, cell_type, mid_edge_flags, node_list)*
UCDstructure_invalid_cell_minmax *(structure)*
UCDstructure_set_cell_active *(structure, active)*
UCDstructure_set_cell_components *(structure, components, number)*
UCDstructure_set_cell_data *(structure, data)*
UCDstructure_set_cell_labels *(structure, labels, delimiter)*
UCDstructure_set_cell_minmax *(structure, min, max)*
UCDstructure_set_cell_units *(structure, labels, delimiter)*

Cell Query Routines

UCDcell_get_information *(structure, cell, name, element_type, material_type, cell_type, mid_edge_flags, node_list)*
UCDstructure_get_cell_active *(structure, active)*

UCDstructure_get_cell_components *(structure, components*)
UCDstructure_get_cell_data *(structure, data)*
UCDstructure_get_cell_label *(structure, number, label)*
UCDstructure_get_cell_labels *(structure, labels, delimiter)*
UCDstructure_get_cell_minmax *(structure, min, max)*
UCDstructure_get_cell_unit *(structure, number, label)*
UCDstructure_get_cell_units *(structure, labels, delimiter)*

Node Manipulation Routines

UCDnode_set_information *(structure, node, name, ncells, cell_list)*
UCDstructure_invalid_node_minmax *(structure)*
UCDstructure_set_node_active *(structure, active)*
UCDstructure_set_node_components *(structure, components, number)*
UCDstructure_set_cell_connect *(structure)*
UCDstructure_set_node_data *(structure, data)*
UCDstructure_set_node_labels *(structure, labels, delimiter)*
UCDstructure_set_node_minmax *(structure, min, max)*
UCDstructure_set_node_positions *(structure, x, y, z)*
UCDstructure_set_node_units *(structure, labels, delimiter)*

Node Query Routines

UCDnode_get_information *(structure, node, name, ncells, cell_list)*
UCDstructure_get_node_active *(structure, active)*
UCDstructure_get_node_components *(structure, components)*
UCDstructure_get_node_data *(structure, data)*
UCDstructure_get_node_label *(structure, number, label)*
UCDstructure_get_node_labels *(structure, labels, delimiter)*
UCDstructure_get_node_minmax *(structure, min, max)*
UCDstructure_get_node_positions *(structure, x, y, z)*
UCDstructure_get_node_unit *(structure, number, label)*
UCDstructure_get_node_units *(structure, labels, delimiter)*

Description

The Setup of the UCD Structure

In order to represent the unstructured data models found in FEA and CFD, a hierarchical model for the data structures has been chosen. At the top level, there is the object (objects will be referred to as ***structures***). Each structure is composed of multiple cells. The ***cells*** occupy the second level in the hierarchy. Each cell, in turn, is composed of multiple ***nodes***. The nodes are at the third level in the hierarchy. It is possible that more than one cell can contain the same node. An example of this three level hierarchy is seen in Figure E-1:

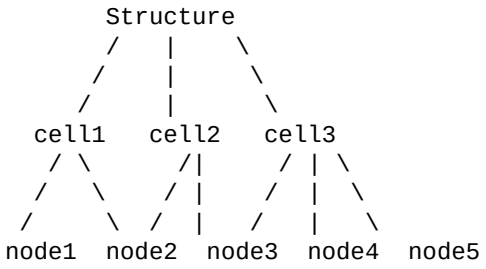

Figure E-1 Hierarchical Structure of Model, Cells and Nodes

Cells, Nodes, and Mid-Edge Nodes

The ucd data structure has eight different cell types. InFigure E-2 each of the cell types is pictured, with its nodes and mid-edge nodes correctly labeled.

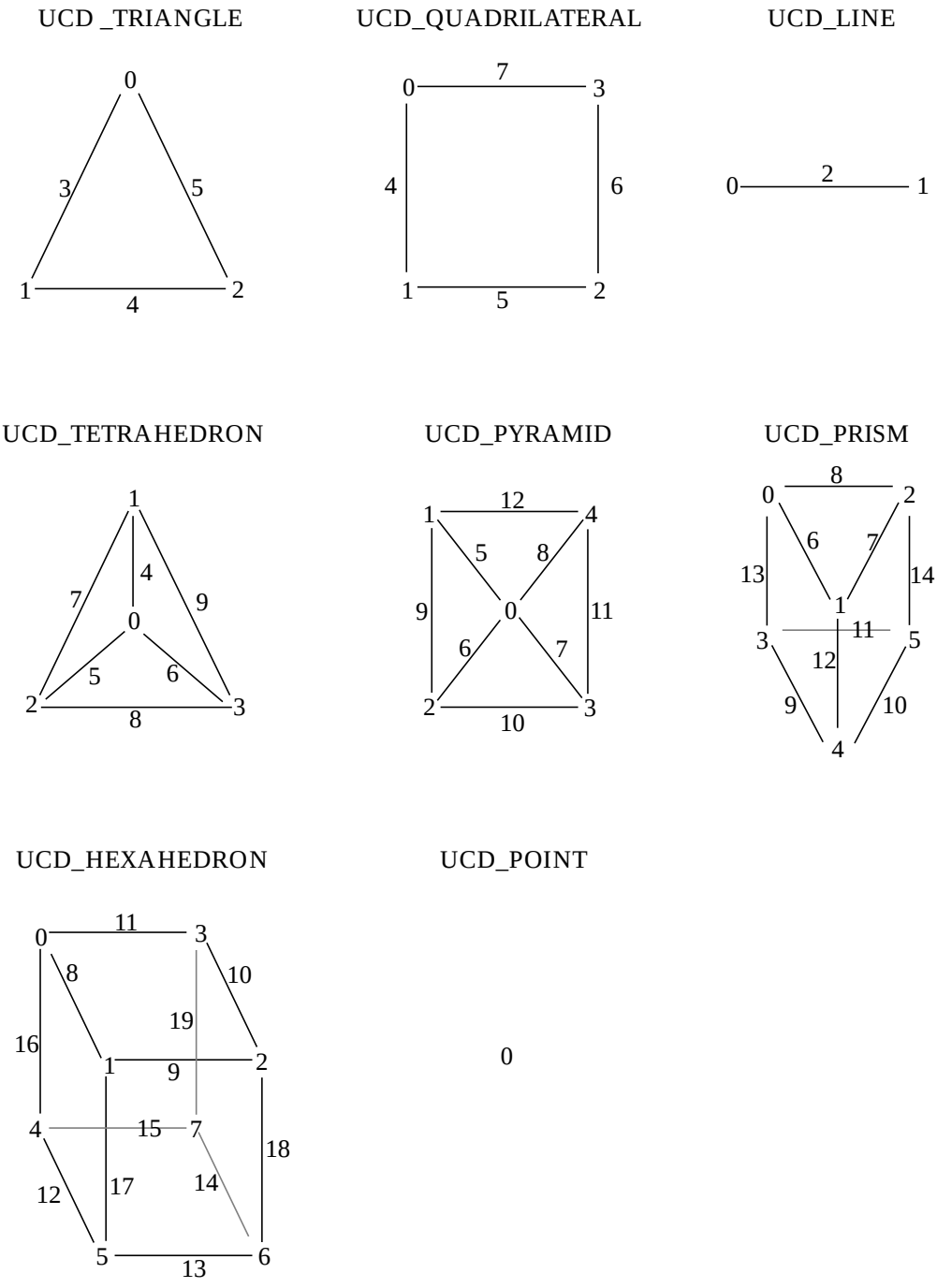

Figure E-2 UCD Cell Types, Nodes, Mid-Edge Nodes, and Node Numbering

Node connectivity (the list of nodes associated with each cell) should follow this numbering convention in UCD files and in the UCD structure.

Nodes are required. Mid-edge nodes are optional. The integer variable *mid_-edge_flags* in the UCD structure indicates which mid-edge nodes are present for a particular cell. Each bit, starting from the most right, indicates the presence of one mide-edge node. The arrangement is ordinal.

For example, if a cell is UCD_TRIANGLE and has 2 mid-edge nodes, 3 and 5, then the bits of the "mid_edge_flags" will be:

```
00........00101
            ---
```

The first rightmost bit indicates that the first mid-edge node present in a **UCD_TRIANGLE**, number 3, is present. The second mid-edge node in a **UCD_TRIANGLE**, number 4, is not present. The third mid-edge node, number 5, is present.

The following UCD modules supplied with this release of this product process mid-edge nodes:

read ucd
write ucd
ucd rslice
ucd probe
ucd to geom

Other modules ignore mid-edge nodes. Users can use the UCD library to write modules that manipulate mid-edge nodes.

UCD Data Structure and Type Definitions

```
typedef union {                         /* name union                        */
        char            *c;             /* character string for label names  */
        int             i;              /* integer for numerical names       */
}  UCD_name;



typedef struct UCD_structure_ {

        /*--------- Structure Header Information ---------*/
        char            *name;          /* structure name                    */
        int             name_flag;      /* are node/cell names chars or ints */
        int             ncells;         /* number of cells                   */
        int             nnodes;         /* number of nodes                   */
        float           min_extent[3];  /* structure extent                  */
        float           max_extent[3];  /* structure extent                  */
        int             data_veclen;    /* length of data vector for struct  */
        float           *data;          /* data for the structure            */
        char            *data_labels;   /* labels for data components        */
```

```
        char            *data_units;     /* labels for data units            */
        int             util_flag;       /* utility flag: all but the 16     */
                                         /* rightmost bits can be used       */
        int             mesh_id;         /* unique id of the mesh instance   */


        /*--------- Cell Information ---------*/
        UCD_name         *cell_name;      /* cell names                       */
        char            **element_type;  /* cell element types               */
        int             *material_type;  /* user defined material types      */
        int             *cell_type;      /* cell types (see above defines)   */
        int             cell_veclen;     /* length of data vector            */
        float           *cell_data;      /* data for cell-based datasets     */
        float           *min_cell_data;  /* min val for cell data components  */
        float           *max_cell_data;  /* max val for cell data components  */
        char            *cell_labels;    /* labels for cell data components   */
        char            *cell_units;     /* labels for cell data units       */
        int             *cell_components;/* array of cell component mix      */
        int             *cell_active;    /* array of active cell components  */
        int             *mid_edge_flags; /* cell edges with mid edge nodes   */
        int             node_conn_size;  /* size of the node connectivity list*/
        int             *node_list;      /* node list of connectivity        */
        int             *node_list_ptr;  /* location of a cell's node list   */
        int             ucd_last_cell;   /* number of last cell              */


        /*--------- Node Information ---------*/
        UCD_name         *node_name;      /* node names                       */
        float           *x, *y, *z;      /* position of the nodes            */
        int             node_veclen;     /* length of data vector            */
        float           *node_data;      /* data vector for the nodes        */
        float           *min_node_data;  /* min val for node data components  */
        float           *max_node_data;  /* max val for node data components  */
        char            *node_labels;    /* labels for node data components  */
        char            *node_units;     /* labels for node data units       */
        int             *node_components;/* array of node component mix      */
        int             *node_active;    /* array of active node components  */
        int             cell_conn_size;  /* size of the cell connectivity list*/
        int             *cell_list;      /* cell list of connectivity        */
        int             *cell_list_ptr;  /* location of a node's cell list   */
        int             ucd_last_node;   /* number of last node              */


        /*--------- Allocation Information (INTERNAL USE ONLY) ----------*/
        enum {
                UCD_ONE_BLOCK,
                UCD_RW_SHM,
                UCD_RO_SHM
        }               alloc_case;     /* storage allocation strategy      */
        int             shm_key;        /* shared memory key                */
        int             shm_id;         /* shared memory id                 */
        char            *shm_base;      /* shared memory base               */
} UCD_structure;
```

The **name_flag** in the structure header information can be used to allocate space for just those optional components that are actually used in your UCD structure, substantially reducing memory requirements for the data. Some of the structure elements (cell_name, element_type, material_type, mid_edge_-flags and node_name) are pointers to data storage that is optional. See the discussion under the **UCDstructure_alloc** call.

File Format for UCD Data Files

The UCD file format is a relatively simple format that can be written out in either binary or ASCII. The module **read ucd** can read files saved in this format.

NOTE: Binary files can be read in much faster. If you write your data file in ASCII, you can shorten the time it takes to read the file by converting it to binary format. In order to do this conversion, build a network with **read ucd** connected to **write ucd** and set **write ucd**'s output parameter to binary.

NOTE: Although cell- and model-based data can be stored within a ucd structure, and you can use the ucd library to manipulate such data, most of the ucd mapper modules supplied with AVS (e.g. ucd iso) will only work with node-based data. You can use the **ucd cell to node** module to convert cell data into a node data representation that can be processed by the supplied module set.

ASCII UCD File Format

The general order of the data is:

1. Numbers defining the overall structure, including the number of nodes, the number of cells, and the total length of the vector of data associated with the nodes, cells, and the model.
2. For each node, its node-id and the coordinates of that node in space. The UCD library allows node-ids to be either chars or integers (**UCDstructure_alloc**'s *name_flag* argument). However, the **read ucd** module only accepts integer node-ids. Any number, including non-sequential numbers, can be used. Mid-edge nodes are treated like any other node.
3. For each cell: its cell-id, material, type (hexahedral, pyramid, etc.), and the list of node-ids that correspond to each of the cell's verticies. This is the "node connectivity list". See above for the order in which node-ids are applied to a cell.)
4. For the data vector associated with nodes, how many components that vector is divided into, followed by the number of elements in each component (e.g., a vector of 5 floating point numbers may be treated as 3 components: a scalar, a vector of 3, and another scalar, which would be specified as 3 1 3 1).

5. For each node data component, a component label / unit label pair, separated by a comma.
6. For each node, the vector of data values associated with it.
7. That is the end of the node definitions. Cell-based data descriptions, if present, then follow in the same order and format as items 4, 5, and 6.
8. The single model-based data descriptions, if present, comes last.

The input file cannot contain blank lines or lines with leading blanks. Comment lines, denoted with a # sign in the first column, can occur before the data starts, but cannot be inserted within the data, either as an additional comment line, or at the end of a data line. The numbers down the left correspond to the above descriptions and are not part of the ASCII file.

```
  # <comment 1>
      .
      .
      .
  # <comment n>
1. <num_nodes> <num_cells> <num_ndata> <num_cdata> <num_mdata>
2. <node_id 1> <x> <y> <z>
  <node_id 2> <x> <y> <z>
      .
      .
      .
  <node_id num_nodes> <x> <y> <z>
3. <cell_id 1> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
  <cell_id 2> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
      .
      .
      .
  <cell_id num_cells> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
4. <num_comp for node data> <size comp 1> <size comp 2>...<size comp n>
5. <node_comp_label 1> , <units_label 1>
  <node_comp_label 2> , <units_label 2>
      .
      .
      .
  <node_comp_label num_comp> , <units_label num_comp>
6. <node_id 1> <node_data 1> ... <node_data num_ndata>
  <node_id 2> <node_data 1> ... <node_data num_ndata>
      .
      .
      .
  <node_id num_nodes> <node_data 1> ... <node_data num_ndata>
7. <num_comp for cell's data> <size comp 1> <size comp 2>...<size comp n>
  <cell-component-label 1> , <units-label 1>
  <cell-component-label 2> , <units-label 2>
      .
      .
      .
```

<cell-component-label n> , <units-label n>
<cell-id 1> <cell-data 1> ... <cell-data num_cdata>
<cell-id 2> <cell-data 1> ... <cell-data num_cdata>
.
.
.
<cell-id num_cells> <cell-data 1> <cell-data num_cdata>

8. <num_comp for model's data> <size comp 1> <size comp 2>...<size comp n>
<model-component-label 1> , <units-label 1>
<model-component-label 2> , <units-label 2>
.
.
.
<model-component-label n> , <units-label n>
<model-id> <model-data 1> <model-data num_mdata>

NOTE: mat_id = material id

NOTE: possible cell types are: (pt, line, tri, quad, tet, pyr, prism, hex)

The UCD structure and library will support either integer or character node-, cell-, and model-ids, (referred to in the library documentation as **names**). However, the **read ucd** module only accepts integer node-ids, cell-ids, and model-ids. This is shown in the example below.

Also note that, at present, most of the UCD modules do not make use of cell and model-based data, thus the input data examples all show "0" for <num-cdata> and <num-mdata>. User-written modules can use the UCD library to manipulate cell- and model-based data.

Example 1: The following is an example of a simple UCD file. This UCD structure has 8 nodes in 1 hexahedral cell. Associated with each node is a single scalar data value, making up one component that this person labels "stress," and specifies a "lb/in**2" unit label. There is no cell- or model-based data.

```
8 1 1 0 0                    <-1.  8 nodes, 1 cell, 1 component of node data
1  0.000   0.000   1.000     <-2.  for each node, its id and node coordinates
2  1.000   0.000   1.000
3  1.000   1.000   1.000
4  0.000   1.000   1.000
5  0.000   0.000   0.000
6  1.000   0.000   0.000
7  1.000   1.000   0.000
8  0.000   1.000   0.000
1  1 hex 1 2 3 4 5 6 7 8     <-3.  cell id, material id, cell type, cell vertices
1 1                          <-4.  num data components, size of each component
stress, lb/in**2             <-5.  component label, units label
1  4999.9999                 <-6.  data vector for each node
2  18749.9999
3  37500.0000
4  56250.0000
5  74999.9999
```

```
6  93750.0001
7  107500.0003
8  5000.0001
```

Example 2: This example has eight nodes in one hexahedral cell. Each node has three scalar components associated with it that are labelled "stress ...", and have "lb/in**2" unit strings. There is no cell or model data.

```
8 1 3 0 0
1  0.000   0.000   1.000
2  1.000   0.000   1.000
3  1.000   1.000   1.000
4  0.000   1.000   1.000
5  0.000   0.000   0.000
6  1.000   0.000   0.000
7  1.000   1.000   0.000
8  0.000   1.000   0.000
1  1 hex 1 2 3 4 5 6 7 8
3 1 1 1
stress sxx, lb/in**2
stress syy, lb/in**2
stress szz, lb/in**2
1       4999.9999     2187.5003     4999.9999
2      18749.9999        0.0003     5624.9999
3      37500.0000        0.0000    11250.0000
4      56250.0000        0.0000    16875.0000
5      74999.9999        0.0001    22499.9999
6      93750.0001       -0.0003    28125.0001
7     107500.0003    -2187.5006    28750.0003
8       5000.0001     2187.4997     5000.0001
```

Example 3: This example has eight nodes in one hexahedral cell. Each node has four data components (three scalar and one 3-vector). Therefore, num_n-data is equal to six. The cell has one scalar component associated with it labelled "compression" in units of "lb/in**2". The model has a character identifier "cube", two components, the first a scalar and the second a three-vector, which are labelled, but have no units.

```
8 1 6 1 4
1  0.000   0.000   1.000
2  1.000   0.000   1.000
3  1.000   1.000   1.000
4  0.000   1.000   1.000
5  0.000   0.000   0.000
6  1.000   0.000   0.000
7  1.000   1.000   0.000
8  0.000   1.000   0.000
1  1 hex 1 2 3 4 5 6 7 8
4 1 1 1 3
stress sxx, lb/in**2
stress syy, lb/in**2
stress szz, lb/in**2
disp, inches
1       4999.9999     2187.5003     4999.9999  0.234  0.0  0.023
2      18749.9999        0.0003     5624.9999  0.204  0.12 0.114
3      37500.0000        0.0000    11250.0000  0.224  0.10 0.114
```

```
4      56250.0000        0.0000    16875.0000  0.224  0.12 0.124
5      74999.9999        0.0001    22499.9999  0.234  0.12 0.124
6      93750.0001       -0.0003    28125.0001  0.134  0.12 0.124
7     107500.0003    -2187.5006    28750.0003  0.134  0.12 0.134
8       5000.0001     2187.4997     5000.0001  0.234  0.12 0.134
1 1
compression, lb/in**2
1 25947.0000
2 1 3
time ,
x mag,
y mag,
z mag,
cube  60   0.756  2.332  -4.079
```

Binary UCD File Format

The following describes the binary format for the UCD data files.

This AVS binary format is different from the binary UCD format in AVS 3. First, it has been reorganized. Following the magic number, there is all-structure information, then all-cell information, then all-model information. Second, the amount of space allocated for component label and unit character storage has been enlarged from 100 to 1024 bytes. Third, the size of the node, cell, and model component list has been made variable, depending upon the number of components and their vector length. It is no longer a fixed size that limits the total vector length of all components to 20 integers.

