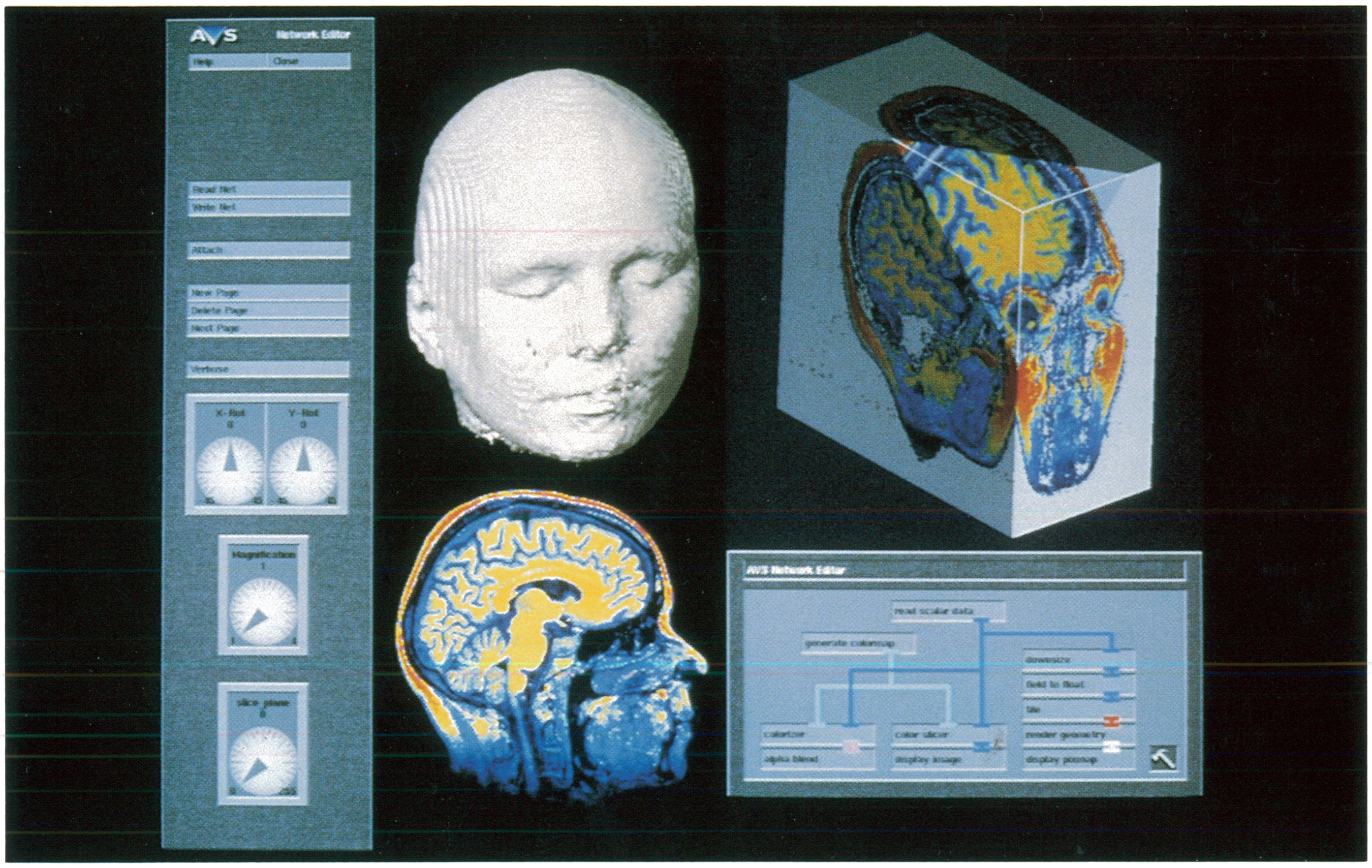
Stardent


AVS
Network Editor
Help
Close
Read Net
Write Net
Attach
New Page
Delete Page
Next Page
Verbose
X-Rot
Y-Rot
Magnification
slice plane
AVS Network Editor
read scalar data
generate colormap
downsize
field to float
tile
colorizer
color slicer
render geometry
alpha blend
display image
display pixmap

Stardent

Application Visualization System

Developer's Guide

MD – 2302

Table of Contents

Stardent

1

Overview

Table of Contents

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

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.

A programmer can extend the capabilities of AVS conveniently by writing a new module. Because AVS operates on fairly general data types, a new module can work together with pre-existing modules to perform computation.

Most modules consist of two functions: the *description function* and the *computation function*. These functions can be written in either the C language 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 does the real work of the module. It is a function that is called with its inputs, parameters, and outputs as arguments. The computation function typically operates on the inputs and parameters to produce new output.

There are two kinds of modules, *subroutine modules* and *coroutine modules*. A subroutine module is invoked by AVS, usually 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.

Many existing simulations and other scientific applications can be converted easily into AVS coroutine modules. Conversions steps include 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 *aggregrate data*. Primitive data items are simple objects such as 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 *fields*. These implement array structures (either uniform or non-uniform, scalar or vector) as well as unconnected and irregular structures. AVS also has other types of aggregate data, including geometries, colormaps, and pixel maps.

In general, modules process aggregate data and use primitive data as parameters (although there are exceptions, and usually any type of data can be used in any situation). Parameters can be modified by the user at runtime using various interactive mechanisms such as dials, sliders, and browsers.

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 or storing AVS data on an external device, such as the display screen or 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. Two ports can be connected when they have matching AVS data types.

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. It also sets up a connection between the module and AVS. Both sides use remote procedure calls 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.

In this release, only one module executes at a time; modules do not execute in parallel.

Stardent

2

AVS Data Types

Table of Contents

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. The more general the data a module can handle for input, the more modules it can be connected to and, therefore, the more reusable it is.

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, and the system makes no distinction between the data types.