The **read ucd** module will read either AVS 3 or AVS 4 format binary data. **write ucd** writes AVS 4 format binary data.

```
1 byte   -   magic number. this should be 7.

4 bytes  -   number of nodes.  (int)

4 bytes  -   number of cells.  (int)

4 bytes  -   number of node data.  (int)

4 bytes  -   number of cell data.  (int)

4 bytes  -   number of model data.  (int)

4 bytes  -   number of nlist nodes (for cell topology).  (int)

(num_cells*16) bytes  -  cell information.  (4 ints per cell:
                  id, material id, number of nodes, cell type).

(num_nlist_nodes*4) bytes  -  cell topology lists.  (ints)

(num_nodes*4) bytes  -  x coordinates for nodes.  (floats)

(num_nodes*4) bytes  -  y coordinates for nodes.  (floats)
```

```
(num_nodes*4) bytes  -  z coordinates for nodes.  (floats)
```

If there is node data:

```
  1024 bytes  -   node data labels. (string)
  1024 bytes  -   node data units. (string)
    4 bytes  -   number of node components.  (int)
   (num_node_data*4)bytes  -   node component list. (ints)
   (num_node_data*4) bytes  -  minimums for node data. (floats)
   (num_node_data*4) bytes  -  maximums for node data. (floats)
   (num_nodes*num_node_data*4) bytes  -  node data. (floats)
   (num_node_data*4) bytes  -   node active list.  (ints)
```

If there is cell data:

```
  1024 bytes  -   cell data labels. (string)
  1024 bytes  -   cell data units. (string)
    4 bytes  -   number of cell components.  (int)
   (num_cell_data*4) bytes  -   cell component list. (ints)
   (num_cell_data*4) bytes  -  minimums for cell data. (floats)
   (num_cell_data*4) bytes  -  maximums for cell data. (floats)
   (num_cells*num_cell_data*4) bytes  -  cell data. (floats)
   (num_cell_data*4) bytes  -   cell active list. (ints)
```

If there is model data:

```
1024 bytes  -   model data labels. (string)
1024 bytes  -   model data units. (string)
  4 bytes  -   number of model components.  (int)
  (num_model_data*4) bytes -   model component list. (ints)
 (num_model_data*4) bytes  -  minimums for model data. (floats)
 (num_model_data*4) bytes  -  maximums for model data. (floats)
 (num_model_data*4) bytes  -  model data. (floats)
  (num_model_data*4) bytes -   model active list. (ints)
```

Structure Manipulation Routines

UCDstructure_alloc

C:

```
#include <ucd_defs.h>
char  *UCDstructure_alloc (name, data_veclen, name_flag,
                ncells, cell_tsize, cell_veclen,
                nnodes, node_csize, node_veclen, util_flag)
char        *name;
int         data_veclen;
int         name_flag;
int         ncells;
int         cell_tsize;
int         cell_veclen;
int         nnodes;
int         node_csize;
int         node_veclen;
int         util_flag;
```

FORTRAN:

```
#include <avs.inc>
CHARACTER*(*) UCDstruct_alloc (name, data_veclen, name_flag,
                ncells, cell_tsize, cell_veclen,
                nnodes, node_csize, node_veclen, util_flag)
CHARACTER*(*)  name
INTEGER        data_veclen
INTEGER        name_flag
INTEGER        ncells
INTEGER        cell_tsize
INTEGER        cell_veclen
INTEGER        nnodes
INTEGER        node_csize
INTEGER        node_veclen
INTEGER        util_flag
```

This function creates a new top level structure and returns a pointer to that structure. If a new structure could not be allocated then NULL is returned.

Following is a description of the arguments:

Input

name
structure name

data_veclen
length of structure data vector

name_flag
Indicates what information is stored in the UCD structure. It can be combined out of several flags using the "|" operator. The flags are:

UCD_CELL_NAMES
allocates space for cell_name in the UCD structure

UCD_CELL_TYPES
allocates space for element_type in the UCD structure

UCD_MATERIAL_IDS
allocates space for material_type in the UCD structure

UCD_MID_EDGES
allocates space for mid_edge_flags in the UCD structure

UCD_NODE_NAMES
allocates space for node_name in the UCD structure

We recommend that you examine your data for all these components and use only those flags that are needed for modules that you are planning to use. This will reduce the memory requirements for the UCD structure. For example if you want to allocate space for just the material ids and mid edge nodes, then you would set **name_flag** to **UCD_MATERIAL_IDS | UCD_MID_-EDGES**. By setting **name_flag** to zero, you will allocate none of the optional components.

ncells
number of cells in the structure

cell_tsize
expected size of the cell's connectivity list

cell_veclen
length of cell data vector

nnodes
number of nodes in the structure

node_csize
expected size of the node's connectivity list

node_veclen
length of node data vector

util_flag
utility flag for general usage (do NOT use 2 rightmost bits)

UCDstructure_free

C:

#include <ucd_defs.h>
int UCDstructure_free *(structure)*
UCD_structure **structure;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_free *(structure)*
INTEGER *structure*

This function frees the storage used by structure. It returns 1 if successful and 0 if failure.

Following is a description of the arguments:

Input

structure
structure to free

UCDstructure_set_data

C:

#include <ucd_defs.h>
int UCDstructure_set_data *(structure, data)*
UCD_structure **structure;*
float **data;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_data *(structure, data)*
INTEGER *structure*
REAL *data*

This function copies the data from the array pointed to by "data" into the structure's data array. There should be data_veclen data elements in this array. It returns 1 if successful and 0 if failure.

Following is a description of the arguments:

Input

structure
structure to find information

data
pointer to the data vector

UCDstructure_set_data_labels

C:

#include <ucd_defs.h>
int UCDstructure_set_data_labels *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_data_labels *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to set the labels for each component in the structure. These labels are for cases when there is structure based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to.

Example: labels = "temp;density;mach number"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_set_data_units

C:

#include <ucd_defs.h>
int UCDstructure_set_data_units *(structure, labels, delimiter)*
UCD_structure **structure;*

char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_data_units *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to set the unit labels for each component in the structure. These labels are for cases when there is structure based data. It returns 1 if successful and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as degrees, meters, etc.

Example: labels = "degrees;meters"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_set_extent

C:

#include <ucd_defs.h>
int UCDstructure_set_extent *(structure, min_extent, max_extent)*
UCD_structure **structure;*
float **min_extent;*
float **max_extent;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_extent *(structure, min_extent, max_extent)*
INTEGER *structure*
REAL *min_extent*
REAL *max_extent*

This routine allows the module writer to set the extent of the structure. It returns 1 if successful and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

min_extent
coordinate extent of structure

max_extent
coordinate extent of structure

UCDstructure_set_header_flag

C:

#include <ucd_defs.h>
int UCDstructure_set_header_flag *(structure, util_flag)*
UCD_structure **structure;*
int *util_flag;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_header_flag *(structure, util_flag)*
INTEGER *structure*
INTEGER *util_flag*

This function sets the header flag bits. This bit flag can be used for any purpose that the module writer wishes. It returns 1 if success, 0 if failure.

NOTE: in util_flag, the eight rightmost bits are reserved for internal usage.

Following is a description of the arguments:

Input:

structure
structure to find information

util_flag
utility flag

UCDstructure_set_mesh_id

C:

#include <ucd_defs.h>
int UCDstructure_set_mesh_id *(structure, mesh_id)*
UCD_structure **structure;*
int *mesh_id;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_mesh_id *(structure, mesh_id)*
INTEGER *structure*
INTEGER *mesh_id*

This function sets the **mesh_id**. It can be used by modules to set the mesh_id to signal downstream modules that the structure's mesh (i.e. node x,y,z positions) has changed. The **mesh_id** is originally set by the **UCDstructure_alloc** function.

Following is a description of the arguments:

Input:

structure
structure to find information

mesh_id
new mesh_id

Structure Query Routines

UCDstructure_get_data

C:

#include <ucd_defs.h>
int UCDstructure_get_data *(structure, data)*
UCD_structure **structure;*
float ***data;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_data *(structure, data)*
INTEGER *structure*
REAL *data*

This function returns a pointer to the array containing the data vector for the structure. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input

structure
structure to find information

data
pointer to the structure data vector

UCDstructure_get_data_label

C:

#include <ucd_defs.h>
int UCDstructure_get_data_label *(structure, number, label)*
UCD_structure **structure;*
int *number;*
char **label;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_data_label *(structure, number, label)*
INTEGER *structure*
INTEGER *number*
CHARACTER*(*) *label*

This routine allows the module writer to query the label for an individual component in the structure. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get labels in

number
individual component number

Output:

labels
string with labels included

UCDstructure_get_data_labels

C:

#include <ucd_defs.h>
int UCDstructure_get_data_labels *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_data_labels *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to get the labels for each component in the structure. These labels are for cases when there is structure based data. It returns 1 if success, 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to.

Example: labels = "temp;density;mach number"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_get_data_unit

C:

#include <ucd_defs.h>
int UCDstructure_get_data_unit *(structure, number, label)*
UCD_structure **structure;*
int *number;*
char **label;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_data_unit *(structure, number, label)*
INTEGER *structure*
INTEGER *number*
CHARACTER*(*) *label*

This routine allows the module writer to query the label for an individual unit in the structure. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get labels in

number
individual component number

Output:

labels
string with labels included

UCDstructure_get_data_units

C:

#include <ucd_defs.h>
int UCDstructure_get_data_units *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_data_units *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to get the unit labels for each component in the structure. These labels are for cases when there is structure based data. It returns 1 if success, 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as degrees, meters, etc.

Example: labels = "degrees;meters"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_get_extent

C:

#include <ucd_defs.h>
int UCDstructure_get_extent *(structure, min_extent, max_extent)*
UCD_structure **structure;*
float **min_extent;*
float **max_extent;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_extent *(structure, min_extent, max_extent)*
INTEGER *structure*
REAL *min_extent*
REAL *max_extent*

This routine allows the module writer to obtain the extent of the structure. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

min_extent
coordinate extent of structure

max_extent
coordinate extent of structure

UCDstructure_get_header

C:

#include <ucd_defs.h>
int UCDstructure_get_header *(structure, name, data_veclen, name_flag, ncells,*

UCD_structure **structure;*
char **name;*
int **data_veclen;*
int **name_flag;*
int **ncells;*
int **cell_veclen;*
int **nnodes;*
int **node_veclen;*
int **util_flag;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_header *(structure, name, data_veclen, name_flag,*
ncells, cell_veclen, nnodes,
node_veclen, util_flag)
INTEGER *structure*
CHARACTER*(*) *name*
INTEGER *data_veclen*
INTEGER *name_flag*
INTEGER *ncells*
INTEGER *cell_veclen*
INTEGER *nnodes*
INTEGER *node_veclen*
INTEGER *util_flag*

This function finds out all the header information about a UCD_structure and returns those values. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

name
structure name

data_veclen
length of structure data vector

name_flag
are node / cell names chars or ints; char = **UCD_CHAR**, int = **UCD_INT**

ncells
number of cells in the structure

cell_veclen
length of cell data vector

nnodes
number of nodes in the structure

node_veclen
length of node data vector

util_flag
utility flag

UCDstructure_get_mesh_id

C:

#include <ucd_defs.h>
int UCDstructure_get_mesh_id *(structure, mesh_id)*
UCD_structure **structure;*
int **mesh_id;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_mesh_id *(structure, mesh_id)*
INTEGER *structure*
INTEGER *mesh_id*

This function gets the mesh_id from a UCD structure. It can be used by UCD modules to determine if the input mesh (i.e. node x,y,z positions) has changed since the last time the module received the structure input from an upstream module.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

mesh_id
current mesh_id for the structure

Cell Manipulation Routines

UCDcell_set_information

C:

```
#include <ucd_defs.h>
int UCDcell_set_information (structure, cell, name, element_type,
                             material_type, cell_type,
                             mid_edge_flags, node_list)
UCD_structure  *structure;
int            cell;
int            name;
char           *element_type;
int            material_type;
int            cell_type;
int            mid_edge_flags;
int            *node_list;
```

FORTRAN:

```
#include <avs.inc>
INTEGER UCDcell_set_information (structure, cell, name,
                                 element_type, material_type,
                                 cell_type, mid_edge_flags,
                                 node_list)
INTEGER        structure
INTEGER        cell
INTEGER        name
CHARACTER*(*)  element_type
INTEGER        material_type
INTEGER        cell_type
INTEGER        mid_edge_flags
INTEGER        node_list
```

This function sets all the information about a particular cell. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

cell
cell to find information

name
cell name

element_type
name of element type

material_type
user defined material type

cell_type
cell type (e.g. **UCD_TRIANGLE**)

mid_edge_flags
does the cell have mid edge nodes

node_list
array of node numbers

UCDstructure_invalid_cell_minmax

C:

#include <ucd_defs.h>
int UCDstructure_invalid_cell_minmax *(structure)*
UCD_structure **structure;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_invalid_cell_minmax *(structure)*
INTEGER *structure*

This routine allows the module writer to set the min / max range of the structure cell data to be invalid. This function should be used after the structure data has been changed by the module and the module does not want to spend the time recomputing the cell minmax. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to set cell min / max invalid

UCDstructure_set_cell_active

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_active *(structure, active)*
UCD_structure **structure;*
int **active;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_active *(structure, active)*
INTEGER *structure*
INTEGER *active*

This function sets the array containing the cell active component list. For instance, if there are four different components in the cell data vector and the module is using the second component, the list would be: (0 1 0 0) It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

active
pointer to the cell active component list

UCDstructure_set_cell_components

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_components *(structure, components, number)*
UCD_structure **structure;*
int **components;*
int *number;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_components *(structure, components, number)*

INTEGER *structure*
INTEGER *components*
INTEGER *number*

This function copies the array containing the cell component list. For instance, if there are four different components in the cell data vector (e.g. scalar, 3-vector, 2-vector, scalar), the component list would be: (1 3 2 1) It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

components
pointer to the cell component list

number
number of components in the list

UCDstructure_set_cell_data

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_data *(structure, data)*
UCD_structure **structure*;
float **data;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_data *(structure, data)*
INTEGER *structure*
REAL *data*

This function copies the cell data from the array pointed to by "data" into the structure's cell data array. There should be cell_veclen*ncells data elements in this array. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

data
pointer to the cell data vectors

UCDstructure_set_cell_labels

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_labels *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_labels *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to set the labels for each component in the structure. These labels are for cases when there is cell based data. It returns 1 if success, 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to.

Example: labels = "temp;density;mach number"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_set_cell_minmax

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_minmax *(structure, min, max)*
UCD_structure **structure;*
float **min;*
float **max;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_minmax *(structure, min, max)*
INTEGER *structure*
REAL *min*
REAL *max*

This routine allows the module writer to set the range of the structure cell data. It should be noted that min and max are arrays of dimension structure->cell_veclen. It returns 1 if success, 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to set min / max in

min
value of minimum data point

max
value of maximum data point

UCDstructure_set_cell_units

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_units *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_units *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to set the unit labels for each component in the structure. These labels are for cases when there is cell based data. It returns 1 if success, 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as degrees, meters, etc.

Example: labels = "degrees;meters"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

labels
string with labels included

delimiter
delimiter between each label

Cell Query Routines

UCDcell_get_information

C:

#include <ucd_defs.h>
int UCDcell_get_information *(structure, cell, name, element_type, material_type, cell_type, mid_edge_flags, node_list)*
UCD_structure **structure;*
int *cell;*
int **name;*
char **element_type;*
int **material_type;*
int **cell_type;*
int **mid_edge_flags;*
int ***node_list;*

FORTRAN:

#include <avs.inc>
INTEGER UCDcell_get_information *(structure, cell, name, element_type, material_type, cell_type, mid_edge_flags, node_list)*
INTEGER *structure*

INTEGER *cell*
INTEGER *name*
CHARACTER*(*) *element_type*
INTEGER *material_type*
INTEGER *cell_type*
INTEGER *mid_edge_flags*
INTEGER *node_list*

This function finds out all the information about a particular cell and returns those values. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

cell
cell to find information

Output:

name
cell name

element_type
name of element type

material_type
user defined material type

cell_type
cell type (e.g. **UCD_TRIANGLE**) data

data
data for cell -based datasets

mid_edge_flags
does the cell have mid edge nodes

node_list
array of node numbers

UCDstructure_get_cell_active

C:

```
#include <ucd_defs.h>
int UCDstructure_get_cell_active (structure, active)
```

UCD_structure **structure;*
int ***active;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_active *(structure, active)*
INTEGER *structure*
INTEGER *active*

This function returns a pointer to the array containing the cell active component list. For instance, if there are four different components in the cell data vector and the module is using the second component, the list would be: (0 1 0 0) The active list is useful when trying to communicate from one module to another which component in the cell data is being worked on. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

active
pointer to the cell active component list

UCDstructure_get_cell_components

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_components *(structure, components)*
UCD_structure **structure;*
int ***components;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_components *(structure, components)*
INTEGER *structure*
INTEGER *components*

This function returns pointers to the array containing the cell component list. For instance, if there are four different components in the cell data vector (e.g. scalar, 3-vector, 2-vector, scalar), the component list would be: (1 3 2 1) It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

components
pointer to the cell component list

UCDstructure_get_cell_data

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_data *(structure, data)*
UCD_structure **structure;*
float ***data;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_data *(structure, data)*
INTEGER *structure*
REAL *data*

This function returns a pointer to the array containing the data vectors for all of the cells in the structure. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

data
pointer to the cell data vectors

UCDstructure_get_cell_label

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_label *(structure, number, label)*
UCD_structure **structure;*

int *number;*
char **label;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_label *(structure, number, label)*
INTEGER *structure*
INTEGER *number*
CHARACTER*(*) *label*

This routine allows the module writer to query the label for an individual component in the structure. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get labels in

number
individual component number

Output:

labels
string with labels included

UCDstructure_get_cell_labels

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_labels *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_labels *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to get the labels for each component in the structure. These labels are for cases when there is cell based data. It returns 1 if success and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to.

Example: labels = "temp;density;mach number"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_get_cell_minmax

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_minmax *(structure, min, max)*
UCD_structure **structure;*
float **min;*
float **max;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_minmax *(structure, min, max)*
INTEGER *structure*
REAL *min*
REAL *max*

This routine allows the module writer to obtain the range of the structure cell data. It should be noted that min and max are arrays of dimension structure->cell_veclen. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get min / max in

Output:

min
value of minimum data point

max
value of maximum data point

UCDstructure_get_cell_unit

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_unit *(structure, number, label)*
UCD_structure **structure;*
int *number;*
char **label;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_unit *(structure, number, label)*
INTEGER *structure*
INTEGER *number*
CHARACTER*(*) *label*

This routine allows the module writer to query the label for an individual unit in the structure. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get labels in

number
individual component number

Output:

labels
string with labels included

UCDstructure_get_cell_units

C:

#include <ucd_defs.h>
int UCDstructure_get_cell_units *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_cell_units *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to get the unit labels for each component in the structure. These labels are for cases when there is cell based data. It returns 1 if success and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as degrees, meters, etc.

Example: labels = "degrees;meters"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

labels
string with labels included

delimiter
delimiter between each label

Node Manipulation Routines

UCDnode_set_information

C:

#include <ucd_defs.h
int UCDnode_set_information *(structure, node, name, ncells, cell_list)*
UCD_structure **structure;*
int *node;*

int *name;*
int *ncells;*
int **cell_list;*

FORTRAN:

#include <avs.inc>
INTEGER UCDnode_set_information *(structure, node, name, ncells, cell_list)*
INTEGER *structure*
INTEGER *node*
INTEGER *name*
INTEGER *ncells*
INTEGER *cell_list*

This function sets all the information about a particular node. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

node
node to find information

name
node name

ncells
number of cells in cell_list

cell_list
array of cell numbers

UCDstructure_invalid_node_minmax

C:

#include <ucd_defs.h>
int UCDstructure_invalid_node_minmax *(structure)*
UCD_structure **structure;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_invalid_node_minmax *(structure)*
INTEGER *structure*

This routine allows the module writer to set the min / max range of the structure node data to be invalid. This function should be used after the structure data has been changed by the module and the module does not want to spend the time recomputing the node minmax. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to set node min / max invalid

UCDstructure_set_cell_connect

C:

#include <ucd_defs.h>
int UCDstructure_set_cell_connect *(structure)*
UCD_structure **structure;*
float **min_extent;*
float **max_extent;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_cell_connect *(structure)*
INTEGER *structure*
REAL *min_extent*
REAL *max_extent*

This routine calculates and sets the cell connectivity list (the list of cells connected to each node) for the structure. (Terminology note: The "cell connectivity list" is, for each node, the list of cells of which it is a member. It is defined by the variable *cell_list* in the **node** data structure. This call calculates and sets this value. The "node connectivity list" is, for each cell, the list of nodes that are a member of it. It is defined by the variable *node_list* in the **cell** data structure. It is defined explicitly for each cell in the input data.)

The space for the cell connectivity list (*cell_conn_size*) must be allocated by **UCDstructure_alloc()**; its argument *node_csize* must be the maximum possible size of the cell connectivity list (usually no more than *cell_tsize*).

The programmer would use this call instead of **UCDnode_set_information**'s *cell_list* parameter; *cell_list* requires that the programmer supply the cell connectivity explicitly.

The information produced by this call can be retrieved with the **UCDnode_-get_information** call's *cell_list* argument.

The function returns 1 if successful and 0 if failure.

The following is a description of the arguments:

Input:

structure
UCD structure whose cell connectivity list to calculate.

UCDstructure_set_node_active

C:

#include <ucd_defs.h>
int UCDstructure_set_node_active *(structure, active)*
UCD_structure **structure;*
int **active;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_node_active *(structure, active)*
INTEGER *structure*
INTEGER *active*

This function sets the array containing the node active component list. For instance, if there are four different components in the node data vector and the module is using the second component, the list would be: (0 1 0 0) The active list is useful when trying to communicate from one module to another which component in the node data is being worked on. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

active
pointer to the node active component list

UCDstructure_set_node_components

C:

#include <ucd_defs.h
int UCDstructure_set_node_components *(structure, components, number)*
UCD_structure **structure;*
int **components;*
int *number;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_node_components *(structure, components, number)*
INTEGER *structure*
INTEGER *components*
INTEGER *number*

This function copies the array containing the node component list. For instance, if there are four different components in the node data vector (e.g. scalar, 3-vector, 2-vector, scalar), the component list would be: (1 3 2 1) It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

components
pointer to the node component list

number
number of components in the list

UCDstructure_set_node_data

C:

#include <ucd_defs.h>
int UCDstructure_set_node_data *(structure, data)*
UCD_structure **structure;*
float **data;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_node_data *(structure, data)*
INTEGER *structure*
REAL *data*

This function copies the node data from the array pointed to by "data" into the structure's node data array. There should be node_veclen*nnodes data elements in this array. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

data
pointer to the node data vectors

UCDstructure_set_node_labels

C:

#include <ucd_defs.h>
int UCDstructure_set_node_labels *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_node_labels *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to set the labels for each component in the structure. These labels are for cases when there is node based data. It returns 1 if success and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to.

Example: labels = "temp;density;mach number"
delimiter = ";"

Input:

structure
structure to find information

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_set_node_minmax

C:

int UCDstructure_set_node_minmax *(structure, min, max)*
UCD_structure **structure;*
float **min;*
float **max;*

FORTRAN:

INTEGER UCDstruct_set_node_minmax *(structure, min, max)*
INTEGER *structure*
REAL *min*
REAL *max*

This routine allows the module writer to set the range of the structure node data. It should be noted that min and max are arrays of dimension structure->node_veclen. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to set min / max in

min
value of minimum data point

max
value of maximum data point

UCDstructure_set_node_positions

C:

#include <ucd_defs.h>
int UCDstructure_set_node_positions *(structure, x, y, z)*
UCD_structure **structure;*
float **x, *y, *z;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_node_positions *(structure, x, y, z)*
INTEGER *structure*
REAL *x, y, z*

This function copies the x, y and z coordinate arrays from the arrays pointed to by "x", "y" and "z" into the structure's node position arrays. There should be nnodes coordinates in each array. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

"**x, y, z**"
pointer to the x,y,z arrays

UCDstructure_set_node_units

C:

#include <ucd_defs.h>
int UCDstructure_set_node_units *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_set_node_units *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to set the unit labels for each component in the structure. These labels are for cases when there is node based data. It returns 1 if success and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as degrees, meters, etc.

Example: labels = "degrees;meters"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

labels
string with labels included

delimiter
delimiter between each label

Node Query Routines

UCDnode_get_information

C:

#include <ucd_defs.h>
int UCDnode_get_information *(structure, node, name, ncells, cell_list)*
UCD_structure **structure;*
int *node;*
int **name;*
int **ncells;*
int ***cell_list;*

FORTRAN:

#include <avs.inc>
INTEGER UCDnode_get_information *(structure, node, name, ncells, cell_list)*
INTEGER *structure*
INTEGER *node*
INTEGER *name*
INTEGER *ncells*
INTEGER *cell_list*

This function finds out all the information about a particular node and returns those values. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

node
node to find information

Output:

name
node name

ncells
number of cells in cell_list

cell_list
array of cell numbers

UCDstructure_get_node_active

C:

#include <ucd_defs.h>
int UCDstructure_get_node_active *(structure, active)*
UCD_structure **structure;*
int ***active;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_active *(structure, active)*
INTEGER *structure*
INTEGER *active*

This function returns a pointer to the array containing the node active component list. For instance, if there are four different components in the node data vector and the module is using the second component, the list would be: (0 1 0 0) It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

active
pointer to the node active component list

UCDstructure_get_node_components

C:

#include <ucd_defs.h>
int UCDstructure_get_node_components *(structure, components)*
UCD_structure **structure;*
int ***components;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_components *(structure, components)*

INTEGER *structure*
INTEGER *components*

This function returns pointers to the array containing the node component list. For instance, if there are four different components in the node data vector (e.g. scalar, 3-vector, 2-vector, scalar), the component list would be: (1 3 2 1) It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

components
pointer to the node component list

UCDstructure_get_node_data

C:

#include <ucd_defs.h>
int UCDstructure_get_node_data *(structure, data)*
UCD_structure **structure;*
float ***data;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_data *(structure, data)*
INTEGER *structure*
REAL *data*

This function returns pointers to the array containing the data vectors for the nodes. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

data
pointer to the node data vectors

UCDstructure_get_node_label

C:

#include <ucd_defs.h>
int UCDstructure_get_node_label *(structure, number, label)*
UCD_structure **structure;*
int *number;*
char **label;*

FORTRAN:

#include <ucd_defs.h>
INTEGER UCDstruct_get_node_label *(structure, number, label)*
INTEGER *structure*
INTEGER *number*
CHARACTER*(*) *label*

This routine allows the module writer to query the label for an individual component in the structure. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get labels in

number
individual component number

Output:

labels
string with labels included

UCDstructure_get_node_labels

C:

#include <ucd_defs.h>
int UCDstructure_get_node_labels *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_labels *(structure, labels, delimiter)*
INTEGER *structure*

CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to get the labels for each component in the structure. These labels are for cases when there is node based data. It returns 1 if success and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as temperature, density, mach number, etc. In turn, these labels would appear on the dials so that the user would have a better understanding of which component each dial is attached to.

Example: labels = "temp;density;mach number"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

labels
string with labels included

delimiter
delimiter between each label

UCDstructure_get_node_minmax

C:

#include <ucd_defs.h>
int UCDstructure_get_node_minmax *(structure, min, max)*
UCD_structure **structure;*
float **min;*
float **max;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_minmax *(structure, min, max)*
INTEGER *structure*
REAL *min*
REAL *max*

This routine allows the module writer to obtain the range of the structure node data. It should be noted that min and max are arrays of dimension structure->node_veclen. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get min / max in

Output:

min
value of minimum data point

max
value of maximum data point

UCDstructure_get_node_positions

C:

#include <ucd_defs.h>
int UCDstructure_get_node_positions *(structure, x, y, z)*
UCD_structure **structure;*
float ***x, **y, **z;*

FORTRAN:

#include <ucd_defs.h>
INTEGER UCDstruct_get_node_positions *(structure, x, y, z)*
INTEGER *structure*
REAL *x, y, z*

This function returns pointers to the arrays containing the x, y and z coordinates of node positions. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

x, y, z
pointer to the x,y,z arrays

UCDstructure_get_node_unit

C:

#include <ucd_defs.h>
int UCDstructure_get_node_unit *(structure, number, label)*
UCD_structure **structure;*
int *number;*
char **label;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_unit *(structure, number, label)*
INTEGER *structure*
INTEGER *number*
CHARACTER*(*) *label*

This routine allows the module writer to query the label for an individual unit in the structure. It returns 1 if success and 0 if failure.

Following is a description of the arguments:

Input:

structure
structure to get labels in

number
individual component number

Output:

labels
string with labels included

UCDstructure_get_node_units

C:

#include <ucd_defs.h>
int UCDstructure_get_node_units *(structure, labels, delimiter)*
UCD_structure **structure;*
char **labels;*
char **delimiter;*

FORTRAN:

#include <avs.inc>
INTEGER UCDstruct_get_node_units *(structure, labels, delimiter)*
INTEGER *structure*
CHARACTER*(*) *labels*
CHARACTER*(*) *delimiter*

This routine allows the module writer to get the unit labels for each component in the structure. These labels are for cases when there is node based data. It returns 1 if success and 0 if failure.

For instance, in the case of a CFD dataset, the module writer might want to label components of the field as degrees, meters, etc.

Example: labels = "degrees;meters"
delimiter = ";"

Following is a description of the arguments:

Input:

structure
structure to find information

Output:

labels
string with labels included

delimiter
delimiter between each label

Examples

This section presents a step-by-step example of how to go about building a UCD structure using the supplied UCD routines. Both FORTRAN and C examples are given. Although this example is not a complete rundown of all the UCD routines, it does give a general overview.

For other examples of UCD routines, the */usr/avs/examples* directory contains several source files: *ucd_extract.c, ucd_thresh.c,* and *gen_ucd.f.*

Allocating a New Structure

The first thing that you will want to do is allocate the UCD structure. The subroutine **UCDstructure_alloc** does this allocation and builds the initial structure. Also, the extent of the structure will be set. The extent is the minimum / maximum value for the structure in coordinate space.

C Code:

```
/* C */

#include <ucd_defs.h>

   UCD_structure  ucd_struct;
```

```
int            data_veclen, name_flag, util_flag;
int            ncells, cell_tsize, cell_veclen;
int            nnodes, node_csize, node_veclen;

/* there will be no model or cell data */
data_veclen = 0;
cell_veclen = 0;

/* no optional components will be allocated in the structure */
name_flag = 0;

/* this structure has 1000 cells and 1000 nodes */
ncells = 1000;
nnodes = 1000;

/* in order to allocate the connectivity list, the expected */
/* size of the cell connectivity list and node connectivity */
/* list must be set.                                        */
cell_tsize = 500;
node_csize = 600;

/* for each node, there will be 5 data values */
node_veclen = 5;

/* use the default utility flag */
util_flag = 0;

/* Everything is setup, so allocate the structure */
ucd_struct = UCDstructure_alloc ('ucd name', data_veclen, name_flag,
                                 ncells, cell_tsize, cell_veclen,
                                 nnodes, node_csize, node_veclen,
                                 util_flag)

/* check to make sure that the structure was properly allocated */
if (ucd_struct == 0) {
   printf ("ERROR: the structure was not properly allocated\en");
   return;
}  /* end if */

/* store the structure's extent: (Note: in this example, it is assumed  */
/* that the floating point values xmin_dim, ymin_dim, etc. have already */
/* been set to be the minimum and maximum ranges of the data in coord-  */
/* inate space.                                                         */
min_extent[0] = xmin_dim; max_extent[0] = xmax_dim;
min_extent[1] = ymin_dim; max_extent[1] = ymax_dim;
min_extent[2] = zmin_dim; max_extent[2] = zmax_dim;
error_flag = UCDstructure_set_extent (ucd_struct, min_extent, max_extent);
if (error_flag == 0) {
   printf ("ERROR: can't set structure extent\en");
   return;
}  /* end if */
```

FORTRAN code:

```
C       FORTRAN

#include <avs.inc>

        integer         ucd_struct
       1                data_veclen, name_flag, util_flag
       2                ncells, cell_tsize, cell_veclen
       3                nnodes, node_csize, node_veclen

C       there will be no model or cell data
        data_veclen = 0
        cell_veclen = 0

C       do not allocate space for the optional structure components
        name_flag = 0

C       this structure has 1000 cells and 1000 nodes
        ncells = 1000
        nnodes = 1000

C       in order to allocate the connectivity list, the expected
C       size of the cell connectivity list and node connectivity
C       list must be set.
        cell_tsize = 500
        node_csize = 500

C       for each node, there will be 5 data values
        node_veclen = 5

C       use the default utility flag
        util_flag = 0

C       Everything is setup, so allocate the structure
        ucd_struct = UCDstruct_alloc ('ucd name', data_veclen, name_flag,
       1                                ncells, cell_tsize, cell_veclen,
       2                                nnodes, node_csize, node_veclen,
       3                                util_flag)

C       check to make sure that the structure was properly allocated
        if (ucd_struct .eq. 0) then
           write (0, *) 'ERROR: the structure was not properly allocated'
           error_flag = 0
           return
         end if

C       store the structure's extent: (Note: in this example, it is assumed
C       that the floating point values xmin_dim, ymin_dim, etc. have already
C       been set to be the minimum and maximum ranges of the data in coord-
C       inate space.
        min_extent(1) = xmin_dim
        max_extent(1) = xmax_dim
        min_extent(2) = ymin_dim
        max_extent(2) = ymax_dim
        min_extent(3) = zmin_dim
```

```
      max_extent(3) = zmax_dim
      error_flag = UCDstruct_set_extent (ucd_struct, min_extent,
     1                                   max_extent)
      if (ucd_struct .eq. 0) then
         write (0, *) 'ERROR: can''t set structure extent'
         error_flag = 0
         return
       end if
```

Storing Information About the Nodes

After the structure has been allocated, you can start storing information about the structure. Start by storing information for the nodes.

C Code:

```
/* C */

   /* store the node information: (Note: in this example, the creation */
   /* of each node's connectivity list is not correct. it is assumed   */
   /* that the you will know what your connectivity list looks like.   */
   /* this example assumes that each node is connected to 2 cells.)    */
   for (i = 0; i < nnodes; i++) {
      count = 2;
      conn_list[0] = i;  conn_list[1] = i+1;
      error_flag = UCDnode_set_information (ucd_struct, i, i, count,
conn_list);
      if (error_flag == 0) {
         printf ("ERROR: can't set node information\en");
         return;
      }  /* end if */
   }  /* end for */


   /* store the node coordinates: (Note: in this example, it is assumed */
   /* that the float arrays xc, yc and zc have already been created.    */
   /* these arrays contain the X coordinate, Y coordinate and Z coord-  */
   /* inate location for each node in the structure.)                   */
   error_flag = UCDstructure_set_node_positions (ucd_struct, xc, yc, zc);
   if (error_flag == 0) {
      printf ("ERROR: can't set node positions\en");
      return;
   }  /* end if */


   /* store the component list which specifies the length of each        */
   /* data component. in this example, the vector length for each node   */
   /* node is 5.  the first component in the vector is temperature,      */
   /* which is a scalar.  the second component in the vector is velocity */
   /* which is a 3-vector.  the third component in the vector is density */
   /* which is a scalar.  so the component list is (1 3 1).              */
   num_components = 3;
   comp_list[0] = 1;  comp_list[1] = 3;  comp_list[2] = 1;
   error_flag = UCDstructure_set_node_components (ucd_struct, comp_list,
```

```
                                                    num_components);
   if (error_flag == 0) {
      printf ("ERROR: can't set node components\en");
      return;
   }  /* end if */


   /* in this example, we want the density component (the third component) */
   /* to be the active component.                                          */
   active_list[0] = 0;  active_list[1] = 0;  active_list[2] = 1;
   error_flag = UCDstructure_set_node_active (ucd_struct, active_list);
   if (error_flag == 0) {
      printf ("ERROR: can't set node active list\en");
      return;
   }  /* end if */

   /* as described above, each node's vector is composed of three      */
   /* components: (temperature, velocity, density).  below, the labels */
   /* for these components are set.                                    */
   sprintf (data_labels, "temperature.velocity.density");
   error_flag = UCDstructure_set_node_labels (ucd_struct, data_labels, '.');
   if (error_flag == 0) {
      printf ("ERROR: can't set node labels\en");
      return;
   }  /* end if */


   /* store the node data: (Note: in this example, it is assumed */
   /* that the float array node_data has already been created.   */
   error_flag = UCDstructure_set_node_data (ucd_struct, node_data);
   if (error_flag == 0) {
      printf ("ERROR: can't set node data\en");
      return;
   }  /* end if */


   /* store the minimum and maximum node data values for each component. */
   /* (Note: in this example, it is assumed that the float arrays        */
   /* mn_node_data and mx_node_data have already been created.           */
   error_flag = UCDstructure_set_node_minmax (ucd_struct, mn_node_data,
                                              mx_node_data)
   if (error_flag == 0) {
      printf ("ERROR: can't set node minimum and maximum data values\en");
      return;
   }  /* end if */
```

FORTRAN Code:

```
C       FORTRAN


C       store the node information: (Note: in this example, the creation
C       of each node's connectivity list is not correct. it is assumed
C       that the you will know what your connectivity list looks like.
C       this example assumes that each node is connected to 2 cells.)
        do i = 1, nnodes
```

```
            count = 2
            conn_list(1) = i
            conn_list(2) = i+1
            error_flag = UCDnode_set_information (ucd_struct, i, i,
           1                                      count, conn_list);
            if (error_flag .eq. 0) then
               write (0, *) 'ERROR: can''t set node information'
               return
             end if
          end do


C         store the node coordinates: (Note: in this example, it is assumed
C         that the real arrays xc, yc and zc have already been created.
C         these arrays contain the X coordinate, Y coordinate and Z coord-
C         inate location for each node in the structure.)
          error_flag = UCDstruct_set_node_positions (ucd_struct, xc, yc, zc)
          if (error_flag .eq. 0) then
             write (0, *) 'ERROR: can''t set node positions'
             return
          end if


C         store the component list which specifies the length of each
C         data component. in this example, the vector length for each node
C         node is 5.  the first component in the vector is temperature,
C         which is a scalar.  the second component in the vector is velocity
C         which is a 3-vector.  the third component in the vector is density
C         which is a scalar.  so the component list is (1 3 1).
          num_components = 3
          comp_list(1) = 1
          comp_list(2) = 3
          comp_list(3) = 1
          error_flag = UCDstruct_set_node_components (ucd_struct, comp_list,
         1                                            num_components)
          if (error_flag .eq. 0) then
             write (0, *) 'ERROR: can''t set node components'
             return
          end if


C         in this example, we want the density component (the third component)
C         to be the active component.
          active_list(1) = 0
          active_list(2) = 0
          active_list(3) = 1
          error_flag = UCDstruct_set_node_active (ucd_struct, active_list)
          if (error_flag .eq. 0) then
             write (0, *) 'ERROR: can''t set node active list'
             return
          end if


C         as described above, each node's vector is composed of three
C         components: (temperature, velocity, density).  below, the labels
C         for these components are set.
```

```
         data_labels = 'temperature.velocity.density'
         error_flag = UCDstruct_set_node_labels (ucd_struct, data_labels, '.')
         if (error_flag .eq. 0) then
            write (0, *) 'ERROR: can''t set node labels'
            return
         end if


C        store the node data: (Note: in this example, it is assumed
C        that the real array node_data has already been created.
         error_flag = UCDstruct_set_node_data (ucd_struct, node_data)
         if (error_flag .eq. 0) then
            write (0, *) 'ERROR: can''t set node data'
            return
         end if


C        store the minimum and maximum node data values for each component.
C        (Note: in this example, it is assumed that the real arrays
C        mn_node_data and mx_node_data have already been created.
         error_flag = UCDstruct_set_node_minmax (ucd_struct, mn_node_data,
        1                                                    mx_node_data)
         if (error_flag .eq. 0) then
            write (0, *) 'ERROR: can''t set node min and max data'
            return
         end if
```

Storing Information about the Cells

Now that node information has been stored, you can start storing information about the cells in the structure.