The currently supported AVS data types are these:

- *Byte* implements 8-bit bytes.
- *Integer* implements standard 32-bit integers.
- *Real* implements 32-bit IEEE single-precision floating-point numbers.
- *String* implements simple text strings.
- *Field* implements n-dimensional arrays with scalar or vector data at each point. Fields also have support for arbitrary rectilinear or irregular coordinate systems, and they can represent lists of points in coordinate space. Fields can contain floating-point, integer, or byte data.
- *Colormap* implements a transfer function that can be used to map a functional value into color and opacity values.
- *Geometry* implements geometric descriptions which can be used by the geometric renderer to view objects. Geometry objects are usually created using calls to subroutines in the geom library; see the **geom**(3V) manual page for more information.
- *Pixel map* is actually a reference to the X server's representation of the rendered form of an image.

Fields can be considered AVS's fundamental data type. They use the full generality of AVS's type system to span a set of commonly used data types. This allows programmers to write modules that are as general as is appropriate for the application while allowing optimized algorithms to be used for specific cases. Typically the output data from a standard scientific simulation can be represented as a field. AVS routines allow 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 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;
{
  float *data_out;
  AVSfield_float *result;

  dim0 = MAXX(field_in);
  dim1 = MAXY(field_in);
  dim2 = MAXZ(field_in);
```

→

```
    data_out = (float *) malloc(dim0*dim1*dim2*sizeof(float));
    result = AVSbuild_3d_field(data_out, dim0, dim1, dim2);

    ... compute ...

    *field_out = result;
    return(1);
}
```

Since FORTRAN programs typically do not use structures in the same way as C programs, FORTRAN computation routines get their arguments as separate elements. For example, a subroutine that takes a 3D scalar field as input gets arguments in this form:

```
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.

The following table 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**	Pointer to **REAL**
string	**char** *	**char** **	**CHARACTER***(*)	Pointer to **CHARACTER***(*)
field	**AVSfield** *	**AVSfield** **	—	—
colormap	**AVScolormap** *	**AVScolormap** **	—	—
geometry	**GEOMedit_list**	**GEOMedit_list** *	INTEGER	INTEGER
pixel map	**AVSpixdata** *	**AVSpixdata** **	—	—

A field is passed to a FORTRAN computation routine as multiple arguments; see the "Fields" section below. A FORTRAN computation routine cannot take a colormap or a pixel map as an argument.

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". On Stardent systems the most significant bit is stored first. 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 represented as integers: "boolean", "tristate", and "oneshot". See the documentation for the **AVSadd_parameter** routine in the appendix "AVS Routines".

Floating-Point Numbers

AVS supports floating-point numbers in IEEE format. 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).

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: 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. Coordinate space has the same number of dimensions as computational space. 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. 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 implicit, the field does not need any coordinate information separate from the data array.

Rectilinear Fields

In rectilinear fields, as in uniform 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.

AVS Mapping Information

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

For a uniform field AVS needs no explicit mapping information. The *X* coordinate for a data element is simply the subscript of the data element along the first dimension of computational space; the *Y* coordinate is the subscript of the data element along the second dimension of computational space; and so on.

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.

The following table 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 Axis	$X = i$ $Y = j$ ...
Rectilinear	Explicit	Computational Dimension to Coordinate Axis	$X = X(i)$ $Y = Y(j)$ ...
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

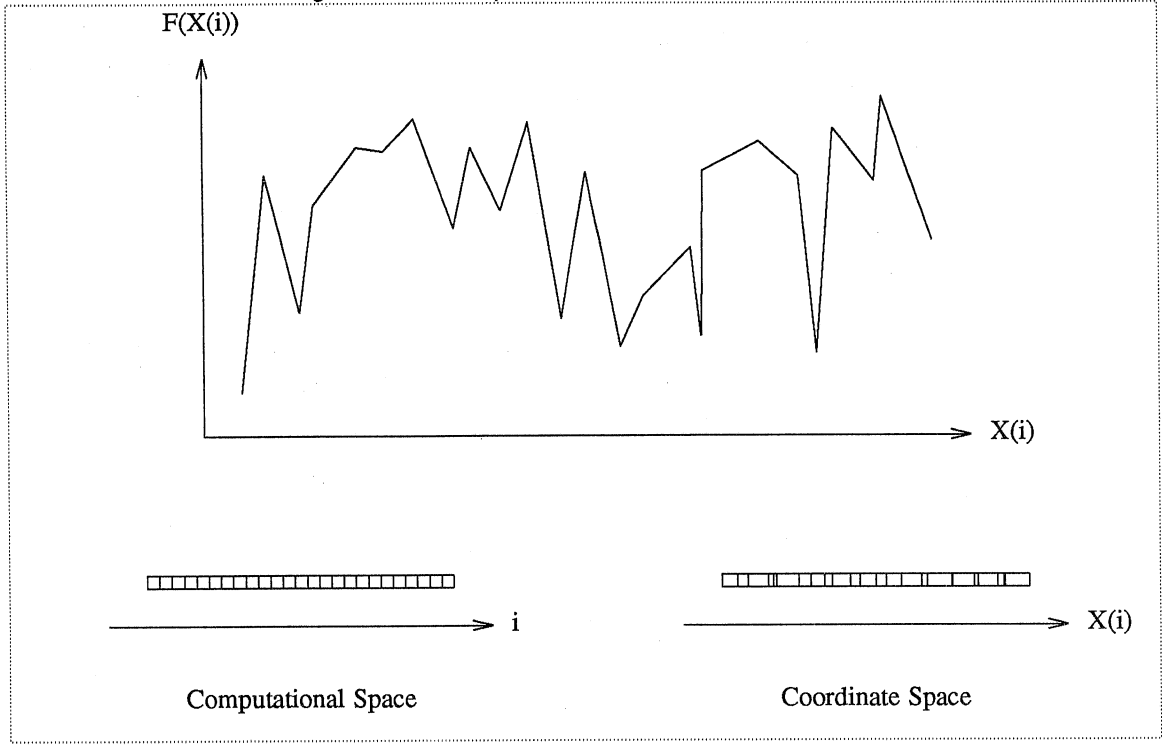
A data set consists of 25 data elements, each representing $F(X)$ for a given value of X. The field consists of 25 elements:

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

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 1 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 2-1. Example 1



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:	50
Number of computational values:	$1*50=50$
Mapping type:	Rectilinear or irregular
Number of coordinate dimensions:	1
Number of coordinate values:	50