C Code:

```
/* C */

   /* create and store each cell. in this example, it is assumed that   */
   /* the structure is made of hexahedrons (UCD_HEXAHEDRON) arranged in */
   /* a grid (x_dim  x  y_dim  x  z_dim).                               */
   n = 0;
   offset = 0;
   z_off = (x_dim + 1) * (y_dim + 1);
   ucd_cell_type = UCD_HEXAHEDRON;
   for (z = 0; z < zdim-1; z++) {
      for (y = 0; y < ydim-1; y++) {
         for (x = 0; x < xdim-1; x++) {
            node_list[4] = offset + x;
            node_list[5] = node_list[4] + 1;
            node_list[6] = node_list[5] + x_dim + 1;
            node_list[7] = node_list[4] + x_dim + 1;

            node_list[0] = node_list[4] + z_off;
            node_list[1] = node_list[5] + z_off;
            node_list[2] = node_list[6] + z_off;
```

```
            node_list[3] = node_list[7] + z_off;

            error_flag = UCDcell_set_information (ucd_struct, n, n, 'brick',
                                                  1, ucd_cell_type, 0,
                                                  node_list);
            if (error_flag == 0) {
               printf ("ERROR: can't set cell information\en");
               return;
            }  /* end if */
            n++;
          }  /* end for x */
          offset = offset + x_dim + 1;
       }  /* end for y */
       offset = z_off * (z + 1);
    }  /* end for z */
```

FORTRAN Code:

```
C       FORTRAN

C       create and store each cell. in this example, it is assumed that
C       the structure is made of hexahedrons (UCD_HEXAHEDRON) arranged in
C       a grid (x_dim  x  y_dim  x  z_dim).
        n = 0
        offset = 0
        z_off = (x_dim + 1) * (y_dim + 1)
        ucd_cell_type = 7

        do z = 0, z_dim - 1
           do y = 0, y_dim - 1
              do x = 0, x_dim - 1
                 node_list(5) = offset + x
                 node_list(6) = node_list(5) + 1
                 node_list(7) = node_list(6) + x_dim + 1
                 node_list(8) = node_list(5) + x_dim + 1

                 node_list(1) = node_list(5) + z_off
                 node_list(2) = node_list(6) + z_off
                 node_list(3) = node_list(7) + z_off
                 node_list(4) = node_list(8) + z_off

               error_flag = UCDcell_set_information(ucd_struct, n, n, 'brick',
     1                                                1, ucd_cell_type, 0,
     2                                                node_list)

                 if (error_flag .eq. 0) then
                    write (0, *) 'ERROR: can''t set cell information'
                    return
                 end if
                 n = n + 1
              end do
              offset = offset + x_dim + 1
           end do
           offset = z_off * (z + 1)
        end do
```

Examples

APPENDIX F

FIELD ARGUMENTS IN FORTRAN

Introduction

For compatibility with an older method of accessing fields from FORTRAN, it is possible to pass a field to a FORTRAN computation subroutine as multiple arguments. This appendix supplies the details needed to read and understand older code.

Note that you should use the techniques described in this section for backward compatibility with FORTRAN code written for AVS2. When developing new FORTRAN modules, refer to the "Manipulating Fields from FORTRAN" section of Chapter 2.

This section also discusses support for allocating memory blocks in FORTRAN using some portable AVS functions that were provided in the AVS2.0P release. These functions may still be of some use but memory allocation should now be handled automatically as part of the new approach at handling fields.

The newer field elements, such as labels, units, and extents, are not passed in as arguments to avoid breaking any existing FORTRAN code. If these elements are required in modules using this approach, the programmer should use the new function, **AVSport_field()** to obtain the field pointer and then pass this value to the new field accessor functions to obtain the elements of interest.

Note that you should not use the techniques described in this appendix for new module development. Refer to Chapter 3 for information on developing modules in FORTRAN and to Chapter 2 for information on the field data type.

Field passing using multiple arguments

In passing fields as arguments to FORTRAN subroutines, AVS generates several arguments for each input port, output port, or parameter declared to be a field, unless the declaration routine calls the function **AVSset_module_flags**. For example, a computation routine that takes as its first input port a "field 3D 3-vector real rectilinear" would be defined as follows (if the

AVS flow executive has not been instructed to pass a single argument via **AVSset_module_flags**):

```
FUNCTION COMPUTE(DATA, NX, NY, NZ, COORDS, ...)
DIMENSION DATA(3, NX, NY, NZ)
DIMENSION COORDS(NX + NY + NZ)
...
```

In this example the single input port has generated five function arguments. The argument *DATA* represents the data of the field, the arguments *NX*, *NY*, and *NZ* represent the three dimensions of the field in computational space, and *COORDS* provides the rectilinear mapping from computational space to coordinate space. The coordinates for the data element *DATA(N,\ I,\ J,\ K)* are as follows:

```
X = COORDS(I)
Y = COORDS(NX + J)
Z = COORDS(NX + NY + K)
```

To see how the subroutine arguments change based on how the input is defined, assume that the function above takes two-dimensional data instead of three dimensional data; it is declared as a "field 2D 3-vector real rectilinear". Then the computation function is defined as follows:

```
FUNCTION COMPUTE(DATA, NX, NY, COORDS, ...)
DIMENSION DATA(3, NX, NY)
DIMENSION COORDS(NX + NY)
...
```

Finally, assume that the field is irregular, with a two-dimensional coordinate space. The field is declared as a "field 2D 3-vector real 2-coordinate irregular". Then the computation function is defined as follows:

```
 FUNCTION COMPUTE(DATA, NX, NY, NCOORD,
+    COORDS, ...)
 DIMENSION DATA(3, NX, NY)
 DIMENSION COORDS(NX, NY, NCOORD)
 ...
```

The following table defines the arguments to a FORTRAN computation function for the complete combination of possible field declaration strings:

Table F-1 Field Arguments for FORTRAN Routines

Field Component	Port Specification	Input or Parameter Argument(s)	Output Arguments
Data	Vector Length not 0	Array: DATA(*)	Pointer to DATA(*)
	Vector Length=0	[No Argument]	[No Argument]
Number of Dimensions	Not Specified	NDIM	NDIM
	Specified	[No Argument]	[No Argument]
Dimensions	NDIM Not Specified	Array: IDIMS(NDIM)	Pointer to IDIMS(NDIM)
	NDIM Specified	IDIM1, IDIM2, IDIM3, ...	IDIM1, IDIM2, IDIM3, ...

Table F-1 Field Arguments for FORTRAN Routines

Field Component	Port Specification	Input or Parameter Argument(s)	Output Arguments
Vector Length	Not Specified	IVLEN	IVLEN
	Specified	[No Argument]	[No Argument]
Data Type	Not Specified	ITYPE	ITYPE
	Specified	[No Argument]	[No Argument]
Mapping Type	Not Specified	IFLAG, NCOORD, COORDS(*)	IFLAG, NCOORD, Pointer to COORDS(*)
COORDS	Rectilinear	COORDS(*)	Pointer to COORDS(*)
	Irregular	NCOORD, COORDS(*)	NCOORD, Pointer to COORDS(*)
	Uniform	[No Argument]	[No Argument]

In this table, you can determine the order of the arguments by reading down the left-hand column. Thus, for a field, if the vector length is declared to be other than 0, the data array is always the first argument. If the number of dimensions is not specified in the declaration string, the number of dimensions is always the next argument. If there is a [No\ Argument] in the column specifying the condition that matches the declaration string you're using, there is no argument at all corresponding to that field component.

In the following example, a computation routine has a field input argument and a field output argument. Both the input port and the output port are specified as "field 3D scalar real uniform".

```
FUNCTION COMPUTE(F, NX, NY, NZ, GP, MX, MY, MZ)
DIMENSION F(NX, NY, NZ), G(NX, NY, NZ)
INTEGER GP
POINTER (GP, G)
...
MX = NX
MY = NY
MZ = NZ
GP = MALLOC(NX*NY*NZ*4)
...
```

In this example, the computation routine maps one 3D field onto another. The actual computation has been omitted; instead we focus on the setup and allocation. The first four arguments to the subroutine represent the input port and the second four arguments represent the output port. Note that the input array is presented directly while the output array is presented via a pointer so that we can allocate the space for it. We do this by setting *MX*, *MY*, and *MZ* and then using the **MALLOC**(3C) routine to allocate the array. (In the call to **MALLOC**, 4 is the number of bytes in a **REAL** data value.)

Array Allocation

As part of handling fields as individual arguments on the compute function calling stack, the module writer needed to handle allocating blocks of memory for the field data and points arrays and also needed to "decode" references to memory blocks that had been allocated in C for input field data. In AVS2.0, some use of POINTER variables was used for this purpose as noted in the examples above. However, the POINTER feature is not a standard FORTRAN feature and cannot be used across a number of platforms so its use is discouraged in favor of treating memory block addresses as simple integers and using several AVS functions to perform memory allocation and referencing operations. These functions are generally not necessary if the new FORTRAN field passing conventions are used since the new accessor functions handle memory allocation and referencing in their own way.

Memory Allocation and Application Portability

The **MALLOC**(3f) dynamic memory allocation function is not a standard FORTRAN 77 function. It varies from implementation to implementation. For example, on one system:

(C language function)

```
unsigned int  size;
char *malloc (size)
```

(FORTRAN language subroutine)

```
EXTERNAL MALLOC
INTEGER SIZE, ADDR
CALL MALLOC (SIZE, ADDR)
```

In the example FORTRAN environment, the memory pointer is passed as an argument to the MALLOC function, and is modified to contain the location of the newly allocated space. In the example C environment, the function follows the C language convention of returning the pointer to the newly allocated space as a function value.

Since dynamic memory allocation is not a standard FORTRAN concept, relying on nonstandard extensions such as pointers makes modules less portable. Two AVS interface functions (**AVSptr_alloc** and **AVSptr_offset**) allow blocks of data passed into the compute routines to be referenced in a completely portable way. In particular, these functions avoid use of pointer variables and the POINTER statement.

These two new routines accept an integer argument passed into the compute function (a data block memory pointer) and a local array of the appropriate type (a reference location). They return an offset index (*N*) into the local array, such that a reference to the *N+1*th element of the local array accesses the first element of the data block.

See Appendix A for descriptions of the **AVSptr_alloc** and **AVSptr_offset** routines.

APPENDIX G

GEOMETRY LIBRARY

Introduction

This appendix contains the documentation for the Geometry Library. The Geometry (geom) Library is a command driven interface to most of the functionality found in AVS's Geometry Viewer. The Geometry Library provides support for geometry processing in AVS.

There are two ways you can use this library to create geometric objects and have AVS display these objects. One way is to create a program that writes geometric data to a geometry file which the AVS geometry viewer can read. Another way is to create an AVS module that outputs a geometry data type containing geometric and attribute information. You can feed the output of this module to a module that accepts the geom data type (usually the **geometry viewer** module) and use the Geometry Viewer to produce an interactive display of the geometric data.

For both a geometry file and a geometry producing AVS module, you must create one or more "geom" objects. A geom object contains a set of graphics primitives. A geom object has no attributes associated with it and has a single object type. The types of objects currently supported by this library are the following:

- Meshes
- Polyhedron
- Polytriangles
- Spheres
- Labels

These object types are the building blocks through which you can create geometric scenes in AVS. The following sections describe these types in more detail.

With the Geometry Library you can read and write files of 3D geometric data and manipulate geometry and the associated attributes. The library also optimizes geometry rendering for the particular machine architecture on which it is used. You can use this library to define new filters to convert data into other format and you can create AVS modules that output a geometry data type.

The remainder of this chapter is organized as follows:

SYNOPSIS section:

- Compiling and linking information
- Summary list of geom routines, showing their parameters

OVERVIEW section:

- An overview of the basic geometry objects

DESCRIPTION section:

- An overview of how geom routines should be used

ROUTINES section:

Descriptions of geom routines, grouped under the appropriate topics:

- Object Creation Routines
- Object Utility Routines
- Object Property Routines
- Object Texture-Mapping Routines
- Object File Utilities
- AVS Module Interface Routines

FORTRAN BINDING section:

- A discussion of the FORTRAN calling sequences for geom routines

Synopsis

Compiling and Linking

A C language application that uses geom routines must use the following header file:

/usr/avs/include/geom.h

A FORTRAN application that uses geom routines must use the following header file:

/usr/avs/include/geom.inc

To link programs or modules containing geom routines, you should use the example Makefile in */usr/avs/examples*. Each of the template compile/link commands contains a series of macro symbols that expand out to include the correct compiler options and libraries (in the correct order) for your platform. In the C case, the symbol FLOWLIBS expands out to a symbol BASELIBS, which includes the base AVS libraries necessary for a compile; plus a symbol LASTLIBS. LASTLIBS contains platform-specific libraries to link with. LASTLIBS itself is defined explicitly in the file */usr/avs/include/Makeinclude*,

which */usr/avs/examples/Makefile* includes as its first step. The compiler flags are defined by the macro symbol CFLAGS, which expands to ACFLAGS and AVS_INC. ACFLAGS is defined in */usr/avs/include/Makeinclude*; AVS_INC is defined within the Makefile.

Using the example Makefile ensures that your modules will compile and link correctly on your platform. Pick a template compile/link command that corresponds to your module's type (subroutine or coroutine) and source language (C or FORTRAN). If you need to modify or supplement the compiler options or libraries, be sure to study the expansion of the macros and insert your modifications in the correct place without leaving any of the necessary base or platform-specific options or librairies out.

Routine Listing

The following list of GEOM routines is organized by functional category. Complete routine descriptions follow in the remainder of this appendix.

Object Creation Routines

GEOMadd_disjoint_line(*obj, verts, colors, n, alloc*)
GEOMadd_disjoint_polygon(*obj, verts, normals, colors, nverts, flag, alloc*)
GEOMadd_disjoint_prim_data(*obj, pdata, n, alloc)*
GEOMadd_disjoint_vertex_data(*obj, vdata, n, alloc)*
GEOMadd_float_colors(*obj, colors, n, alloc*)
GEOMadd_int_colors(*obj, colors, n, alloc*)
GEOMadd_label(*obj, text, ref_point, offset, height, color, label_flags*)
GEOMadd_normals(*obj, normals, n, alloc*)
GEOMadd_polygon(*obj, nverts, indices, flags, alloc*)
GEOMadd_polygons(*obj, plist, flags, alloc*)
GEOMadd_polyline(*obj, verts, colors, n, alloc*)
GEOMadd_polyline_prim_data(*obj, pdata, i, n, alloc)*
GEOMadd_polyline_vertex_data(*obj, vdata, i, n, alloc)*
GEOMadd_polytriangle(*obj, verts, normals, colors, n, alloc*)
GEOMadd_polytriangle_prim_data(*obj, pdata, i, n, alloc)*
GEOMadd_polytriangle_vertex_data(*obj, vdata, i, n, alloc)*
GEOMadd_prim_data(*obj, pdata, n, alloc)*
GEOMadd_radii(*obj, radii, n, alloc*)
GEOMadd_vertex_data(*obj, pdata, n, alloc)*
GEOMadd_vertices(*obj, verts, n, alloc*)
GEOMadd_vertices_with_data(*obj, verts, normals, colors, n, alloc*)
GEOMcreate_label(*extent, label_flags*)
GEOMcreate_label_flags(*font_number, title, background, drop, align, stroke*)
GEOMcreate_mesh(*extent, verts, m, n, alloc*)
GEOMcreate_mesh_with_data(*extent, verts, normals, colors, m, n, alloc*)
GEOMcreate_obj(*type, extent*)
GEOMcreate_polyh(*extent, verts, n, plist, flags, alloc*)
GEOMcreate_polyh_with_data(*extent, verts, normals, colors, n,*

plist, flags, alloc)
GEOMcreate_scalar_mesh(*xmin, xmax, ymin, ymax, mesh, colors, n, m, alloc*)
GEOMcreate_sphere(*extent, verts, radii, normals, colors, n, alloc*)
GEOMdestroy_obj(*obj*)
GEOMget_font_number(*name, bold, italic*)

Object Utility Routines

GEOMauto_transform(*obj*)
GEOMauto_transform_non_uniform(*obj*)
GEOMauto_transform_list(*objs, n*)
GEOMauto_transform_non_uniform_list(*objs, n*)
GEOMcreate_normal_object(*obj,scale*)
GEOMctv_mesh_to_polytri(*tobj, flags*)
GEOMcvt_polyh_to_polytri(*tobj, flags*)
GEOMflip_normals(*obj*)
GEOMgen_normals(*obj, flags*)
GEOMnormalize_normals(*obj*)
GEOMset_computed_extent(*obj, extent*)
GEOMset_extent(*obj*)
GEOMset_object_group(*obj, name*)
GEOMset_pickable(*obj, pickable*)
GEOMunion_extents(*obj1, obj2*)

Object Property Routines

GEOMadd_int_value(*obj, type, value*)
GEOMquery_int_value(*obj, type, value*)
GEOMset_color(*obj, color*)

Object Texture-Mapping Routines

GEOMadd_polytriangle_uvs(*obj, uvs, i, n, alloc*)
GEOMadd_uvs(*obj, uvs, n, alloc*)
GEOMcreate_mesh_uvs(*obj, umin, vmin, umax, vmax*)
GEOMdestroy_uvs(*obj*)

Object Vertex Transparency Routines

GEOMadd_vertex_trans(*obj, vtrans, n, alloc*)
GEOMadd_polytriangle_vertex_trans(*obj, vtrans, i, n, alloc*)

Object File Utilities

GEOMread_obj(*fd, flags*)
GEOMread_text(*fp, flags*)
GEOMwrite_obj(*obj, fd, flags*)
GEOMwrite_text(*obj, fp, flags*)

Object Debugging Routines

GEOMcheck_obj(*name, flags, func*)

AVS Module Interface Routines

GEOMdestroy_edit_list(*list*)
GEOMedit_backface(*list, name, mode*)
GEOMedit_camera_orient(*list, name, flags, scale, at, up, from*)
GEOMedit_camera_params(*list, name, options, val*)
GEOMedit_camera_project(*list, name, flags, front, back, fov, wsize*)
GEOMedit_center(*list, name, center*)
GEOMedit_clip_plane(*edit_list, obj, clip, state*)
GEOMedit_color(*list, name, color*)
GEOMedit_concat_matrix(*list, name, matrix*)
GEOMedit_depth_cue_params(*list, name, flags, depth_front, depth_back, depth_scale*)
GEOMedit_geometry(*list, name, obj*)
GEOMedit_parent(*list, name, parent*)
GEOMedit_light(*list, name, type, status*)
GEOMedit_position(*list, name, position*)
GEOMedit_properties(*list, name, ambient, diffuse, specular, pec_exp, transparency, spec_col*)
GEOMedit_projection(*list, name, parent*)
GEOMedit_render_mode(*list, name, mode*)
GEOMedit_selection_mode(*list, name, mode, flags*)
GEOMedit_set_matrix(*list, name, matrix*)
GEOMedit_subdivision(*list, name, subdiv*)
GEOMedit_transform_mode(*list, name, redirect, flags*)
GEOMedit_texture(*list, name, texture*)
GEOMedit_texture_options(*list, name, options, val*)
GEOMedit_visibility(*list, name, visibility*)
GEOMedit_window(*list, name, window*)
GEOMinit_edit_list(*list*)

Overview: AVS Geometry Object Data Structure

Unlike traditional graphics programming interfaces, AVS has a very rigid object data base structure. This allows users to access the data base at a very high level. Instead of adding and deleting "structure elements" as you might in PHIGS, for example, AVS allows users to create and manipulate AVS objects. An AVS object has a list of geometry, a list of child AVS objects, and a set of attributes which can either be defined for this object, or inherited from the object's parent. An AVS object also has a single parent object that defines where it fits into the object hierarchy.

When a new geometry viewer scene is created, there is a single AVS object called "top". By default, this object has no geometry and no child objects. When you read the example geometry file, *teapot.geom*, into the geometry viewer using the **Read Object** button, AVS creates a single AVS object called "teapot.1". This object is initially made a child of the object "top" and inherits all of its attributes from the object "top".

Note that AVS objects are distinct from geom objects. More than one geom object can be associated with a single AVS object. For example, the *teapot.geom* geometry file contains a separate mesh geom object for each major piece of the teapot, as well as one for the handle and one for the lid. etc. In fact, there are 32 geom objects for that single AVS object. It is useful to think of geom objects as primitives.

Geometry Object Types (Geometry Primitives)

This section describes each individual geom type.

Mesh Objects

The mesh object type contains a 2D array of vertices ordered such that adjacent vertices in the array are connected. If the dimensions of the 2D vertex array are MxN, a mesh object forms (M-1)*(N-1) quadrilaterals. A single quadrilateral is a simple mesh object with M=N=2.

Polyhedrom Objects

A polyhedron object contains a 2D (number of vertices x 3) array specifying the X, Y, and Z coordinates of each vertex and a separate list of connectivity information. The connectivity list is a 1D array of integers. The first integer in the list (call it *n*) contains the number of vertices in the first polygon of the polyhedron. Following this integer are *n* consecutive integers (beginning with 1) representing the element in the vertex array that corresponds to the respective vertex data. The last index of the first polygon is followed by an integer representing the number of vertices in the second polygon. This pattern continues until the value of *n* is zero, which terminates the list. The following example shows the contents of the connectivity list used to describe a

polyhedron containing two polygons, one with four vertices and the second with five vertices.

```
Connectivity list: {4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 0}
```

The index values in the connectivity list are "1 based". This makes no difference to the C programmer because AVS interprets the index "1" to indicate the first vertex in the vertex array.

The object types mesh and polyhedron contain an implied surface and wireframe description of the geometry that they represent. For the mesh object, the wireframe description is a 2D array of lines across the rows and columns of the mesh. For the polyhedron object, the wireframe description of the object is formed by the edges of each polygon in the object.

In AVS, the primary object types used to create surface descriptions of geometric objects are the mesh and polyhedron object types. You can easily create a third common description of polygonal data, disjoint polygons, using the "polyhedron" object type. This library provides routines to make this conversion easy.