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:	50
Number of computational values:	$2*50=100$
Mapping type:	Rectilinear or irregular
Number of coordinate dimensions:	1
Number of coordinate values:	50

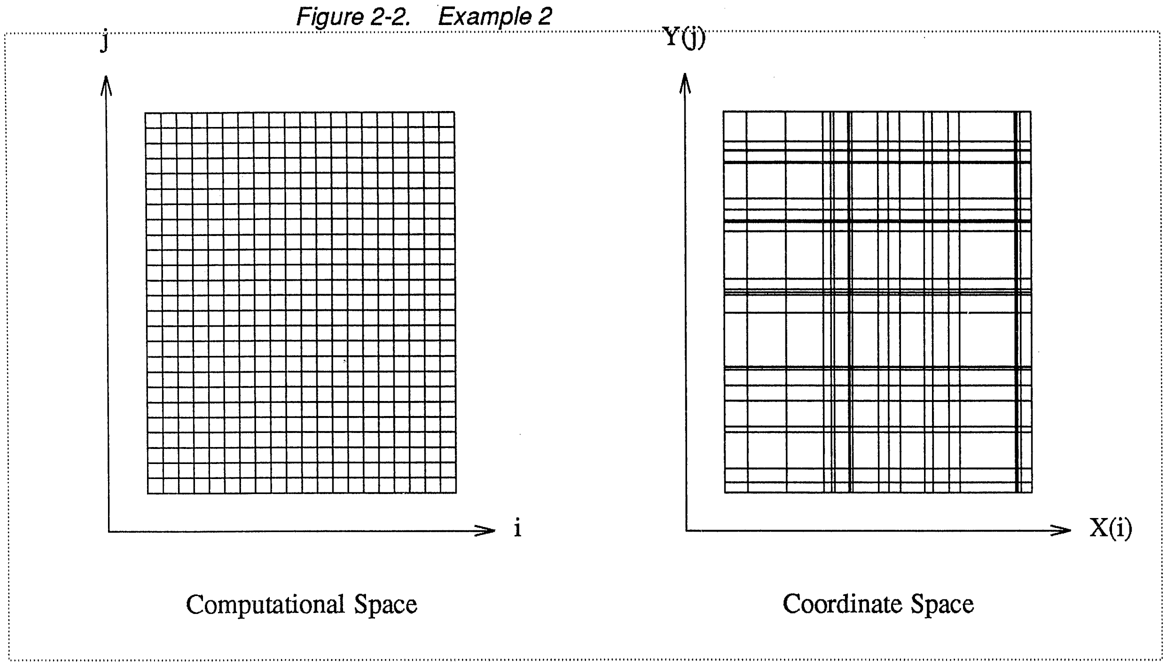
Example 2

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:

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

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

Figure 2-2. Example 2



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:	20×25
Number of computational values:	1∗20∗25 = 500
Mapping type:	Rectilinear
Number of coordinate dimensions:	2
Number of coordinate values:	20+25 = 45

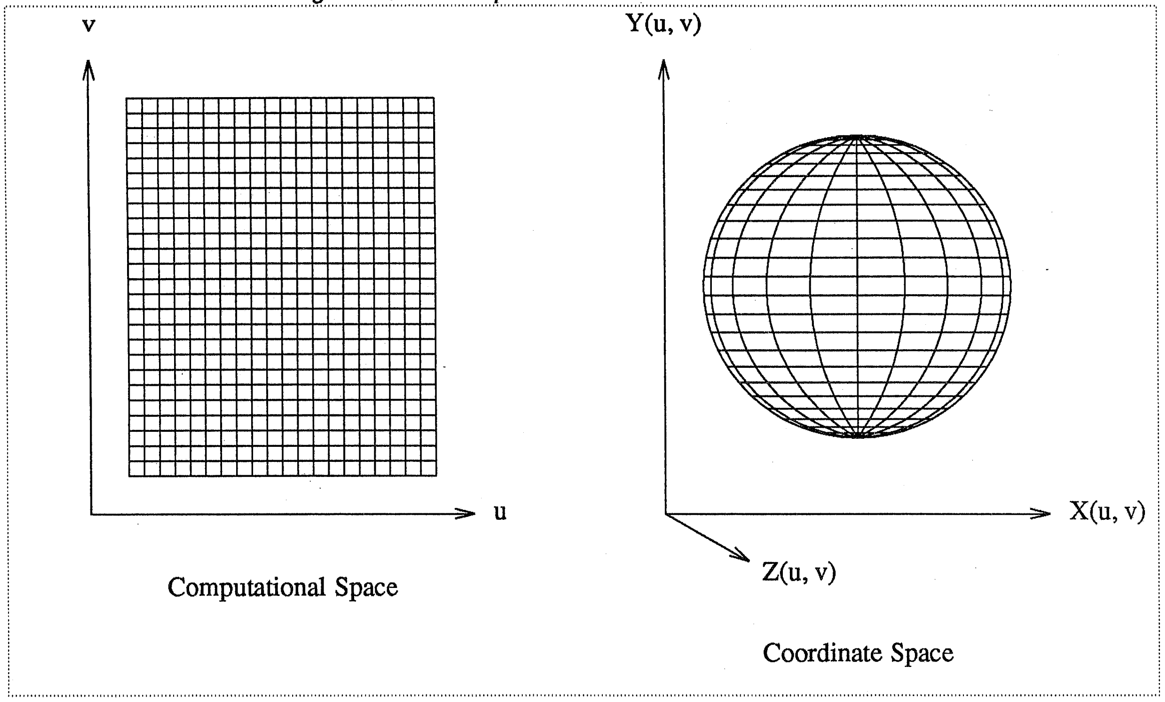
Example 3

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:

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

The field is irregular, with 500 *X* coordinates, 500 *Y* coordinates, and 500 *Z* coordinates. Each cell in coordinate space has curvilinear bounds. Figure 3 shows the mapping between computational and coordinate space.

Figure 2-3. Example 3



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:	20×25
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:

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

The field is uniform.

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:	256×256
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 256×256 pixels. Each data element is a single byte. The field consists of 6553600 elements:

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

The field is uniform.

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:	256×256×100
Number of computational values:	1*256*256*100 = 6553600
Mapping type:	Uniform
Number of coordinate dimensions:	3
Number of coordinate values:	0

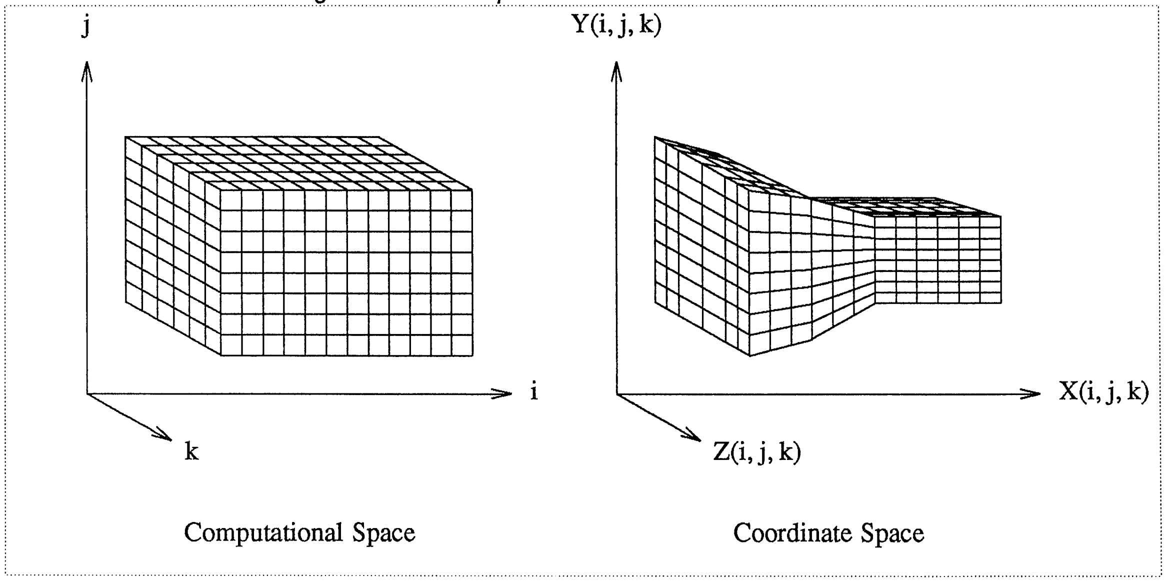
Example 6

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:

$$\left\{ V_n(X(i,j,k),\ Y(i,j,k),\ Z(i,j,k)),\ i{=}1,12,\ j{=}1,8,\ k{=}1,6,\ n{=}1,5 \right\}$$

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 2-4. Example 6



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:	12×8×6
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 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.

❑ 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 the appendix "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 rectilinear or irregular field, an array of floating-point values representing the coordinates of the field.

  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 the appendix "AVS C Language Field Macros".

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 example, 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.

If a module accepts some subtypes of fields but not all, it should check the inputs and signal an error if the input is of a type it doesn't accept. That is, if a module accepts 2D and 3D scalar uniform fields of floating-point numbers, it should declare the input as "field scalar uniform float", and then the module's computation routine should check the number of dimensions in the input field.

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	"scalar", "1-vector"
	n	"*n*-vector"
Data Type	byte	"byte", "char"
	integer	"integer", "int"
	real	"real", "float"
	double	"double", "real*8"
Number of Coordinates	*n*	"*n*-coord", "*n*-space"
Mapping Type	uniform	"uniform"
	rectilinear	"rectilinear"
	irregular	"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 argument, 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 laid out as follows (using **AVSfield_char** 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, true if field is single malloc */
  int uniform;           /* mapping type: UNIFORM, etc. */
  int *dimensions;       /* dimension along each axis; length is ndim */
  float *points;         /* coordinates for nonuniform fields */
  unsigned char *data;   /* the field data itself as chars (bytes)*/
} AVSfield_char;
```

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, 12×8×6. 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)
uniform        IRREGULAR
dimensions     dims[3] = { 12, 8, 6 }
points         coords[3][6][8][12]
data           data[6][8][12][5]
```

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 the appendix "AVS C Language Field Macros" for more information.

Manipulating Fields from FORTRAN

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. For example, a computation routine that takes as its first input port a "field 3D 3-vector real rectilinear" is defined as follows:

```
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 2-4. Field Arguments to FORTRAN Routines

Field Component	**Port Specification**	**Input or Parameter Argument(s)**	**Output Argument(s)**
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, ...
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(*)
	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)
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.)

Creating Fields

AVS provides routines for creating a field from a data array and possibly an array of coordinates. The general routine is **AVSbuild_field**. **AVSbuild_2d_field** and **AVSbuild_3d_field** are simpler interfaces for scalar uniform fields with floating-point data. These routines return a pointer to an **AVSfield** structure. See the appendix "AVS Routines" for more information.

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 would be declared as "field 1D 0-vector 3-coordinate irregular". Such a field has no data array. The number of dimensions should still be declared to be 1, and the one dimension of the field in computational space is still 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.

Byte	Component
0	blue
1	green
2	red
3	alpha

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

Volume Data

AVS generally represents volumes as 3D scalar fields of bytes, usually declared as "field 3D scalar byte". The value of each byte is between 0 and 255 inclusive. Some modules use the field data as indices into colormaps. For many AVS modules that deal with volumes, each dimension of the field must be less than 256.

Colormaps

A colormap is a transfer function that assigns a color to each integer 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 declaring a series of parameters:

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

      INTEGER size
      REAL lower, upper
      REAL hue(256), sat(256), val(256), alpha(256)
```

A FORTRAN routine can output a colormap as follows:

```
INTEGER FUNCTION my_module(size, lower, upper, phue, psat, pval, palpha)
      INTEGER size
      REAL lower, upper
      POINTER (phue,hue), (psat,sat), (pval,val), palpha,alpha)
      REAL hue(256), sat(256), val(256), alpha(256)
```

Note the use of POINTER variables to supply an extra level of indirection.