Polytriangle Objects

Certain rendering platforms can make use of the shared vertices in adjacent polygons to improve the efficiency of rendering. For this reason, we have added a third primitive that represents surface information, the polytriangle strip. In a polytriangle strip, each vertex makes a triangle with the previous two vertices. For a list of N vertices, there are N-2 triangles. If your object is such that each vertex is shared by a number of different triangles and your hardware is most efficient at rendering this type of primitive, the polytriangle strip can be the most efficient description in which to represent your object. The geom library provides routines to convert objects from the mesh and polyhedron primitive types into the polytriangle primitive.

Note that since the polytriangle primitive can only represent triangles, it does not normally contain information necessary for providing an appropriate wireframe description of an object. If we have an object made up entirely of quadrilaterals and used the polytriangle representation as our wireframe description of the object, AVS would naturally draw edges in our object that should not be drawn. For this reason, the polytriangle object type contains both a wireframe and a surface description of the object. If the rendering mode is "lines" the geometry viewer uses the wireframe description, if it is "surface", it uses the surface description.

The surface description is an array of polytriangle strips (where each strip is a single array of connected triangles). The wireframe description contains an array of connected lines and an array of disjoint lines. Each connected line is a single array of vertices such that each vertex draws a line to the previous vertex. If a connected line has N vertices, it contains N-1 lines. The array of disjoint lines contains an even number of vertices such that each successive pair of vertices forms a line. If there are N vertices in a disjoint line, there are N / 2 lines.

Usually, the polytriangle strip is not the best choice for representing surface data. It can be efficient for representing objects that have only wireframe information, however. If an object has only wireframe information AVS draws these lines even if the rendering mode is set to surface.

Sphere Objects

The sphere object is used to represent objects that contain a list of spheres or dots. Spheres have a radius as well as a location. Dots are represented as spheres without any radius information.

Label Objects

Label objects are used to represent text that generally annotates or titles other geometric data. There are three classes of labels: annotation labels, titles, and "stroke" labels. Titles have a location in screen coordinates and are not transformed by either the camera or the object's transformation. Annotation labels are transformed with geometry but are always parallel to the screen plane. Stroke text is transformed like geometry, however, it is not supported on all platforms.

Geometry File Filters

Writing a geometry program that outputs data to a file is most suitable for batch programs and one-time conversions of geometric data. AVS provides examples of these types of filters that convert standard geometry file formats, such as Wavefront and movie BYU, into the AVS geometry file format. (See */usr/avs/filters/wfront_geom.c* and */usr/avs/filters/byu.c* for these examples.)

In general, each geom files contains the geometry for a single AVS object. However, the geom file contains virtually no support for setting object attributes or hierarchy. Once the data is in a geom file, you can specify geometry viewer attributes using a separate object (OBJ) script file that associates one or more geom files with hierarchy and attribute information. You can also use OBJ script files to create flipbook animations of geometric objects. See the "Geometry Viewer Script Language" appendix in the *AVS User's Guide* for more information.

By default, a single AVS object is kept in a single geom file. You can override this to allow multiple AVS objects to be generated from multiple geom objects in a single geom file (see the **geomset_object_group** routine and the "Geometry Viewer Script Language " appendix of the *AVS User's Guide* for more information).

Geometry Producing Modules

An AVS module can dynamically modify the geometry and attributes of AVS objects in the Geometry Viewer in a general programmatic way. AVS provides many geometry producing modules. The isosurface module produces a different geometric object for each new threshold parameter sent to the module. With an AVS module, you can control much more than just the geometry of an object. The module can change the object hierarchy, change object attributes, orientations, etc. With upstream data, the module can receive picking information from the user and can obtain information about how an object is being transformed.

Each **geometry viewer** module produces a single geometry scene. When a geometry producing module executes, it causes the **geometry viewer** module to make changes to the scenes defined by all of the downstream **geometry viewer** modules (usually there is just one downstream **geometry viewer** module and just one scene). For example, a module may specify a change in the color of the object named "top", or rotate the object named "slice" by 90 degrees.

When using the Geometry Viewer, the user and the module operate on the same database and can affect the object in much the same way. The Geometry Viewer allows you to rotate objects, change attributes, add new objects, etc. Modules can use the geom library described in this appendix to perform the same actions.

Typically, the modules provide only a subset of the information required to view the data, allowing you to make the rest of the controls with the Geometry Viewer. The module may, for example, provide geometry for an object, but not specify the orientation matrix or other attributes. You can then make these settings using the Geometry Viewer.

The Edit List

During each execution of a module, the module provides a list of changes that it wants made to the scene for that execution. When a **geometry viewer** module receives this list of changes, it goes through the list in order, applying the changes that it receives and redrawing the scene, if necessary.

The data type for communicating geometry from the geometry producing module to the **geometry viewer** module, therefore, is a list of changes that are to be made to the scene for that execution. We call this data type an "edit list" since it is a list of "edits" to be made to the scene. See the beginning the "AVS Module Interface Routines" section of this appendix for more information about edit lists and the routines use to modify them.

The remainder of this appendix is devoted to describing the Geometry Library routines that you can use (along with other AVS supplied routines) to implement geometry producing modules. Also, see Chapter 3 for more information on modules.

Description

The geom library provides functions for reading and writing objects, creating objects from a variety of different data formats, processing objects (such as generating normals), and "compiling" databases into a format that is most efficient for the hardware to render.

The geom library contains both a C and a FORTRAN version of each routine. The main routine descriptions in this manual page discuss the C versions. For a discussion of the FORTRAN calling sequences, see the "Fortran Binding" section near the end of this chapter.

Many routines allow a NULL value for some arguments. In all such cases, the constant **GEOM_NULL** should be used to represent a NULL value.

The basic entity in the geom package is the geom object. Several types of objects are supported:

GEOM_LABEL
GEOM_MESH
GEOM_POLYHEDRON
GEOM_POLYTRI
GEOM_SPHERE

Each object can have colors or normals associated with each vertex in the object, but it need not have either.

Label
A label contains one or more text strings normally used to label an AVS object or view. The label is presented as annotation text: its position can be transformed, but the text always appears upright in a plane parallel to the display surface.

Mesh
A mesh object contains one two-dimensional array of vertices.

Polyhedron
A polyhedron contains a list of vertices and a list of polygons, which are defined by indirectly referencing the vertices that the polygon contains.

Polytriangle
A polytriangle object contains a list of polytriangle strips, a list of polylines, or a list of disjoint lines. When a single object has both line and surface data, the line data is assumed to be an alternate wireframe description for the same geometry (only one should be displayed at a given time).

Sphere
A sphere object contains a list of points and a corresponding list of radii.

Object Creation Routines

Many routines can be used to create an object. The goal is to provide entry points that allow many different data formats to be simply converted to the internal data base format. Different routines are used by different filters.

The creation routines for the geom library define some simple data formats:

- A list of vertices is a 2D array of floats of X, Y, and Z.
- A list of normals is a 2D array of floats of NX, NY, and NZ.
- A list of float colors is a 2D array of floats of R, G, B (in the range of 0.0 to 1.0).
- A list of integer colors is also defined where R, G, and B are bytes packed into an integer using the shifts **AVS_RED_SHIFT**, **AVS_GREEN_SHIFT**, and **AVS_BLUE_SHIFT**, defined in */usr/avs/include/port.h*. All colors are stored internally as arrays of floats with values between 0 and 1.
- A list of floating point transparency values that are associated with each vertex. The values should be in the range of 0.0-1.0 with 0.0 being opaque and 1.0 being transparent. The transparency value is meant to be interpolated across the surface and then used as the transparency attribute for each individual rendered point. Many renderers ignore this type of data.
- A list of extents is a 1D array of 6 floats in the order: MIN X, MAX X, MIN Y, MAX Y, MIN Z, MAX Z.
- User-supplied primitive data. The user may associate a single integer with each primitive where a primitive is defined as a line or polygon.
- User-supplied vertex data. This is similar to user-supplied primitive data but is associated with a vertex instead of a primitive.

Creating an Object

A geom object is initially created without any data at all. Data is then added to the object incrementally. A typical sequence would be to create an object of type polyhedron, add a polygon list, add vertices, then add normals. Notice that an object can be in an intermediate state where it doesn't make sense — it can have a polygon list without vertices, for example.

To reduce the number of procedure calls required to create an object, the geom library also provides macro functions that create and add various pieces of data. When one of these calls has a parameter for an optional piece of data (*normals* and *colors*, for example), **GEOM_NULL** can be used to indicate that this object does not have this type of data.

Extents

Each object can have extent information associated with it. The extent of an object is determined by the minimum and maximum values of the coordi-

nates of the object's vertices. Routines that create objects take optional extent information. Passing GEOM_NULL for this parameter indicates no extent is being specified. This is usually the best way to specify the extents unless you have explicit knowledge of what the extents should be for the object.

If you do not supply extent information during the creation of your object, it is generated for you when and if it is needed by some other part of the system. It is generated by finding the minimum and maximum values of X, Y, and Z in your vertex list. For spheres, the radii is used to determine the extents.

In some situations, you might want to provide extent information that is not the same as the object's actual extents. For example, if you have a time series of data where the object's extents are expanding (an explosion, for example) you may want to set the extent for the whole series to be large enough to avoid clipping the scene as the extents required increase in dimension. This way you can normalize the object (center it in the view) and not loose portions of it as the time series progresses

You should not set the extent so that it is smaller than the geometry of your object, as the system relies on the extent to display all of the geometry.

Flags

Many routines have an *alloc* flag as a parameter. If this flag is set to **GEOM_COPY_DATA**, the geom routine allocates its own space and copies the data of the object. If this flag is set to **GEOM_DONT_COPY_DATA**, the geom routine does not copy the data. In this case, you must allocate space for the data using the C entry library routine **malloc**(3C). It is usually easier to allow the routine to allocate the required space.

User Supplied Primitive Data

You may associate a single integer with each line or polygon primitive within an object. This integer should contain no more than 4 bytes of significant information. Unlike other data values that are associated with the object, this data is not interpreted by the geometry viewer. It is returned to the user for pick correlation purposes using upstream data. See the section on upstream data in Chapter 4 for more information on how to use this within a module.

For polyhedrons, there can be a single value for each polygon in the object. For meshes, there can be "m-1" * "n-1" values (this is the number of quadrilaterals in the mesh). For triangle strips, there can be "n-2" values. For disjoint lines, there can be "n / 2" and for polylines there can be "n-1".

User Supplied Vertex Data

This is similar to user-supplied primitive data but is associated with a vertex instead of a primitive. During any particular pick, the user picks a primitive, but AVS returns the information corresponding to the closest selected vertex as well. For a particular object, there can be a single piece of user-supplied data for each vertex in the object.

Vertex and primitive data are not applicable to all object types. The following chart shows what object types can use each type of data:

Table G-1 Object Vertex and Primitive Data Applicability

Type	Primitive	Vertex
GEOM_POLYHEDRON	yes	yes
GEOM_POLYTRI	yes	yes
GEOM_SPHERE	no	yes
GEOM_LABEL	no	no
GEOM_MESH	yes	yes

GEOMadd_disjoint_line

GEOMadd_disjoint_line(*obj, verts, colors, n, alloc***)**
GEOMobj **obj;*
float **verts;*
float **colors;*
int *n;*
int *alloc;*

Adds an array of lines to an object of type **GEOM_POLYTRI**. It adds the vertices of this disjoint line to any existing disjoint lines in the object.

GEOMadd_disjoint_polygon

GEOMadd_disjoint_polygon(*obj, verts, normals, colors, nverts, flag, alloc***)**
GEOMobj **obj;*
float **verts, *normals, *colors;*
int *nverts;*
int *alloc;*
int *flag;*

Adds a disjoint polygon to a polyhedron object. The polygon has *nverts* vertices which are specified in the array *verts*. The *normals* and *colors* arguments can contain the *normals* and *colors* for the object, or they can be **GEOM_-NULL**. The *flag* argument contains two pieces of information: the nature of the polygon (whether it is convex, concave, or complex), and whether the vertices of the polygon should be shared with the other vertices in the object. Shared vertices create an object whose vertices approximate a smooth object

(such as a sphere); unshared vertices create an object that is faceted. The *flag*s **GEOM_SHARED** and **GEOM_NOT_SHARED** are used to determine whether the vertices are shared or not. The values **GEOM_CONCAVE**, **GEOM_CONVEX**, and **GEOM_COMPLEX** can be OR'd with the other value to produce the *flag* argument.

Specifying shared vertices causes this routine to try to determine whether any vertices in the object are already represented. Instead of adding a new vertex when an old identical vertex is found, it uses a reference to this vertex. This process can take considerable time when the number of vertices in the object is large, but it produces an object that is significantly more efficient to render, because the resulting object contains fewer vertices to transform, light, and shade.

GEOMadd_disjoint_prim_data

GEOMadd_disjoint_prim_data(*obj, pdata, n, alloc***)**
GEOMobj **obj;*
float **pdata;*
int *n;*
int *alloc;*

This routine should only be used for objects of type **GEOM_POLYTRI** that have disjoint line primitives in them. It allows the user to associate primitive data with the disjoint lines in a polytriangle type object. The number "n" should be the number of disjoint lines in the object— note that this is one-half the number of vertices that the object contains.

GEOMadd_disjoint_vertex_data

GEOMadd_disjoint_vertex_data(*obj, vdata, n, alloc***)**
GEOMobj **obj;*
int **vdata;*
int *n;*
int *alloc;*

This routine should only be used for objects of type **GEOM_POLYTRI** that have disjoint line primitives in them. It allows the user to associate vertex data with the disjoint lines in a polytriangle type object. The number "n" should be the number of vertices in the disjoint line object. Note that this is twice the number of disjoint lines.

GEOMadd_float_colors

GEOMadd_float_colors(*obj, colors, n, alloc***)**
GEOMobj **obj;*
float **colors;*

int *n;*
int *alloc;*

Adds a list of float colors to an object. The Red, Green, and Blue values are stored separately (i.e., it takes three floats to specify an RGB), and should range between 0 and 1. This routine cannot be used with objects of type **GEOM_POLYTRI**.

GEOMadd_int_colors

GEOMadd_int_colors(*obj, colors, n, alloc***)**
GEOMobj **obj;*
unsigned long **colors;*
int *n;*
int *alloc;*

Adds a list of integer colors to an object. The RGB values are packed into a single integer using the shifts **AVS_RED_SHIFT**, **AVS_GREEN_SHIFT**, and **AVS_BLUE_SHIFT**, defined in */usr/avs/include/port.h*i. This routine cannot be used with an object of type **GEOM_POLYTRI**.

GEOMadd_label

GEOMadd_label(*obj, text, ref_point, offset, height, color, label_flags***)**
GEOMobj **obj;*
char **text;*
float *ref_point[3];*
float *offset[3];*
float *height;*
float *color[3];*
int *label_flags;*

This routine adds a text string and related characteristics to an existing label object. Each label object can have more than one text string, along with related characteristics for each string. Each such string is added by a separate call to **GEOMadd_label** for the same label object. All data is copied (**GEOM_-COPY_DATA** is assumed). The arguments are as follows:

text
The text string for the label.

ref_point
offset
These arguments determine the positioning of the label. The reference point is an array of X, Y, and Z coordinates. For a label used as a window title, these are in screen space, with (-1, -1, -1) at the lower left rear corner and (1, 1, 1) at the upper right front corner, and are not transformed. For other labels the coordinates of the reference point are transformed. The

offset is an array of X, Y, and Z values in screen space. After the reference point is transformed (if necessary), the offset is applied to determine the final position of the reference point in screen space. The label always appears upright and in a plane parallel to the display surface.

height
The height of the label in screen space.

color
An RGB triple specifying the text color, or **GEOM_NULL** to indicate that the foreground color (usually white) should be used. (When the text background rectangle is drawn, the window background color is used for the rectangle.)

label_flags
An integer returned by a call to **GEOMcreate_label_flags**, or a value of -1 to indicate that the default label flags in the label object, added by the call to **GEOMcreate_label**, should be used. For a given label object, either all text strings must use the default label flags, or no text strings can use the default label flags. That is, either all calls to **GEOMadd_label** must pass -1 for the *label_flags* argument, or no calls to **GEOMadd_label** can pass -1 for the *label_flags* argument.

GEOMadd_normals

GEOMadd_normals(*obj, normals, n, alloc***)**
GEOMobj **obj;*
float **normals;*
int *n;*
int *alloc;*

Adds a list of *normals* to an object. This routine cannot be used with objects of type **GEOM_POLYTRI** or **GEOM_SPHERE**.

GEOMadd_polygon

GEOMadd_polygon(*obj, nverts, indices, flags, alloc***)**
GEOMobj **obj;*
int *nverts;*
int **indices;*
int *flags;*
int *alloc;*

Adds a polygon to a polyhedron object. The *indices* argument specifies an array of *nverts* integers, where each integer is an index into a vertex array that is added with the **GEOMadd_vertices** call either before or after this call is made. If multiple calls to **GEOMadd_vertices** are made, the first vertex added always remains the first vertex in the list. The indices in this array are "1

based". The first vertex in the list is 1, not 0. The *flags* argument can be either **GEOM_CONCAVE** or **GEOM_CONVEX**.

GEOMadd_polygons

GEOMadd_polygons(*obj, plist, flags, alloc***)**
register GEOMobj **obj;*
register int **plist;*
int *flags;*
int *alloc;*

Adds a polygon list to a polyhedron object. The polygon list is an array of **ints** where the first **int** (*plist*[0]) indicates the number of vertices in the first polygon. The number of vertices (*plist*[0]) is followed by that number of indices into the vertex list. The second polygon's vertex list immediately follows the first. The list is terminated with a 0 number of vertices after the last polygon's vertex list. As with the **GEOMadd_polygon** routine, the vertex indices are "1 based". The first vertex in the list is 1, not 0. The flags argument can be either **GEOM_CONCAVE** or **GEOM_CONVEX**.

GEOMadd_polyline

GEOMadd_polyline(*obj, verts, colors, n, alloc***)**
GEOMobj **obj;*
float **verts;*
float **colors;*
int *n;*
int *alloc;*

Adds a polyline to an object of type **GEOM_POLYTRI**. The colors argument can be **GEOM_NULL**.

GEOMadd_polyline_prim_data

GEOMadd_polyline_prim_data(*obj, pdata, i, n, alloc***)**
GEOMobj **obj;*
int **pdata;*
int *i;*
int *n;*
int *alloc;*

This routine should be used only for objects of type **GEOM_POLYTRI** that contain polyline primitives. It allows the user to associate vertex data with the polylines in a polytriangle type object. The number *n* is the number of vertices in the polyline object. Note that this is the number of disjoint lines - 1. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, etc.

GEOMadd_polyline_vertex_data

GEOMadd_polyline_vertex_data(*obj, vdata, i, n, alloc***)**
GEOMobj **obj;*
int **vdata;*
int *i;*
int *n;*
int *alloc;*

This routine should be used only for objects of type **GEOM_POLYTRI** that contain polyline primitives. It allows the user to associate vertex data with the polylines in a polytriangle type object. The number *n* is the number of vertices in the polyline object. Note that there are *n* vertices, but the number of lines is *n*-1. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, etc.

GEOMadd_polytriangle

GEOMadd_polytriangle(*obj, verts, normals, colors, n, alloc***)**
GEOMobj **obj;*
float **verts;*
float **normals;*
float **colors;*
int *n;*
int *alloc;*

Adds a polytriangle to the object. Note that *colors* is an array of **float** *colors*, not **int** *colors*. An object can contain more than one polytriangle strip.

GEOMadd_polytriangle_prim_data

GEOMadd_polytriangle_prim_data(*obj, pdata, i, n, alloc***)**
GEOMobj **obj;*
int **pdata;*
int *i;*
int *n;*
int *alloc;*

This routine should be used only for objects of type **GEOM_POLYTRI** that contain polytriangle strip primitives. It allows the user to associate vertex data with the polytriangle strips in a polytriangle type object. The number *n* is the number of triangles in the polytriangle object. Note that this is the number of vertices - 2. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, etc.

GEOMadd_polytriangle_vertex_data

GEOMadd_polytriangle_vertex_data(*obj, vdata, i, n, alloc***)**
GEOMobj **obj;*
int **vdata;*
int *i;*
int *n;*
int *alloc;*

This routine should be used only for objects of type **GEOM_POLYTRI** that contain polytriangle strip primitives. It allows the user to associate vertex data with the polytriangle strips in a polytriangle type object. The number *n* is the number of triangles in the polytriangle object. Note that this is the number of vertices - 2. The value of *i* specifies the particular primitive within the object, with which you want to associate the data. The first primitive is 0, the second is 1, etc.