Geometries

A geometry object describes changes to the geometry of a particular scene that is represented by a module input or output. AVS allows a user module to create geometry objects as outputs. It is possible for a module to use geometry objects as inputs, but this release of AVS does not support writing user modules that do this. Geometry output is typically used as input to an AVS-supplied renderer module such as the geometry viewer.

A geometry data object is called an *edit list*. This is 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 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").

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:

- ❑ The geometric data definining 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

Manipulating Edit Lists

Each time a module is invoked, it starts 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; see the manual page **geom**(3V). 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 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)
```

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 pixel values and resides in client memory.)

A pixel map 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 a boolean flag indicating whether or not the pixmap is a buffer drawable created by **XdbUpdateWindows**(3H). If the pixmap is a buffer drawable, a routine should use **XdbUpdateWindows** instead of **XCopyArea** to copy the pixmap to the window.

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 you should use XdbUpdateWindows */
                 /* to update */
} AVSpixdata;
```

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

Stardent

3

AVS Modules

Table of Contents

Chapter 3
AVS Modules

Modules

A module is a 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 a new module.

Module Components

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 workspace.

Type

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

Data	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.
Renderer	A module that renders or stores AVS data, usually 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. 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 input data and therefore do not have input ports. Renderer modules often display or write their own output data and therefore do not have output ports.

When an instance of a module exists in AVS, each input port can be connected to an appropriate output port of another module, and each output port can be connected to an appropriate input port of another module. A pair of ports can be connected 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". An output port cannot be connected to an input port of the same module.

For some input ports, a connection to an output port of another module is required before the module can be invoked. For other input ports, a connection is optional.

Parameters

A *parameter* is a variable that has a constant value during an invocation of the 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 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 the appendix "AVS Routines".

Each parameter is usually connected to a widget that enables the user to change the value of the parameter between module invocations. A widget is a virtual input device such as a dial, a file browser, or a Spaceball. A parameter can be connected only to a widget that is compatible with the parameter's 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 the appendix "AVS Routines".

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 the appendix "AVS Routines".

When appropriate, a module can alter the current value or bounds of a parameter dynamically. AVS then updates any widget associated with the parameter. See the documentation for the **AVSmodify_parameter** routine in the appendix "AVS Routines".

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. Following are the functions and their purposes:

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

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. AVS invokes this procedure when the user makes an instance of the module, as by moving the module icon from the Network Editor module palette to the workspace. The initialization function may take such actions as allocating memory or creating a window. The initialization function has no arguments and returns no meaningful value.
- A module may have a *destruction* function. AVS invokes this procedure when the user destroys the module, as by moving the module icon from the Network Editor workspace to the "hammer" icon. The destruction function may take such actions as freeing memory or destroying a window. The destruction function has no arguments and returns no meaningful value.

The Description Function

The description function describes the module's name, type, inputs, outputs and parameters using a set of library functions. A C language file can contain more than one module and therefore more than one description function. The file must contain a routine called **AVSinit_modules** that refers to all the description functions in the file. A FORTRAN file can contain only one module and therefore only one description function. A FORTRAN description function must be named **AVSINIT_MODULES**. The description function has no arguments and returns no meaningful value.

Following is the C language version of an example description function for a module to compute the threshold of a 3-dimensional scalar field. The threshold module is created with one input port, one output port, and two parameters.

```
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);
}
```

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
```

The most common steps in a description function are as follows:

- ❑ 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.

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.
- ❑ 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. Following is a C language example:

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

Following is a FORTRAN example:

```
      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. The computation function should be declared to return an integer. It should return a value of 0 to indicate an error. In this case 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 arguments are presented in the same order as the arguments to a C language computation function, but each port or parameter can generate more than one argument to the computation routine. The number of arguments for each port or parameter depends on the data type declared in the description function and, for a port, on whether the port is input or output. For example, an input port declared as "field 3D scalar uniform" in the description function generates four arguments to a FORTRAN computation routine. For more information on arguments to FORTRAN computation functions, see the "AVS Data Types" chapter.

For a C language computation function, an argument that represents an input port or a parameter is usually passed as a pointer to an object of the C storage type that corresponds to the AVS data type of the port or parameter declared in the description function. 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. 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.

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.