GEOMadd_prim_data

GEOMadd_prim_data(*obj, pdata, n, alloc***)**
GEOMobj **obj;*
int **pdata;*
int *n;*
int *alloc;*

Associate the array of primitive data with the object specified. This routine can be used only for objects of type **GEOM_POLYHEDRON** and **GEOM_-MESH**. For objects of type **GEOM_MESH**, there value *n* should be equal to: (m-1) * (n-1). Where "m" and "n" are the dimensions of the mesh.

GEOMadd_radii

GEOMadd_radii(*obj, radii, n, alloc***)**
GEOMobj **obj;*
float **radii;*
int *n;*
int *alloc;*

This routine adds the radii supplied to an object of type **GEOM_SPHERE**. The number *n* contains the number of spheres in the object. The *alloc* parameter is **GEOM_DONT_COPY_DATA** if the data has been allocated using the **malloc**(3C) routine, and has not been freed by the application. It should be **GEOM_COPY_DATA** otherwise.

GEOMadd_vertex_data

GEOMadd_vertex_data(*obj, vdata, n, alloc***)**
GEOMobj **obj;*
int **vdata;*
int *n;*
int *alloc;*

Associates the array of vertex data with the object specified. This routine can be used only with objects of type: **GEOM_MESH, GEOM_POLYHEDRON**, and **GEOM_SPHERE**. The number of data elements *n*, should be equal to the number of vertices in the object.

GEOMadd_vertices

GEOMadd_vertices(*obj, verts, n, alloc***)**
GEOMobj **obj;*
float **verts;*
int *n;*
int *alloc;*

Adds a list of vertices to an object. This routine should not be used for objects of type polytriangle (use **GEOMadd_polytriangle** instead).

GEOMadd_vertices_with_data

GEOMadd_vertices_with_data(*obj, verts, normals, colors, n, alloc***)**
GEOMobj **obj;*
float **verts;*
float **normals;*
unsigned int *colors;*
int *n;*
int *alloc;*

Adds *vertices*, *colors*, and *normal*s to the object. It assumes **integer** *colors*. Both the *normals* and *colors* parameters can be **GEOM_NULL**. This is a macro function combining the **GEOMadd_vertices, GEOMadd_normals,** and **GEOMadd_int_colors** routines.

GEOMcreate_label

GEOMobj *
GEOMcreate_label(*extent, label_flags***)**
float **extent;*
int *label_flags;*

This routine creates a label object. Each label object can have more than one text string, along with related characteristics for each string. Each such string is added by a separate call to **GEOMadd_label** for the same label object. The *label_flags* argument to **GEOMcreate_label** is normally an integer returned by a call to **GEOMcreate_label_flags**. It specifies default characteristics for all text strings in the label object. Either all text strings must use the default label flags, or no text strings can use them; see **GEOMadd_label** for more information. The *extent* argument can be **GEOM_NULL** if no extent is known.

GEOMcreate_label_flags

int
GEOMcreate_label_flags(*font_number, title, background, drop, align, stroke***)**
int *font_number, title, background, drop, align, stroke;*

This routine creates and returns a bit mask that is used to represent some characteristics of a label. The label flags are added by a call to **GEOMcreate_label** or to **GEOMadd_label**. To add a text string and related characteristics to the label, use the **GEOMadd_label** routine. The arguments are as follows:

font_number
An integer from 0 through 21 that specifies the font for the label's text string. The **GEOMget_font_number** call will generate a number to use here (which may differ from platform to platform) based upon a text string font name such as "Helvetica" or "Times", together with two booleans that flag bold or italic.

title
A value of 1 means that the label is to be used as a title for the window. The label is drawn in an absolute position with respect to screen space, which is defined so that (-1, -1, -1) is the lower left rear corner and (1, 1, 1) is the upper right front corner. A value of 0 means that the reference point of the label is transformed before the label is drawn. See the documentation for the **GEOMadd_label** routine for more information.

background
A value of 1 means that both the foreground text and the background rectangle that encloses the text are drawn. A value of 0 means that only the foreground text is drawn.

drop
A value of 1 means that a one-pixel drop-shadow highlight is added to the text. This makes the text stand out against a background of similar color. A value of 0 means that no highlight is added.

align
Specifies the position of the reference point within the label and therefore the alignment of the label. A value of **GEOM_LABEL_LEFT** places the reference point at the lower left corner of the label. A value of **GEOM_LABEL_CENTER** places the reference point at the bottom center of the la-

bel. A value of **GEOM_LABEL_RIGHT** places the reference point at the lower right corner of the label.

stroke
Not implemented; the value should be 0.

GEOMcreate_mesh

GEOMobj *
GEOMcreate_mesh(*extent, verts, m, n, alloc***)**
float **extent;*
float **verts;*
int *m, n;*
int *alloc;*

Creates a mesh from a 2D array of vertices. The dimensions of the array are specified by the *m* and *n* parameters. The first *n* vertices constitute the first row of the mesh. There are *m* rows of vertices. The extent parameter can be **GEOM_NULL** if no extent is known.

GEOMcreate_mesh_with_data

GEOMobj *
GEOMcreate_mesh_with_data(*extent, verts, normals, colors, m, n, alloc***)**
float **extent;*
float **verts;*
float **normals;*
unsigned long **colors;*
int *m, n;*
int *alloc;*

This routine is a macro function combining the **GEOMcreate_mesh**, **GEOMadd_int_colors**, and **GEOMadd_normals** routines.

GEOMcreate_obj

GEOMobj *
GEOMcreate_obj(*type, extent***)**
int *type;*
float **extent;*

Type should be one of **GEOM_LABEL**, **GEOM_MESH**, **GEOM_POLYHEDRON**, **GEOM_POLYTRI** or **GEOM_SPHERE**. Extent can be either the extent of the object or **GEOM_NULL** if no extent is known. This routine creates an object of the specified type. Initially the object has no data.

GEOMcreate_polyh

GEOMobj *
GEOMcreate_polyh(*extent, verts, n, plist, flags, alloc***)**
float **extent;*
float **verts;*
int *n;*
int **plist;*
int *flags;*
int *alloc;*

This routine is a macro function combining the **GEOMcreate_obj**, **GEOMadd_vertices**, and **GEOMadd_polygons** routines. The *flags* argument can be either **GEOM_CONCAVE** or **GEOM_CONVEX**.

GEOMcreate_polyh_with_data

GEOMobj *
GEOMcreate_polyh_with_data(*extent, verts, normals, colors, n, plist, flags, alloc***)**
float **extent;*
float **verts;*
float **normals;*
unsigned long **colors;*
int *n;*
int **plist;*
int *flags;*
int *alloc;*

This routine is a macro function combining the **GEOMcreate_polyh**, **GEOMadd_int_colors**, and **GEOMadd_normals** routines. The *flags* argument can be either **GEOM_CONCAVE** or **GEOM_CONVEX**

GEOMcreate_scalar_mesh

GEOMobj *
GEOMcreate_scalar_mesh(*xmin, xmax, ymin, ymax, mesh, colors, n, m, alloc***)**
float *xmin, xmax, ymin, ymax*
float **mesh, *colors;* /* Colors is R,G,B */
int *n, m;*

Creates a mesh from a single array of scalar values (a height field). The scalars are taken to be the Z component of the object. X will be evenly spaced between *xmin* and *xmax*, and Y will be evenly spaced between *ymin* and *ymax*. The colors parameter can be **GEOM_NULL**.

GEOMcreate_sphere

GEOMobj *
GEOMcreate_sphere(*extent, verts, radii, normals, colors, n, alloc***)**
float **extent;*
float **verts;*
float **radii;*
float **normals;*
unsigned long **colors;*
int *n, alloc;*

Creates a sphere object. The vertices (*vert*) argument specifies the sphere centers. The *normals* and *colors* arguments can be **GEOM_NULL**. When the value of *radii* is **GEOM_NULL,** the spheres will be rendered as dots. The *normals* are generally not used. To create a sphere with **float** *colors*, use the **GEOMadd_float_colors** routine after the sphere is created.

GEOMdestroy_obj

GEOMdestroy_obj(*obj***)**
GEOMobj **obj;*

Frees all memory associated with the object, including memory given to a "create" call with the flag **GEOM_DONT_COPY_DATA**.

GEOMget_font_number

int
GEOMget_font_number(*name, bold, italic***)**
char **name;*
int *bold, italic;*

This routine is used to convert a font name, bold and italic values, and produce a font number that is used as the first argument to the routine **GEOMcreate_label_flags**. The label flags are then used to define the font, alignment, and other drawing attributes for the label primitive.

This routine is given a font name which can be one of: "Courier", "Helvetica", "New Century", "Times", "Charter", "Symbol", "Roman", "Script", or "Mathematics". It also takes two boolean values to indicate whether the font should be drawn with bold or italic style. The return of this routine is the font number to use in the routine **GEOMcreate_label_flags**.

Not all fonts and styles are implemented on all renderers. If a font is not implemented by a particular renderer, it will be simulated with the closest approximating font that is available.

Object Utility Routines

Once a geom object has been created, the utility routines can be used. These routines control: object normals (the generation of proper normals is critical to an accurate and illustrative description of geometry); object autotransformations (the transformation of individual objects and groups of objects to fit within a unit cube (-1 to 1 in X, Y, and Z)—this is distinct from the general transfomation of objects within world space using edit list calls; and object format conversions).

This last is necessary because on some architectures, the most efficient object format is the polytriangle for nonsphere surface descriptions and the polyline for wireframe descriptions. Two utility routines convert data to the proper type. The conversion routines convert to either polytriangles, polylines, or both, depending on the setting of the *flags*. Sphere primitives should not be converted. This conversion should be performed after normals have been generated for the object (if normals are desired) as the conversion results in a loss of vertex coherence information.

GEOMauto_transform

GEOMauto_transform(*obj***)**
register GEOMobj **obj;*

GEOMauto_transform_non_uniform

GEOMauto_transform_non_uniform(*obj***)**
register GEOMobj **obj;*

Transforms the object specified to lie within the cube from -1 to 1 in X, Y, and Z. The scaling and translation factors are uniform for **GEOMauto_transform** and nonuniform for **GEOMauto_transform_non_uniform**.

GEOMauto_transform_list

GEOMauto_transform_list(*objs, n***)**
register GEOMobj ***objs;*
register int *n;*

GEOMauto_transform_non_uniform_list

GEOMauto_transform_non_uniform_list(*objs, n***)**
register GEOMobj ***objs;*
register int *n;*

Transforms the list of objects specified to lie within the cube from -1 to 1 in X, Y, and Z. First the bounding box of all objects in the list is generated, then scaling and translation factors are computed to transform this box to lie inside the cube from -1 to 1 in X, Y, and Z. The scaling and translation factors are uniform for **GEOMauto_transform_list** and nonuniform for **GEOMauto_transform_non_uniform_list**. The relative sizes of objects in the list are not affected.

GEOMcreate_normal_object

GEOMobj *
GEOMcreate_normal_object(*obj,scale***)**
GEOMobj **obj;*
float *scale;*

This routine takes an object that has normal data and returns an object that consists of disjoint lines that represent the normals of that object. The normals will be of length *scale* times their current length.

GEOMcvt_mesh_to_polytri

GEOMcvt_mesh_to_polytri(*tobj, flags***)**
GEOMobj **tobj;*
int *flags;*

Creates a polytriangle or polyline description of the mesh object given. The resulting object contains one large polytriangle strip (if the *flags* argument is **GEOM_SURFACE**) and *n* * *m* polylines (if the *flags* argument is **GEOM_-WIREFRAME**).

GEOMcvt_polyh_to_polytri

GEOMcvt_polyh_to_polytri(*tobj, flags***)**
GEOMobj **tobj;*
int *flags;*

Uses a graph traversing algorithm to generate either a surface or a wireframe description of a polyhedron object, depending on the *flags* argument. It attempts to share as many vertices as possible. The surface algorithm can take a reasonably long time to complete for very large objects. The wireframe algorithm creates polylines for large connected strips and disjoint lines for smaller ones. The *flags* argument can be **GEOM_SURFACE**, **GEOM_WIREFRAME**, or **GEOM_EXHAUSTIVE**. The **GEOM_EXHAUSTIVE** flag should be used only in dire circumstances.

GEOMflip_normals

GEOMflip_normals(*obj***)**
GEOMobj **obj;*

This routine inverts the direction of the normals in the object given.

GEOMgen_normals

GEOMgen_normals(*obj, flags***)**
GEOMobj **obj;*
int *flags;* /* 0 or GEOM_FACET_NORMALS */

Generates surface normals for **GEOM_MESH** or **GEOM_POLYHEDRON** objects. By default, it assumes that the object is an approximation of a smooth surface. If the flags field is **GEOM_FACET_NORMALS** and the object is a polyhedron, a separate normal is generated for each facet (polygon). A copy of the facet normal is associated with each of the facet verticies. A vertex that is shared by multiple facets is replicated for each facet. This replication decreases the rendering performance of the object. Normals generated by this routine are guaranteed to be of unit length.

GEOMnormalize_normals

GEOMnormalize_normals(*obj***)**
GEOMobj **obj;*

Normalizes (converts to unit length) the normals of the object specified. Normals are normalized automatically by the **GEOMgen_normals** routine.

GEOMset_computed_extent

GEOMset_computed_extent(*obj, extent***)**
GEOMobj **obj;*
float *extent[6];*

Sets the extent of the object to the *extent* passed in. The *extent* passed in should contain in order: *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*. This can be used in conjunction with either **auto_transform** routine to perform arbitrary scaling and translation of objects.

GEOMset_extent

GEOMset_extent(*obj***)**
GEOMobj **obj;*

Sets the extent of the object given. Currently, it is not implemented properly for objects of type **GEOM_SPHERE**.

GEOMset_object_group

GEOMset_object_group(*obj, name***)**
GEOMobj **obj;*
char **name;*

This routine is used when storing multiple AVS objects (groups of geom objects) in a single *geom* file. By default, when AVS reads a *geom* file it places all geom objects in that file into a single AVS object. The **read_subset** script language command can read a subset of the geom objects in a *geom* file and place only those geom objects into an AVS object. Each geom object to be placed into the same AVS object must have the same group name, which is added by the **GEOMset_object_group** routine. The **read_subset** command takes a group name as an argument and places all geom objects in the *geom* file that have that group name into a single AVS object. The **read_subset** command ignores all geom objects in the *geom* file that do not have that group name.

GEOMset_pickable

GEOMset_pickable(*obj, pickable***)**
GEOMobj **obj;*
unsigned long *pickable;*

This routine sets the pickable state of an object. If multiple objects are placed in a *geom* file, by default they are not pickable individually. If this attribute is set to "1", they can be picked individually when running the AVS viewing application.

GEOMunion_extents

GEOMunion_extents(*obj1, obj2***)**
GEOMobj **obj1, *obj2;*

Sets the extent of *obj1* to include the extent of *obj2*. It generates the extents of both objects if they aren't set already.

Object Property Routines

A feature of the geom library allows arbitrary value lists to be associated with each object. These value lists can then be interpreted by packages reading in the objects. The object format supports values that are arbitrarily long. Currently only integer values are supported by the subroutine interface.

GEOMadd_int_value

GEOMadd_int_value(*obj, type, value***)**
GEOMobj **obj;*
int *type;*
int *value;*

Adds an integer property to an object. Currently the only fully supported property of an object is the color (type **GEOM_COLOR**).

GEOMquery_int_value

int
GEOMquery_int_value(*obj, type, value***)**
GEOMobj **obj;*
int *type;*
int **value;*

An integer value can be queried with this routine. The type is an integer value. The only fully supported property type is **GEOM_COLOR**. Its value can be queried with:

GEOMquery_int_value(*obj, GEOM_COLOR, &color***);**

This routine returns 0 if no color was associated with the object.

GEOMset_color

GEOMset_color(*obj, color***)**
GEOMobj **obj;*
unsigned long *color;*

Sets the *color* property of an object (by calling the **GEOMadd_int_value** routine).

Object Texture Mapping Routines

Each surface object can have *uv* data associated with each vertex. The *uv* data consists of two floating point values per vertex, which specify a mapping into a texture map. The value of *u*=0, *v*=0 is the index into the upper left hand corner of the texture map; *u*=1, *v*=1 is the lower right hand corner. These values are stored in memory as an array of floating point values.

GEOMadd_polytriangle_uvs

GEOMadd_polytriangle_uvs(*obj, uvs, i, n, alloc***)**
GEOMobj **obj;*
float **uvs;*
int *i;*
int *n;*
int *alloc;*

Each polytriangle object has an array of polytriangle strips. This routine is used to add *uv* data to a polytriangle object. These polytriangle strips are kept in an array in the order in which they were added: the first polytriangle is index 0, the second is index 1, etc. This routine adds *uv* data for a single polytriangle strip. The index of the polytriangle strip is the variable *i*. This polytriangle strip should have *n* vertices. The *alloc* parameter is **GEOM_DONT_COPY_DATA** if the data has been allocated using the **malloc**(3C) routine, and has not been freed by the application. It should be **GEOM_COPY_DATA** otherwise.

GEOMadd_uvs

GEOMadd_uvs(*obj, uvs, n, alloc***)**
GEOMobj **obj;*
float **uvs;*
int *n;*
int *alloc;*

This routine adds the *uv* data to an object of type **GEOM_POLYHEDRON**, or **GEOM_MESH**. *n* should specify the number of vertices in the object. The *alloc* parameter is **GEOM_DONT_COPY_DATA** if the data has been allocated using the **malloc**(3C) routine, and has not been freed by the application. It should be **GEOM_COPY_DATA** otherwise.

GEOMcreate_mesh_uvs

GEOMcreate_mesh_uvs(*obj, umin, vmin, umax, vmax***)**
GEOMobj **obj;*
double *umin, vmin, umax, vmax;*

This routine creates *uv* data for a mesh object such that the 0,0 vertex in the mesh will have *u*=0, *v*=0, and the *n,m* vertex in the mesh will have *u*=1, *v*=1. It is an error to use this routine with an object that is not of type **GEOM_MESH**.

GEOMdestroy_uvs

GEOMdestroy_uvs(*obj***)**
GEOMobj **obj;*

This routine takes an object that has *uv* data for each vertex and turns it into an object that doesn't have *uv* data for each vertex.

Object Vertex Transparency Routines

GEOMadd_vertex_trans

GEOMadd_vertex_trans(*name, vtrans, n, alloc*)
GEOMobj **name;*
float **vtrans;*
int *n, alloc;*

This routine adds vertex transparency values to a geometry object of type **GEOM_MESH** and **GEOM_POLYHEDRON**. This routine should not be used with objects of type: **GEOM_SPHERE**, **GEOM_LABEL** and **GEOM_-POLYTRI**. Vertex transparency values, specified by the argument *vtrans* should be floating point values in the range 0.0-1.0. The value 0.0 will produce a completely opaque surface and the value 1.0 will produce an transparent object. The parameter *n* corresponds to the number of floating point values supplied with this call. This number should correspond to the number of vertices in the object. The *alloc* flag can be either the value **GEOM_COPY_DATA**, or the value **GEOM_DONT_COPY_DATA**. The flag **GEOM_DONT_-COPY_DATA** should only be used by C applications that have allocated the data using the *malloc* utility. If **GEOM_DONT_COPY_DATA** is used, the user should not free the data This will be taken care of by GEOM when the object itself is destroyed. If this is not the case, the user should use the flag **GEOM_-COPY_DATA** and is then responsible for the maintenance of this storage.

GEOMadd_polytriangle_vertex_trans

GEOMadd_polytriangle_vertex_trans(*name, vtrans, i, n, alloc*)
GEOMobj **name;*
float **vtrans;*
int *i, n, alloc;*

This routine adds vertex transparency values to a geometry object of type **GEOM_POLYTRI**. This routine should not be used with objects of type: **GEOM_MESH**, **GEOM_POLYHEDRON**, **GEOM_SPHERE**, and **GEOM_-LABEL**.

Vertex transparency values, specified by the argument *vtrans* should be floating point values in the range 0.0-1.0. The value 0.0 will produce a completely opaque surface and the range 1.0 will produce an transparent object. The parameter *n* corresponds to the number of floating point values supplied with this call which should equal the number of vertices contained in the individual polytriangle strip.

The parameter *i* specifies which polytriangle these vertex colors should be associated with. Polytriangles are numbered in the order in which they are added to the polytriangle object with 0 being the first triangle strip. The *alloc* flag can be either the value **GEOM_COPY_DATA**, or the value **GEOM_DONT_COPY_DATA**. The flag **GEOM_DONT_COPY_DATA** should only be used by C applications that have allocated the data using the *malloc* utility. If **GEOM_DONT_COPY_DATA** is used, the user should not free the data. This will be taken care of by GEOM when the object itself is destroyed. If this is not the case, the user should use the flag **GEOM_COPY_DATA** and is then responsible for the maintenance of this storage.

Object File Utilities

Prior to writing data to files in most implementations, each platform determines which format (e.g., **GEOM_POLYTRI** or **GEOM_POLYHEDRON)** is most efficient for that platform and automatically converts the data to this format.