A C language 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, a C language 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 meaningful value. 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. 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);
}
```

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 executable file has only one module and one main program. A FORTRAN module does not have a separate **AVSINIT_MODULES** function; instead, its description function is itself named **AVSINIT_MODULES**.

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, 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*. 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 any.
- ❑ Enters a server routine that loops indefinitely, waiting for remote procedure calls from AVS and then executing the 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 any, and then terminate 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 and 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 for the module.
- ❑ 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_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 omits the call to **AVScorout_wait.** If a coroutine module computes continuously, it should usually provide a parameter to allow the user to stop the computation. The module should check the value of this parameter after the call to **AVScorout_input.**

After calling **AVScorout_init**, a coroutine module might ensure that input is available before beginning computation. It can do this in a loop, calling **AVScorout_wait** and then **AVScorout_input** until the input data is not null.

A coroutine module can also use the **AVScorout_exec** routine. This routine waits until the flow executive has stopped running and then returns. This allows the module to ensure that the network has processed the output of each computational iteration before sending more output so that no data is lost.

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 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 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**. 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.

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 likely that when the module computation function is executed, some ports or parameters have not changed since the previous execution of the computation function. The module might be able to avoid some computation for 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 is executed, 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. (That file can contain more than one C language subroutine module.) The source code can be 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.

A C language 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, a C language programmer supplies a function called **AVSinit_modules** to invoke the description functions for all modules in the file.

A FORTRAN executable file has only one module and one main program. A FORTRAN module does not have a separate **AVSINIT_MODULES** function; instead, its description function is itself named **AVSINIT_MODULES**.

Writing Coroutines

A basic coroutine module as written by a programmer consists of a main program and a description function, with optional initialization and destruction functions. 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 language and FORTRAN programs. Some include files are needed for nearly all modules, while others are needed only if the module is using data of a particular type. The summary in the appendix "AVS Routines" lists any include files needed for each 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 language and FORTRAN programs can refer to an include file using the following syntax:

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

C Language Include Files

Most C language modules should include a single file, *avs.h*. This file contains definitions not specific to particular data types. 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.

FORTRAN Include Files

Most FORTRAN modules should include a single file, *avs.inc*. This file contains definitions not specific to particular data types as well as definitions needed when using data of most AVS types.

FORTRAN modules that use geometries should include the file *geom.inc*.

Compiling and Linking Modules

To compile and link a module, use **cc**(1) for C language modules and **f77**(1) for FORTRAN 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**(3V) library, which requires linking with a number of library files. For details see the **geom**(3V) manual page.

Converting an Existing Application to a Module

Many existing simulations, batch data converters, and other scientific applications can be converted 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:

- ❑ 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.
- ❑ Choose the AVS data type that is most appropriate for each input, output, and parameter.
- ❑ Write a description function to declare the module and its inputs, outputs, and parameters.
- ❑ 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.
- ❑ 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.
- ❑ Ensure that the program allocates and frees memory for AVS outputs where necessary. The **AVSinitialize_output** and **AVSautofree_output** routines make this task easier.
- ❑ Use **AVSmessage** or its variants to handle errors in the program.
- ❑ Ensure that the program uses appropriate AVS include files. Most C language programs should include *<avs/avs.h>* and any files needed for particular data types. Most FORTRAN programs should include *<avs/avs.inc>*.

- ❑ 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 **dbx**(1) debugger. A common sequence for using the debugging facility is as follows:

- ❑ Compile the module using the **–g** option to **cc**(1) or **f77**(1). This causes the compiler to generate information needed by **dbx.**
- ❑ In an **xterm** window, invoke the command **avs_dbx.** The last argument to the command is the name of the executable file that contains the module you want to debug. In the **xterm** window, the **avs_dbx** command invokes **dbx** with the executable pathname as the last argument. When **dbx** has started, you can set breakpoints and invoke other **dbx** commands. Do not type `run` yet.
- ❑ 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.
- ❑ Create an instance of the module. In the Network Editor, move the module icon from the module palette to the workspace.
- ❑ In the **xterm** window, AVS prints the message:

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

  Type `run` to cause **dbx** to run the executable file that contains the module.
- ❑ 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 **dbx.** Interaction with **dbx** takes place in the **xterm** window.

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

avs_dbx [**–id**] [**–mod** *module_name*] [*dbx_options*] *file*

The *file* argument is the name of the executable file that contains the module. If more than one option is present, the options must be given to **avs_dbx** in the order listed here. The following options are available:

–id	If this option is present, the module runs under **dbx** during an invocation of the module for *identification* as well as during an invocation of the module for *instantiation*. When you use the **Read Module** Network Editor command, AVS invokes the module for identification. The module description function is called, and the module declarations are conveyed to AVS. The module's process then exits. If the **–id** option is present this invocation of the module runs under **dbx**; by default it does not run under **dbx.** 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 **dbx** whether or not the **–id** option is present. During this invocation the description function is called again; the description function then runs under **dbx** even if the **–id** option is not present.
–mod *module_name*	Some executable files may contain more than one module. If the **–mod** option is present, only the module named *module_name* is run under **dbx**. By default all modules in the file are run under **dbx.** If the **–id** option is also present, the description functions for all modules in the executable file are run under **dbx** when the executable is invoked for identification. The description functions for all modules in the executable file are always run under **dbx** when the module is invoked for instantiation.
dbx_options	These options are passed to **dbx.** For more information, see the **dbx**(1) manual page.

Following are some notes on using **avs_dbx**:

- ❑ You can run more than one module under **dbx** by invoking **avs_dbx** in multiple **xterm** windows. However, if you want to run a module under **dbx**, you cannot make more than one instance of the same module.
- ❑ To run a module under **dbx**, you must invoke **avs_dbx** before you make the instance of the module, as by 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 type `run` under **dbx** 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 **dbx**, you cannot manipulate any widgets for that module or make any Network Editor connections to or from that module until after you have typed `run` under **dbx** or have continued from a **dbx** breakpoint.
- ❑ Avoid **dbx** commands that might disrupt the synchronization of the module's execution with AVS execution. For example, do not use the **dbx** `run` command to restart a module from a breakpoint; use `continue` instead.

- 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 **dbx**, you must reinvoke **avs_dbx** before making an instance of the recompiled module.
- You can use the **avs_dbx** command with both subroutine and coroutine modules.

Module Examples

The appendix "Examples of AVS Modules" contains example source code for several AVS modules. Source code for these and other examples is also available in the directory */usr/avs/examples*.

Appendix A. AVS Routines

Routines for Module Description Functions

AVSset_module_name

C:

```
#include <avs/avs.h>
AVSset_module_name(name, type)
    char        *name;
    int         type;
```

FORTRAN:

```
AVSSET_MODULE_NAME(NAME, TYPE)
    CHARACTER*(*)  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:

Module Type	C Constant	FORTRAN String
Data	MODULE_DATA	'data'
Filter	MODULE_FILTER	'filter'
Mapper	MODULE_MAPPER	'mapper'
Renderer	MODULE_RENDER	'renderer'

The module name appears in the module icon and other portions of the Network Editor and Application Builder user interface. The module type determines the category in the Network Editor module palette in which the module icon appears.

AVScreate_input_port

C:

```
#include <avs/avs.h>
int AVScreate_input_port(name, type, required)
    char        *name, *type;
    int         required;
```

FORTRAN:

```
#include <avs/avs.inc>
AVSCREATE_INPUT_PORT(NAME, TYPE, REQUIRED)
    CHARACTER*(*)  NAME, TYPE
    INTEGER        REQUIRED
```

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:

Data Type	*type* String
byte	"byte"
integer	"integer"
real	"real"
string	"string"
field	"field"
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 *required* argument is a constant indicating whether or not a connection to the port is required before the module can be invoked. Possible values are **REQUIRED**, meaning that a connection is required, and **OPTIONAL**, meaning that a connection is not required.

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:

AVSCREATE_OUTPUT_PORT(*NAME, TYPE***)**
    CHARACTER*(*) *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 **AVSinitialize_output**.

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**.

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**.