GEOMread_obj

GEOMobj *
GEOMread_obj(*fd, flags***)**
int *fd;*
int *flags;*

Performs a read operation on the file descriptor given and interprets the data it finds as a geom object. Data can be stripped off by specifying the **GEOM_NORMALS** flag (to strip off the normals), the **GEOM_VCOLORS** flag (to strip of the colors), or the OR of these values (to strip off normals and colors). A *flags* value of 0 means leave the data intact.

GEOMread_text

GEOMobj *
GEOMread_text(*fp, flags*)
FILE **fp;*
int *flags;*

This routine reads the text GEOM file from the file pointer *fp*. Normally this routine is used on a file that was previously output from the routine **GEOM-**

write_text. The *flags* argument can be used to strip off the normals or colors in the object by supplying the **GEOM_NORMALS** or **GEOM_VCOLORS** arguments. Normally this parameter is 0.

GEOMwrite_obj

GEOMwrite_obj(*obj, fd, flags***)**
GEOMobj **obj;*
int *fd;*
int *flags;*

Writes a geom object to a file. The *fd* parameter is a file descriptor representing the file or device to write to. Data can be stripped off by specifying the **GEOM_NORMALS** flag (to strip off the normals), the **GEOM_VCOLORS** flag (to strip of the colors), or the OR of these values (to strip off normals and colors). A flags value of 0 means leave the data intact.

GEOMwrite_text

GEOMwrite_text(*obj, fp, flags***)**
GEOMobj **obj;*
FILE **fp;*
int *flags;*

This routine writes an ASCII version of the geom object specified to the stream *fp*. This routine is implemented for all geom object types and is useful for debugging or transporting geom data to different architectures.

Object Debugging Facilities

GEOMcheck_obj

GEOMcheck_obj(*name, flags, func*)
GEOMobj **name;*
int *flags;*
int *(*func)();*

This routine verifies that the geom object that you supply is a valid geometry object. It ensures that your object contains:

- zero or a positive number of vertices.
- the same number of vertices, normals, colors etc.
- no "not-a-number" floating point values as produced in some undefined floating point operations.
- color values that are within the 0.0-1.0 range.

- extent minimum is less than extent maximum.
- polyhedron objects have no indices that are out of range of the number of vertices.

When this routine encounters an error, it will supply a reasonably detailed message indicating the nature of the error. If the func argument is passed in with a NULL value, this message will be sent to "standard error" of the calling process. Otherwise, the function pointer func will be called with a NULL terminated error message as the first argument:

```
(*func)(error_message)
char *error_message;
```

The *flags* argument is currently not used and should be set to 0.

This routine will be called automatically every time that each geometry object is either read from or written to a file, or passed as an input or output from a module when the environment variable **AVS_GEOM_VERIFY** is set in the shell before either AVS or the geometry filter is executed. You can do this as follows in the C-shell:

```
setenv AVS_GEOM_VERIFY
./avs
```

Or in the Bourne shell:

```
AVS_GEOM_VERIFY=1
export AVS_GEOM_VERIFY
```

AVS Module Interface Routines

Module interface routines allow you to create, modify, and destroy edit lists.

Edit Lists

The data type used by AVS modules that handle geometries is an *edit list*. The AVS data type for an edit list is **GEOMedit_list**. An *edit list* is an arbitrarily long list of changes to be applied to a scene. Each change pertains to a particular object of type GEOMobj or to a light source. Changes are made in the order specified in the edit list.

AVS allows a user module to create *edit lists* as outputs; AVS does not support using them as inputs. A C language module computation routine declares an argument representing an input port or parameter as **GEOMedit_list** and an argument representing an output port as **GEOMedit_list *** (note the single asterisk).

Geometry output is typically used as input to a geometry renderer module such as the Geometry Viewer.

Each object or light is referred to by a name which is an ASCII string. Any object that doesn't already exist is created the first time an attempt to change that particular object is made. By default, an object name is modified by the port through which it is communicated. This prevents two different modules from modifying each other's objects. For example, two "plate" modules would each try to modify the data for the object named "plate". Since the name is modified by the port, the first plate module modifies *plate.0*, and the second modifies *plate.1*. When it is desirable for a module to use the absolute name of an object, it can precede the object name by a % character (e.g., "%plate").

AVS has routines that allow a module to change several properties of an object in an edit list:

- The geometric data defining the object
- Surface or line color
- Render mode (Gouraud, Phong, wireframe, etc.)
- Parent (the name of the parent object)
- Texture mapping
- Material properties
- Transformation

The name of each light source in an edit list is a string of the form "light*n*", where *n* is an integer from 1 through 16.

Certain edit list commands take the name of an object or camera / view as an argument. This camera name can be one of two forms either: "camera*n*" where *n* is an integer ranging between 1 and the number of cameras defined for the current scene, or the camera name can refer to the *title* of the particular camera.

The value *n* in the "camera*n*" scheme is defined by the order in which the cameras are created. Using *n* = 1 refers to first camera created for the scene. Note how this way of naming cameras changes when a camera is deleted from the scene.

The camera *title* refers to a particular camera. The name of a particular camera will not change when another camera is deleted. The title of a camera can be specified using the Geometry Viewer CLI when the camera is created. Otherwise, the camera name follows the scheme: "Camera *m*" where *m* is an integer, starting at 0, that increases each time that a new camera is created. The title index *m* is different from the previous naming index *n* only when a camera is deleted.

Each time a module is invoked, it should start with an empty edit list. It places into the edit list changes that it wants to be made for this invocation. In creating and using edit lists, geometry objects, and light sources, a module uses routines in the geom library. A module typically uses the following steps in preparing an edit list for output:

- Initialize the edit list, using **GEOMinit_edit_list**. This creates a new list or empties an existing list.
- Create and modify geometry objects or lights sources, using routines in the geom library.
- Modify the edit list, using routines whose names begin with **GEOMedit** in C (such as **GEOMedit_geometry**).

Coroutine module should use **AVScorout_output** to output the list.

A module must deallocate an existing edit list before reusing the list. For a subroutine module, the edit list passed to the module as an output argument is the edit list the module created on its last execution. The module must deallocate this list at the start of each invocation of the module, normally by calling the **GEOMinit_edit_list** routine before modifying the list. Coroutine modules can use **GEOMinit_edit_list** to deallocate/initialize a list after calling **AVScorout_output**.

There is also a **GEOMdestroy_edit_list** call. **GEOMdestroy_edit_list** destroys the edit list but does not create a new one. Once called, there is no longer a valid edit list to use the various **GEOMedit...** calls with. Hence, you can not call both **GEOMdestroy_edit_list** and **GEOMinit_edit_list** with the same pointer. You should only use **GEOMdestroy_edit_list** from within a coroutine if you know that you will not be doing any more geometry outputs.

Object Transformations

Some modules writers require detailed knowledge of how the transformation matrices of objects is maintained. This section describes this in detail and assumes significant knowledge of how computer graphics transformations are traditionally done.

Each object as a 4x4 homogenous transformation matrix and a 3x1 position vector that describes the current position and orientation of the object. The 4x4 matrix is treated in C as a 4x4 array of floats: e.g. float matrix[4][4]; and in FORTRAN as: REAL*4 matrix(4,4). In C, the translation component of this matrix is: X = matrix[3][0], Y = matrix[3][1], Z = matrix[3][2] and in FORTRAN it is: X = matrix(1,4), Y = matrix(2,4), Z = matrix(3,4).

At the point at which the matrix is applied to the object, the 3x1 position vector is added onto the matrix. It is kept separate so that we can define a fixed object center. The first transformation that we apply to the 4x4 matrix is a translate of the center of the object to the origin. After all of the rotations and scales we then apply a translation from the object center back to the origin. Then we tack on to the end the translation to the position of the object.

The vertex transformation can be depicted as follows:

Verts * [Trans(-center)] * [Rotates + Scales] * [Trans(center)] + Position

In general, the module writer does not have to be aware of this level detail. Note that if you use the routine **GEOMedit_set_matrix** for an object, though, that you are replacing the transformations that define the object center and therefore negate any center that you might have set.

Each child object is transformed by the complete matrix of its parent object. This occurs for each object all the way up to the top-level object. After its transformation has been applied, we are now in the "world coordinate" system. The world coordinate system is where light sources are defined.

Light Transformations

Light sources have a simpler transformation scheme than objects. They currently do not have a center of rotation, just a 4x4 transformation matrix and a position. The resulting position of light sources is determined by applying the resulting complete transformation of the light by the default location of the light source.

This is handled slightly differently for each different type of light source:

- ambient lights are not transformed at all
- directional lights are transformed as a direction vector with a homogenous coordinate of zero. Effectively this means that translations are ignored. The default direction vector is pointing into the scene: (0, 0, -1)
- point light sources are by default placed at: (0, 0, 1). This point is transformed regularly by the transformation matrix of the light.
- spot light sources are by default such that the position of the light source is at (0, 0, 0). This position is transformed regularly by the transformation matrix. The spot light also has a direction vector which has a default of (0, 0, -1) and this is transformed as directional light sources are.

Camera Transformations

The camera position is defined by a single 4x4 matrix and a 3x1 position vector that defines the cameras orientation and position and a separate 4x4 matrix that defines the projection for the camera matrix.

The resulting viewing pipeline is depicted as follows:

(Verts * [Obj Matrices] * [View Orientation] + Position) * [Projection]

The main utility for keeping the projection in a separate matrix is because it prevents us from applying any transformations after the perspective transformation.

In the Geometry Viewer, when you turn on and off the **Perspective** and **Front/ Back Clipping** buttons, you are modifying the default projection matrix. When you scale or rotate the camera, you are post-concatenating onto the

"View Orientation" matrix. Using the GEOM routine **GEOMedit_set_matrix** with a camera sets the "View Orientation" matrix. Using the GEOM routine edit list Routines

GEOMdestroy_edit_list

GEOMdestroy_edit_list(*list***)**
GEOMedit_list *list;*

Destroys an existing edit list.

GEOMedit_backface

GEOMedit_backface(*list*, *name*, *mode*)
GEOMedit_list *list;*
char **name;*
int *mode;*

This routine can be used to change the named object's backface properties. Backface properties determine how polygons that are facing away from the viewer are drawn. Vertices that are oriented in a clockwise fashion are back-facing. This is equivalent to the "right-hand" rule of orienting polygons.

Possible values for the *mode* argument are: **GEOM_BACKFACE_NORMAL**, **GEOM_BACKFACE_CULL_BACK**, **GEOM_BACKFACE_CULL_FRONT**, **GEOM_BACKFACE_FLIP**, **GEOM_BACKFACE_INHERIT**.

GEOM_BACKFACE_NORMAL
This mode causes the object's backfaces to be drawn in the normal back-face mode for the specific renderer. Some renderers "flip the normals" when a polygon is backfacing so that the backside of the polygon is lit in the same way as the front face. Other renderers light the backface of the polygon with the ambient intensity as the normal rendering mode.

GEOM_BACKFACE_FLIP
Some renderers support both the mode where backfaces are lit and the mode where backfaces are colored with only the ambient intensity. If this is the case, the normal mode will be where backfaces are lit with ambient intensity. In this case, the **GEOM_BACKFACE_FLIP** mode can be used to cause the normals to be flipped and the backfaces to lit like front faces. Using bi-directional light sources is a partial work around for systems that do not support the **GEOM_BACKFACE_FLIP** rendering attribute.

GEOM_BACKFACE_CULL_BACK
This backface mode causes the renderer to not draw polygons that are backfacing.

GEOM_BACKFACE_CULL_FRONT
Some renderers support this rendering mode in which front faces are not drawn. This mode is of limited utility and is only useful when the renderer does not accurately determine front versus backfaces.

GEOM_BACKFACE_INHERIT
This mode causes the specified object to inherit the backface property of its parent object. This is the default backface property for a newly created object.

GEOMedit_camera_orient

GEOMedit_camera_orient(*list, name, flags, scale, at, up, from*)
GEOMedit_list *list;*
char **name;*
int *flags;*
float *scale, at[3], up[3], from[3];*

This routine sets the camera orientation of the camera specified by *name*. See the section "Edit Lists" for information on how to specify camera names. The *flags* argument contains the or'd combination of the values: **GEOM_CAMERA_SCALE**, **GEOM_CAMERA_AT**, **GEOM_CAMERA_UP**, **GEOM_CAMERA_FROM**. If you use one of the above flags, the corresponding argument will be used to set the camera orientation. If a flag is not specified, then the corresponding argument you supply will be ignored by the routine. You must, however, supply a valid floating point number or array of floating point numbers in its place. Normally, you can use the value **GEOM_CAMERA_ALL** to indicate that all of the values should be applied to the camera.

See the section in the "Geometry Viewer Subsystem" chapter of the *User's Guide* on "Camera Options" to determine the precise interpretation of the values: *scale*, *at*, *up*, and *from*.

GEOMedit_camera_orient modifies the same transformation as the routine **GEOMedit_set_matrix** when **GEOMedit_set_matrix** is used with a camera name argument.

GEOMedit_camera_params

GEOMedit_camera_params(*list, name, options, val*)
GEOMedit_list *list;*
char **name;*
int *options, val;*

This routine modifies camera parameters for the camera specified by the *name* parameter. (See the section on "Edit Lists" for more indicates which parameters values are to be modified. The *val* parameter indicates whether to turn the parameters on or off. Possible values for the *options* parameter are:

GEOM_CAMERA_DEPTH_CUE
GEOM_CAMERA_ZBUFFER
GEOM_CAMERA_SORT_TRANSPARENCY
GEOM_CAMERA_GLOBAL_ANTIALIAS
GEOM_CAMERA_PERSPECTIVE
GEOM_CAMERA_AXES
GEOM_CAMERA_FREEZE
GEOM_CAMERA_SHOW
GEOM_CAMERA_DOUBLE_BUFFER
GEOM_CAMERA_SHADOWS

The *val* parameter is a 1 to turn the specified parameters on, and a 0 to turn the specified parameters off.

GEOMedit_camera_project

GEOMedit_camera_project(*list, name, flags, front, back, fov, wsize*)
GEOMedit_list *list;*
char **name;*
int *flags;*
float *front, back, fov, wsize;*

This routine sets the camera projection matrix of the camera specified by *name*. See the section "Edit Lists" for information on how to specify camera names. The *flags* argument can contain the or'd combination of the values: **GEOM_CAMERA_FRONT**, **GEOM_CAMERA_BACK**, **GEOM_CAMERA_WSIZE**, **GEOM_CAMERA_FOV**. If you use one of the above flags, the corresponding argument will be used to set the camera orientations. If a flag is not specified, then the corresponding argument you supply will be ignored. You must, however, pass a valid floating point number in its place. Normally you can use the value: **GEOM_CAMERA_ALL** to indicate that all of the values should be applied to the camera.

See the "Camera Options" section in the "Geometry Viewer Subsystem" chapter of the *User's Guide* for an explanation of the precise interpretation of the values: *front*, *back*, *fov*, and *wsize*.

This routine modifies the same transformation as the routine: **GEOMedit_projection**.

GEOMedit_center

GEOMedit_center(*list, name, center***)**
GEOMedit_list *list;*
char **name;*
float *center[3];*

Sets the center of rotation of the object specified. This does not currently work for lights. The center of rotation is defined before the object's transformation matrix is applied. It should, therefore, be defined in the same coordinate system as the vertices of the object.

GEOMedit_clip_plane

GEOMedit_clip_plane(*edit_list, name, clip, state*)
GEOMedit_list *list;*
char **name;*
char **clip;*
int *state;*

This routine can be used to specify arbitrary clip planes from a module. Arbitrary clip planes cause geometric objects to be clipped by a plane that has an arbitrary position and orientation. This is implemented using a geometric object to represent the clip plane. We call this the "clip object." The clip object's transformation is then used to specify the position and orientation of the clip plane. The default position of the clip plane is (0,0,0) and the initial normal to the clip plane is the Y axis. Objects are "inside" the clip plane if they are in the positive direction of the Y axis. The position and orientation are transformed by the clip object's current transformation matrix.

An object can be clipped either to the "inside" or "outside" of the clip object. If an object is clipped to the "inside" and the clip object has an identity transformation, only the geometry that has a positive Y component will be drawn. If an object is clipped to the "outside", only geometry with a negative Y component will be drawn.

The **GEOMedit_clip_plane** defines the object with name *clip* to be a clip object for the object named *name*. If the object named *clip* does not exist, a new object is created as an immediate child of the *top* object. *clip* can be any existing object. All that is used from the clip object is its transformation matrix to define a clip plane that will clip *name*.

The clip state of an object, defined by the *state* parameter, can be set to one of the following values:

GEOM_CLIP_INSIDE
GEOM_CLIP_OUTSIDE
GEOM_CLIP_IGNORE
GEOM_CLIP_INHERIT

The default clip state for a given clip plane object and object to be clipped is **GEOM_CLIP_INHERIT**. Here the clip state is inherited by all descendants of *name* that do not specify a different state for the same clip object. For example, if you specify that the object "bar" should clip the "top" object to the inside (with state **GEOM_CLIP_INSIDE**), all descendants of "top" (including "bar" itself) will be clipped by the clip plane defined by the current transformation of "bar". If you then specify that "bar" should *not* be clipped by "bar" (with

state **GEOM_CLIP_IGNORE**), all descendants of "top" except for "bar" and its descendants will be clipped.

An arbitrary number of clip planes may be defined, but different rendering implementations may provide a different number of actual clip planes.

This feature is only supported on some renderers. It can always be accessed through the software renderer if your particular hardware renderer does not support it.

GEOMedit_color

GEOMedit_color(*list, name, color***)**
GEOMedit_list *list;*
char **name;*
float *color[3];*

Sets the color of the object named *name*. The *color* argument is an RGB triple, each float in the range 0.0 to 1.0. If the *name* argument is "camera*n*", where *n* is an integer ranging from 1 to the number of views, this routine sets the background color for the view specified by the index *n*. If the *name* argument is "light*n*", where *n* is an integer ranging from 1 to the number of light sources, this routine sets the light source color for the light source specified by the index *n*.

GEOMedit_concat_matrix

GEOMedit_concat_matrix(*list, name, matrix***)**
GEOMedit_list *list;*
char **name;*
float *matrix[4][4];*

Post-concatenates *matrix* to the matrix of the object named *name*. If the *name* argument is "camera*n*", where *n* is an integer ranging from 1 to the number of views, this routine post-concatenates *matrix* to the camera matrix for the view specified by the index *n*. If the *name* argument is "light*n*", where *n* is an integer ranging from 1 to the number of light sources, this routine post-concatenates *matrix* to the light matrix for the light source specified by the index *n*.

GEOMedit_depth_cue_params

GEOMedit_depth_cue_params(*list, name, flags, depth_front, depth_back, depth_scale*)
GEOMedit_list *list;*
char *name;
int *flags;*
float *depth_front, depth_back, depth_scale;*

This routine modifies the depth cueing attributes for the camera defined by *name*. See the "Edit Lists" section for more information on how to specfiy camera names. The *flags* argument is an or'd combination of the following constants: **GEOM_DEPTH_CUE_FRONT**, **GEOM_DEPTH_CUE_BACK**, and **GEOM_DEPTH_CUE_SCALE**. If you use one of the above constants, the corresponding attribute will be modified. Alternatively, you can specify the constant **GEOM_DEPTH_CUE_ALL** to modify *depth_front*, *depth_back*, and *depth_scale* parameters.

See the "Camera Options" section of the "Geometry Viewer Subsystem" chapter in the *User's Guide* for the precise interpretation of the depth cue parameters: *depth_front*, *depth_back*, and *depth_scale*,

GEOMedit_geometry

GEOMedit_geometry(*list, name, obj***)**
GEOMedit_list *list;*
char **name;*
GEOMobj **obj;*

Specifies a change in the geometry for an object named *name* in the edit list *list*. The first edit geometry entry in an edit list for a specific object replaces all existing geometry for this object with the geometry specified by *obj*. All other edit geometry entries for the object named *name* simply add additional geometry to that object.

Entering geometry into an edit list does not copy the geometric description of the object. Instead, it creates a reference to the **GEOMobj** specified in the call to **GEOMedit_geometry**. This means that a module must take care not to modify the **GEOMobj** until the edit list has been destroyed. The module must also destroy its own reference to the **GEOMobj**, using **GEOMdestroy_obj**, when it is finished with the geometry. For most purposes, a call to **GEOMdestroy_obj** should be made after every call to **GEOMedit_geometry**.

GEOMedit_parent

GEOMedit_parent(*list, name, parent***)**
GEOMedit_list *list;*
char **name;*
char **parent;*

Sets the parent of the object named *name* to be the object named *parent*. The top level object is referred to by a "NULL" name entry.