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_init_proc

C:

```
AVSset_init_proc(init_func)
    int              (*init_func)();
```

FORTRAN:

```
AVSSET_INIT_PROC(INIT_FUNC)
    EXTERNAL     INIT_FUNC
```

This routine declares the 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.

AVSset_destroy_proc

C:
AVSset_destroy_proc(*destroy_func***)**
    int *(*destroy_func)();*

FORTRAN:
AVSSET_DESTROY_PROC(*DESTROY_FUNC***)**
    EXTERNAL *DESTROY_FUNC*

This routine declares the 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.

AVSadd_parameter

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

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

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 **ints** 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. Each value must fit into an integer-size memory slot or must be a pointer to a larger memory allocation. Values representing **floats** in C must be pointers to allocated memory. The routine **AVSadd_float_parameter** handles this allocation automatically.

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.

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*(*)**
"choice"	**char** *	**CHARACTER*(*)**
"colormap"	**AVScolormap** *	—
"field"	**AVSfield** *	—
"delta_matrix_4x4"	**AVSfield** *	—

AVS passes fields to FORTRAN computation functions as multiple arguments; see the "AVS Data Types" chapter.

Following are notes on some of these types:

integer The *minval* argument is the minimum value; the *maxval* argument is the maximum value.

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. Setting a value of 0, using **AVSmodify_parameter**, clears the counter. 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 must be **NULL** in C (0 in FORTRAN) or an allocated string. 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.

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 "!".

colormap The *minval* and *maxval* arguments are ignored. A FORTRAN computation routine cannot take a colormap as a parameter argument.

field The only supported field type is "field 2D scalar real". This is used for handling 4×4 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 × 4 delta matrices used by the Spaceball. The *minval* and *maxval* arguments are ignored.

This routine returns an integer parameter identifier that is used as an argument to some other AVS routines, such as **AVSconnect_widget**.

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 Stardent 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.

AVSconnect_widget

C:

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

FORTRAN:

AVSCONNECT_WIDGET(*PARAM_NUM, WIDGET_TYPE***)**
    INTEGER   *PARAM_NUM*
    CHARACTER*(*) *WIDGET_TYPE*

This routine declares a preference that a parameter for a module being defined in the current description function be connected to a specified widget. A parameter can be connected only to a widget that is compatible with the parameter's type. If this routine is called with an impermissible widget type, AVS ignores the preference and issues a warning.

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.**

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 typein 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.
	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.
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.
colormap	color_editor	Colormap editor.
field	track	Cursor-tracking virtual trackball.
delta_matrix_4x4	spaceball_client	Spaceball.

AVSadd_parameter_prop

C:

```
AVSadd_parameter_prop(param_num, prop_name, prop_type, prop_value)
    int          param_num;
    char         *prop_name, *prop_type;
    int          prop_value;
```

FORTRAN:

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

Property Name	Property Type	C Data Type	FORTRAN Data Type	Widget Types
title	string	**char** *	**character*(*)**	dial, idial, slider, islider, toggle, tristate, oneshot, radio_buttons
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

width	integer	**int**	**integer**	toggle, tristate, oneshot, typein, text, browser, text_browser, radio_buttons
height	integer	**int**	**integer**	toggle, tristate, oneshot, typein, browser, text_browser, radio_buttons
columns	integer	**int**	**integer**	radio_buttons

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 10 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.

AVSmodify_parameter

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

FORTRAN:
#include *<avs/avs.inc>*
AVSMODIFY_PARAMETER(*NAME, FLAGS, INIT, MINVAL, MAXVAL***)**
    CHARACTER*(*) *NAME*
    INTEGER *FLAGS, 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.

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; for more information, see the documentation for **AVSadd_parameter**. 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 — floats and strings, in particular.)*

*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; perform further computations with the copy, not the original.*

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 Stardent C, when a float is passed as an argument it is converted to a **double**.

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

WARNING *See WARNING under **AVSmodify_parameter** above.*

AVSchoice_number

C:
AVSchoice_number(*name, string***)**

```
     char          *name, *string;
```

FORTRAN:
AVSCHOICE_NUMBER(*NAME, STRING***)**

```
     CHARACTER*(*)  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 is the string passed to the computation function as the value of the parameter.

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.

Routines for Coroutine Modules

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.

AVScorout_input

C:

```
int AVScorout_input(input1, input2, ..., param1, param2, ...)
    char           **input1, **input2, ...;
    int            *param1, *param2, ...;
```

FORTRAN:

```
AVSCOROUT_INPUT(INPUT1, INPUT2, ..., PARAM1, PARAM2, ...)
    POINTER         (INPUT1, I1), (INPUT2, I2), ...
    CHARACTER*(*)   I1, I2, ...
    INTEGER         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).

The routine returns 0 if a required input or parameter is missing. Otherwise, it returns the number of inputs and parameters supplied.

AVScorout_output

C:
AVScorout_output(*output1*, *output2*, ...)
    char   **output1*, **output2*, ...;

FORTRAN:
AVSCOROUT_OUTPUT(*OUTPUT1*, *OUTPUT2*, ...)
    CHARACTER*(*)   *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.

If the user has disabled the module or the flow executive, this routine may hang for an arbitrary time before returning.

AVScorout_wait

C:
AVScorout_wait()

FORTRAN:
AVSCOROUT_WAIT()

This routine waits until the user changes a parameter value or until an upstream module sends more input. It then returns.

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.

Routines for Selective Computation

AVSinput_changed

C:

```
int AVSinput_changed(port_name, i)
    char            *port_name;
    int             i;
```

FORTRAN:

```
AVSINPUT_CHANGED(PORT_NAME, I)
    CHARACTER*(*)  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.

AVSparameter_changed

C:

```
int AVSparameter_changed(param_name)
    char            *param_name;
```

FORTRAN:

```
AVSPARAMETER_CHANGED(PARAM_NAME)
    CHARACTER*(*)  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.

AVSmark_output_unchanged

C:

```
AVSmark_output_unchanged(port_name)
    char            *port_name;
```

FORTRAN:

```
AVSMARK_OUTPUT_UNCHANGED(PORT_NAME)
    CHARACTER*(*)  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.

Routines for Creating Fields

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(*)
```

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*

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*

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.

AVSfield_alloc

C:
#include <*avs/field.h*>
char * **AVSfield_alloc**(*template, dims*)
  AVSfield **template;*
  int **dims;*

This routine creates and allocates memory for a field. It returns a pointer to a **char**, which should be cast to a pointer to an **AVSfield**.

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.

There is no FORTRAN equivalent for this routine.

AVSfield_make_template

C:
#include *<avs/field.h>*
AVSfield_make_template(*field_in, template*)
    AVSfield   **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));
```

There is no FORTRAN equivalent for this routine.

AVSfield_copy_points

C:
#include *<avs/field.h>*
AVSfield_copy_points(*field_in, field_out*)
    AVSfield   **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.

There is no FORTRAN equivalent for this routine.

Routines for Module Initialization

AVSinit_modules

C:
AVSinit_modules()

FORTRAN:
AVSINIT_MODULES()

This routine is defined by the AVS programmer. 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**. The definition differs depending on whether the module source is C or FORTRAN:

- In C, each 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, each file can define only one module. The module description function itself should be called **AVSINIT_MODULES.**

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

AVSmodule_from_desc

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

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

For modules written in C, each file can define more than one module. The programmer-supplied routine **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.

There is no FORTRAN equivalent for this routine. In FORTRAN, the module description function itself is called **AVSINIT_MODULES**.

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.

For modules written in C, each file can define more than one module. The programmer-supplied routine **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.

There is no FORTRAN equivalent for this routine. In FORTRAN, the module description function itself is called **AVSINIT_MODULES.**

Routines for Handling Errors

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;*
char	**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*(*)	*VERSION*
INTEGER	*SEVERITY*
CHARACTER*(*)	*MODULE, FUNCTION_NAME*
CHARACTER*(*)	*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 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 *avs_message.log* in the current working directory. The log file may contain additional information beyond that presented in the dialog box, including the *version* string. When AVS starts up, any existing *avs_message.log* file in the current working directory is renamed to *avs_message.log~*.

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*(*) *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.

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*(*) *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.

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*(*) *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.

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*(*) *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*(*) *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.

Appendix B. AVS C Language Field Macros

Macros for Obtaining Field Dimensions

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* coordinate 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

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*.

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()
{
        int 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 field of
C 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
      external threshold
      external AVScreate_input_port, AVScreate_output_port
      integer AVScreate_input_port, AVScreate_output_port

C Set the module name and type
      call AVSset_module_name('ex2-threshold', '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

      call AVSadd_parameter('min', 'float', 0.0, 0.0, 255.0)
      call AVSadd_parameter('max', 'float', 255.0, 0.0, 255.0)

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 represent
C the input field: f, nx, ny, nz.  The second four arguments represent
C the output field: gp, mx, my, mz.  Since we used AVSinitialize_output
C in the description routine, gp, mx, my, and mz will already have
C the appropriate values.  Note that for output field, we set up
C the pointer declaration for the data field.  The last two arguments
C are the minimum and the maximum, read from dials manipulated by
C the user.  Note that they are presented to the subroutine in the
C 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)
      pointer (gp, g)
      dimension g(mx, my, nz)
      real fmin, fmax

      do k = 1, nz
         do j = 1, ny
            do i = 1, nx
               if (f(i, j, k) .gt. fmax) then
                  g(i, j, k) = 0.0
               elseif (f(i, j, k) .lt. fmin) then
                  g(i, j, k) = 0.0
               else
                  g(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 whenver 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 polygon_compute();	/* declare the compute function (below) */
	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 for Your Modules and Networks

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.

You can also arrange for your help files to be visible to the **man**(1) shell command.

Help Files — Format and Naming Conventions

Each help screen in AVS is implemented as an ASCII text file, with a *.txt* filename suffix. 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.

NOTE *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.*

The name of the help file must match the name of the associated module or network. Replace SPACE characters in the module or network name with underscores. For example:

Module/Network Name	**Help Filename**
clarify edge	clarify_edge.txt
easy vu 2	easy_vu_2.txt

You can include comment lines in your help files. Any line that begins with a period (**.**) character is suppressed when the file is displayed.

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, we describe integrating the files into the AVS help facility. Then, we describe 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. At many installations, system areas are not 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.

Store your help files in a separate area, e.g. */usr/local/avs/helpfiles*. When you invoke AVS, use the environment variable AVS_HELP_PATH to point to your help data. For example, in the C shell:

```
% setenv AVS_HELP_PATH /usr/local/avs/helpfiles
% avs
```

You can include more than one directory in AVS_HELP_PATH, separating the entries with colon (**:**) characters, e.g.:

```
% setenv AVS_HELP_PATH /usr/john/avs/:/usr/cory/avs
```

You may want to set the environment variable in your shell startup file (*.cshrc* for the C shell, *.profile* for the Bourne shell).

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 that filename in the entire directory hierarchy under the first entry in AVS_HELP_PATH. If such a file exists, 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.

The **Help** button in the Network Construction window does *not* use AVS_HELP_PATH — it always looks in */usr/avs/runtime/help* for help files. Likewise for the Geometry Viewer.

The Image and Volume Viewers *do* use the first entry in AVS_HELP_PATH as the initial help directory.

Man Command

The **man**(1) command can be used to view the help files for the AVS modules. These files appear to the **man** command to be in directory */usr/man/catman/man6*. This name is a symbolic link to the actual location of the module help files, */usr/avs/runtime/help/modules*.

Here is a suggested procedure for making manual pages for your own modules visible to the **man** command:

- ❑ Create the manual pages as ASCII text files (with no TAB characters) in a "non-system" location, e.g. */usr/local/avs/helpfile*. You can use the *.txt* filename suffix. (Actually, the suffix is immaterial.)

- ❑ In the manual page area, create a symbolic link to this location in the man page area, e.g:

  ln -s /usr/local/avs/helpfiles /usr/man/catman/man6L

  Here, Section "6L" has the mnemonic meaning "local version of Section 6".

- ❑ Use this form of the **man** command to view the module help files:

  man 6L clarify_edge