GEOMedit_light

GEOMedit_light(*list, name, type, status***)**
GEOMedit_list *list;*
char **name, *type;*
int *status;*

Changes the light source representation for a light source. The light source name is "light*n*", where *n* is an integer from 1 through 16. The *type* argument is one of "spot", "directional", "point", or "bi-directional". If the *status* argument is 1, the light source is on; if the *status* argument is 0, the light source is off.

GEOMedit_position

GEOMedit_position(*list, name, position***)**
GEOMedit_list *list;*
char **name;*
float *position[3];*

Sets the position vector for the object specified. Positions are always applied after the matrix that you can set with **GEOMedit_set_matrix**. See the section on Geometry Viewer transformations for more information on the position.

GEOMedit_properties

GEOMedit_properties(*list, name, ambient, diffuse, specular, spec_exp, transparency, spec_col***)**
GEOMedit_list *list;*
char **name;*
float *ambient, diffuse, specular, spec_exp;*
float *transparency, spec_col[3];*

Changes the material properties of the object named *name*. The properties are ambient, diffuse, and specular reflection coefficients; specular exponent; transparency; and specular color (as an RGB triple). Values to be changed are in the range 0.0 to 1.0 except for the specular exponent, which is greater than or equal to 1.0. If any value is -1.0, that property is not changed.

GEOMedit_projection

GEOMedit_projection(*list, name, projection***)**
GEOMedit_list *list;*
char **name;*
float *projection[4][4];*

Sets the projection matrix for a particular camera. The argument "name" is as described above under "Edit Lists". See the section on Geometry Viewer transformations in this chapter for more information on how the projection is applied.

GEOMedit_render_mode

GEOMedit_render_mode(*list, name, mode***)**
GEOMedit_list *list;*
char **name;*
char **mode;*

Sets the render mode of the object named *name* to one of "gouraud", "phong", "lines", "smooth_lines", "no_light", "inherit", or "flat".

GEOMedit_selection_mode

GEOMedit_selection_mode(*list,name,mode,flags***)**
GEOMedit_list *list;*
char **name;*
char **mode;*
int *flags;*

This routine sets the selection mode of the object given. It can be used for two major purposes: 1) to make an object "unpickable" by the user, 2) to allow an upstream module to receive pick information when this object is selected.

Values for the "mode" argument are:

notify
Notify the calling module when the object "name" is selected by the user. If the object is subsequently selected, and the module has an input port connected to the render geometry output port named: "Geom Info", the module will be executed with some information pertaining to the specifics of the selection.

normal
Restore the selection mode of the object to normal. No module will receive selection information from the object.

ignore
Do not allow the user to pick the object specified. Any attempt to pick the object will result in a pick of the parent object instead.

The flags argument is only relevant when the **notify** mode is set. It should contain one or more of the following flags: **BUTTON_DOWN**, **BUTTON_UP**, **BUTTON_MOVING**. The definition for these flags is contained in the include file <avs/udata.h>.

The flags field indicates for what button states information should be redirected to the module.

GEOMedit_set_matrix

GEOMedit_set_matrix(*list, name, matrix***)**
GEOMedit_list *list;*
char **name;*
float *matrix[4][4];*

Sets the transformation matrix for the object named *name* to the matrix *matrix*. If the *name* argument is "camera*n*", where *n* is an integer ranging from 1 to the number of views, this routine sets the camera matrix for the view specified by the index *n*. If the *name* argument is "light*n*", where *n* is an integer ranging from 1 to the number of light sources, this routine sets the light matrix for the light source specified by the index *n*.

GEOMedit_subdivision

GEOMedit_subdivision(*list, name, subdiv*)
GEOMedit_list *list;*
char **name;*
int *subdiv;*

This routine sets the sphere subdivision object attribute of the object specified by *name* to the value *subdiv*. The *subdiv* parameter determines the number of polygons used to tesselate a sphere on systems that cannot render spheres directly. A subdivision parameter of **1** draws spheres as tetrahedra. A parameter of 2 subdivides each triangle in the tetrahedra into three triangles. This process occurs so that with each increment in the subdivision parameter, each triangle in the previous subdivision level is subdivided into three new triangles.

GEOMedit_texture

GEOMedit_texture(*list, name, texture***)**
GEOMedit_list *list;*
char **name;*
char **texture;*

This routine sets the texture of a particular object. The texture argument can be the keyword "dynamic" or a filename containing the path name of a texture file. The keyword "dynamic" tells the module to set the texture that is present on the "Texture Input" port of the **geometry viewer** module. If there is currently no data present on that port, setting the texture to "dynamic" will be ignored.

If a filename is specified, the Geometry Viewer will read in this file and apply the texture contained in this file to the object specified. The filename must be accessible from the host that the **geometry viewer** module is running on. The format of the file is the same as the file format for reading in textures from the Geometry Viewer. See the *User's Guide* documentation on texture mapping in the "Geometry Viewer Subsystem" chapter for more details.

On systems that do not supported texture mapping, this attribute will be ignored.

GEOMedit_texture_options

GEOMedit_texture_options(*list, name, options, val*)
GEOMedit_list *list;*
char **name;*
int *options, val;*

This routine is used to change the method by which a particular named object is texture mapped by turning on and off particular texture mapping options. The object whose texture options are to be modified is referred to by the *name* argument. The *options* argument specifies which options are to be changed. Possible values are: **GEOM_TEXTURE_FILTER**, **GEOM_TEXTURE_ALPHA**, and **GEOM_TEXTURE_VOLUME**. The *val* argument specifies whether or not the particular flag is to be turned on or off. A value of 1 specifies that the flag is to be turned on, a 0 specifies that the value is to be turned off. The **GEOM_TEXTURE_FILTER** argument, when turned on, indicates that the object, when texture mapped, should be rendered with filtered (i.e., antialiased) textures. The **GEOM_TEXTURE_ALPHA** argument when enabled specifies that the alpha channel of the texture map should be used for opacity values when rendering the texture mapped object. An opacity of 0 means that the object is totally transparent, an opacity of 255 means that pixel is totally opaque. The **GEOM_TEXTURE_VOLUME** specifies that a 3D texture should be treated as a volume primitive for the object. In the case of volume rendering, the extents of the object to be volume renderered are defined by the "window" of the object as defined by the **GEOMedit_window** routine.

GEOMedit_transform_mode

GEOMedit_transform_mode(*list,name,mode,flags***)**
GEOMedit_list *list;*
char **name;*
char **mode;*
int *flags;*

This routine sets the transformation mode of the object with the given name. It can be used for two purposes: 1) to prevent the user from accidentally transforming an object which should always be defined in the coordinate system of

its parent and 2) to allow an upstream module to receive notification when the object is transformed by the user.

Values for the "mode" argument are:

normal
Restore the transform mode to the default or normal mode. In this case, the "flags" argument is ignored.

parent
Any transformations that are applied to this object are redirected to the parent object. This mode can be used to prevent the user from transforming this object relative to its parent object. In this case, the "flags" argument is ignored.

notify
Notify the calling module when the object "name" is transformed by the user. If the object is subsequently transformed, and the module has an input port connected to the render geometry output port named: "Transform Info", the module will be executed with a variety of information including the transformation matrix of the object.

redirect
This mode is similar to the "notify" mode above. There are only two differences: 1) the transformation matrix that is accumulated for the object and passed to the module is not used in transforming the geometry of the object, 2) since the Geometry Viewer is not going to directly transform the object when the transformation matrix changes, it does not refresh the display. This mode is useful when the module is always going to regenerate the geometry of the object each time that the transformation matrix changes. Note that since the identity matrix will be used when rendering the object always, that the module will have to transform any vertices generated itself.

The flags argument is only relevant when the "notify" or "redirect" transform mode is set. It should contain one or more of the following flags: **BUTTON_DOWN**, **BUTTON_UP**, **BUTTON_MOVING**. The definition for these flags is contained in the include file <*avs/udata.h*>.

The flags field indicates for what button states information should be redirected to the module. Transformations that are not caused by mouse movement use the state **BUTTON_UP**.

See the "Upstream Data" section of the "Advanced Topics" chapter for more information on using the transform mode from a module.

GEOMedit_visibility

GEOMedit_visibility(*list, name, visibility***)**
GEOMedit_list *list;*

char **name;*
int *visibility;*

Sets the visibility of the object named *name*. If the *visibility* argument is 1, the visibility is set to on; if the *visibility* argument is 0, the visibility is set to off; if the *visibility* argument is -1, the object is deleted.

GEOMedit_window

GEOMedit_window(*list,name,window***)**
GEOMedit_list *list;*
char **name;*
float *window[6];*

This routine allows the user to specify the "window" of interested for an object. It allows a module to cause the Geometry Viewer to "auto-normalize" and "auto-center" the top level object when the window changes. By default, the Geometry Viewer will only display geometry that is in the range -5 to 5 in X and Y. Either you must scale and translate your data to lie in this range or you must change the transformation matrix of either the object, a parent of the object or the camera so that your geometry will become viewable.

The **GEOMedit_window** routine implements a mechanism whereby the geometry viewer will handle this scale and translate automatically. It does so in a way that allows multiple geometry producing modules to cooperatively decide on a global scale / translate that displays all geometries that are produced. It also keeps that data in the natural coordinate system in which the data is defined. This allows the Geometry Viewer to display data sets that are defined in the same physical coordinate system simultaneously without distorting their interrelationships.

The window that is specified by the module contains an array of 6 floating point numbers in the order: minimum X, maximum X, minimum Y, maximum Y, minimum Z, maximum Z. These values define a bounding box relative to the top level object (i.e. not transformed by the object's own transformation). This box should contain the range of the coordinate system of interest. For example, if your vertices lie within 0-100 in X, Y and Z, your window should be: 0, 100, 0, 100, 0, 100. The window should include any transformations that are going to be applied to the object (not including the top level object). For example, if your vertices are defined as above, but you are going to scale your object down by a factor of 2, the window should be set to : 0, 50, 0, 50, 0, 50.

The window is associated with a particular object. While that object still exists in the scene and it still has a window defined for it, it will continue to be used to determine the scale and position of the top level object.

The Geometry Viewer maintains a global "window" that includes the extent of all windows currently defined in the scene. Whenever an "edit window" request is received, the Geometry Viewer recomputes the new window by com-

puting a box that surrounds all of the windows currently defined. If the new global window is different from the old global window, the scene is scaled and translated so that the new global window will lie inside of the viewable region of the screen. The rotation / scale center of the top level object is also set to be the center point of the global window.

If the window does not change between subsequent "edit window" requests, the top level object's transformation is left unchanged.

GEOMinit_edit_list

GEOMedit_list
GEOMinit_edit_list(*list***)**
GEOMedit_list *list;*

Initializes an existing edit list (removes all existing entries). If list is **GEOM_-NULL**, it returns a new empty edit list.

FORTRAN Binding

All of the geom routines also have a FORTRAN calling sequence. To call a routine from a FORTRAN program, you must use a slightly different routine name and different data declarations:

Routine Name:
Replace the **GEOM** prefix with **geom_ (***note the underscore***)**.

Data Declarations:
The following table shows how to convert C-language data declarations into FORTRAN declarations:

Table G-2 Converting C Language Data Declarations to FORTRAN

C Declaration	FORTRAN Declaration	
int	*var*	**INTEGER**
float	**var*	**REAL**
unsigned	**int**	*var*
unsigned	**int**	**var*
float	*var*	**REAL**
float	**var*	**REAL**
double	*var*	**REAL**
double	**var*	**REAL**
GEOMobj	**var*	**INTEGER**
GEOMobj	***var*	**INTEGER**
GEOMedit_list	*var*	**INTEGER**
GEOMedit_list	**var*	**INTEGER**

Many routines allow a NULL value for some arguments. In all such cases, the constant **GEOM_NULL** must be used to represent a NULL value.

Files

/usr/avs/include/geom.h	C language header file
/usr/avs/include/geom.inc	FORTRAN header file
/usr/avs/lib/libgeom.a	geom library

APPENDIX H

THE F77_BINDING UTILITY PROGRAM

Introduction

AVS includes a utility program, **f77_binding**, that generates inter-language interface functions. Such functions allow code written in C to call subprograms written in Fortran, and vice-versa. Using **f77_binding** makes it easier to create code that will port easily to different platforms.

f77_binding can also be used to generate Fortran include files that provide constant and function declarations.

Inter-Language Calling Conventions

There is no standard inter-language protocol for C and Fortran-77, but there are some commonly used conventions. The fundamental conventions to be established are:

- Function naming rules and length restrictions.
- Matching the C pass-by-value convention to the Fortran pass-by-reference convention.
- Handling of string arguments.
- Handling of function return values.

You specify the particular conventions to be applied as command-line options to the **f77_binding** command, as described in the sections that follow.

Function Naming Rules

In order to reference functions across the C-Fortran language barrier, many compilers use special naming conventions. These conventions include capitalization of the function name and the addition of special suffixes. For example:

- On HP, Sun, IBM, Silicon Graphics and DEC systems, a Fortran program that calls a C function named **hello** works only if there is a C function named **hello_**. That is, the interface function name is all lowercase and has an underscore suffix.

- On ST1500 / ST3000 systems, the corresponding interface function is **HELLO**. That is, the name is all uppercase and has no suffix.

The **-case** and **-suffix** options to **f77_binding** allow you to specify these (and other) naming conventions. You can generate interface functions that allow Fortran functions to call C functions (**-result f77_to_c**), and vice-versa (**-result c_to_f77**). The file */usr/avs/include/Makeinclude* uses the macro F77_BIND_FLAGS to specify the appropriate conventions for each AVS implementation. This file should be included in your makefile.

The Fortran-77 standard does not permit names of Fortran functions to exceed 6 characters in length (although many Fortran implementations are much more generous). **f77_binding** can generate a short form of a long function name (**-name** option), for use with strict implementations.

Matching C and Fortran Calling Conventions

C functions expect to receive an argument *value*; if the function needs to modify the argument, the *address* of the value must be passed. A Fortran subprogram always expects that the *address* of its argument is being passed to it. Thus, when a C function is called from a Fortran routine, it must dereference the pointers it is being given to get their values. Similarly, a Fortran subprogram called from a C function should be passed the addresses of the C arguments when necessary. **f77_binding** handles this pointer conversion automatically, using the argument declarations.

Handling String Arguments

A string consists of two pieces of information: a character array and a length. Different Fortran compilers implement string passing differently. Some pass strings as two arguments, a character array and an extra length argument tacked onto the end of the argument list. Others pass a two word structure that contains both a pointer to the character array and the length together in one argument. **f77_binding** relies on a series of macros defined in */usr/avs/include/port.h* to handle both these cases.

Handling Function Return Values

f77_binding generally handles simple scalar return values: **float** and **int**. Other return values (e.g. strings) are much less portable. In some cases, returned string values can be handled, but this will not work across all systems and is discouraged. See "Argument Declarations" below for more details.

f77_binding Function Declarations

Interface functions are produced by **f77_binding** based on **function declarations**. A function declaration suitable for use as input to **f77_binding** is similar to an ordinary C function declaration, except that it describes the argument types as well. It consists of a return type, a function name, and a list of argument types. For example:

```
int AVSautofree_output( int);
```

This function declaration describes a function **AVSautofree_output** that returns an integer and takes one integer argument. **f77_binding** can produce the following interface function for use by a Fortran routine calling a C function:

```
int avsautofree_output_(arg0)
int *arg0;
{
   return(AVSautofree_output(*arg0));
}
```

Return Types

f77_binding recognizes the following return types, which use C type names:

int
INTEGER*4: Use this for returning pointer values to Fortran

c_int
Fortran subroutine call to a C function that would ordinarily return an integer

float
REAL*4

double
REAL*8

void
Fortran subroutine

Function Names

A function name part of a function declaration has the following three-part format:

optional-shortname fullname optional-suffix

Note that no SPACE characters may occur within the function name — if you use more than one part, they must be concatenated to form a single "word".

The *fullname* is the name of the function to be called by the interface function, with arguments that are properly packaged.

The optional *shortname* is a shorter function name (six characters or fewer), for use on systems that do not tolerate longer function names. If you include this alternative, you must separate it from the *fullname* with an up-arrow (^) character. For example:

```
dblchk^double_check
```

The *optional-suffix* may be either of the following:

@raw
: In some cases, the standard argument packaging is inadequate and you must write a custom interface function. The suffix **@raw** causes the interface function to call a function named *fullname***_raw** that expects its arguments to be unmodified (not packaged). The *fullname***_raw** function can "manually" package or unpackage the arguments, using the string-handling macros as necessary. You may want to use **f77_binding** without the **@raw** suffix to generate a template for this function.

@f
: This suffix is similar to **@raw**, except that the function *fullname***_f** expects its arguments to have already been packaged. Use this suffix when you don't want to call the target function directly, but want to add some intermediate processing of arguments, such as adjusting argument values between languages or post-processing the return value from the target function.

Argument Declarations

The argument declarations establish the C type of the argument and are used to determine both the number of arguments and how they are to be handled. Recognized types include the following:

Table H-1 Recognized C and FORTRAN Types

C type	FORTRAN type	Description
float	REAL*4	Scalar float
float *	REAL*4 array	Array of floats
double	REAL*8	Scalar double
int	INTEGER*4	Scalar integer
int *	INTEGER array	Array of integers
int(*)()		Integer function pointer
any*	INTEGER*4	Pointer to any local data type structure

Table H-1 Recognized C and FORTRAN Types

C type	FORTRAN type	Description
char *	character*(*)	Used for character string arguments
char []	character *(*)	Special case for short strings. It allocates a **static char** array in the interface function, rather than allocating and freeing storage with each call. Maximum length is declared in *<avs/port.h>*.
answer char *		Function result is copied into the argument
return char *		Function returns a char* (must be first argument)

Other Lines

An input file to **f77_binding** can contain other lines besides function declarations. Lines that start with "#include" are passed along directly to the output. Comments surrounded by "/* */" are conditionally passed through to the output, depending on the **-comments** option. When the output is a Fortran include file, they are automatically preceded by a comment header, "C ".

Fortran Include Files

f77_binding can generate Fortran include files from either conventional C header files or function declaration files.

Use the **-result f77_parm** option to process a conventional C header file. With this option, **f77_binding** converts "#define" statements into the equivalent Fortran PARAMETER statements where possible, and also converts values as necessary. By default, **f77_binding** examines all lines. To skip over a block of lines, surround them with "#ifndef F77" ... "#endif F77". Lines to be examined *only* by **f77_binding** can be highlighted using "#ifdef F77" ... "#endif F77".

A "#define" line in the C header file can include a special comment that specifies a short name (<= 6 characters).

```
#define GEOM_MESH  1 /*_F77_s: GPMESH */
```

The comment must start with the string "_F77_s".

Use the **-result f77_func** option to process a function declaration file. This options allows the same input to generate both the interface functions and the corresponding Fortran function declarations for use in an include file.

f77_binding Command-Line Syntax

The format of the **f77_binding** command is:

f77_binding [*options*] [*input-file*] [**-o** *output-file*]

If no input file is specified on the **f77_binding** command line, input is read from standard input. If no output file is specified with the **-o** option, program output is sent to standard output.

Options

The **f77_binding** program accepts the following command-line options.

-name short
Create interface functions with short (6-character) names.

-name long
Create interface functions with long names.

-name both
(Default) Create both long-name and short-name interface functions.

-case lower
(Default) Interface function names should be all lowercase.

-case upper
Interface function names should be all uppercase.

-comments
Include comments in the output.

+comments
(Default) Exclude comments from the output.

-external
(Default) When creating a Fortran include file, add EXTERN declarations for all function declarations.

+external
Omit EXTERN declarations for Fortran functions when creating an include file.

-result f77_to_c
(Default) Create interface functions for use by Fortran routines calling C functions.

-result c_to_f77
Create interface functions for use by C functions calling Fortran subprograms.

-result f77_func
Produce Fortran include file function declarations.

-result f77_parm
Produce Fortran include file from C header file.

-suffix *AAA*
Add suffix *AAA* to the end of the name of each interface function.

-subst *AAA BBB*
Change occurrences of string *AAA* in the input function declarations to string *BBB* in the names of the interface functions generated. You can also use this option when generating the Fortran include file declarations ("-result f77_func").

-o *file*
Write output to the specified file. File output will have an additional comment added noting the original source file (unless standard input was used).

-usage
Display usage message.

Examples

In the */usr/avs/examples* directory, the **qix_f** module calls the C function **drand48** by declaring it in the *qix_rand.h* header file as:

```
double drand48();
```

and then adding the following to the Makefile:

```
include $(ROOT)/usr/avs/include/Makeinclude
F77_BIND = $(ROOT)/usr/avs/bin/f77_binding
qix_rand.c: qix_rand.h
     $(F77_BIND) qix_rand.h $(F77_BIND_FLAGS) -o qix_rand.c
```