IRIS Explorer Module Writer’s Guide
Release 3

Document Number NP2873
ISBN 1−85206−111−1

Introduction

"'The question is,' said Humpty Dumpty, 'which is to be master – that's all.'"

Footnote: *Through the Looking Glass*, by Lewis Carroll

The *IRIS Explorer Module Writer's Guide* contains the information you need to master the tools for constructing and installing your own custom–built modules in IRIS Explorer. The options for taking control at different stages of the module–building process are numerous, and you can choose your level according to your current needs.

If your needs are fairly simple, you can take full advantage of the Module Builder's capabilities for automating the process and do very little work yourself, beyond supplying a basic algorithm. On the other hand, you may want to exploit those aspects of the IRIS Explorer product that let you as module writer control the process almost entirely.

To let the IRIS Explorer program do the work is very convenient, and most of the time it is sufficient, but it does set limits on what you can achieve. To be master yourself takes more effort but offers in return access to all the power and flexibility of the IRIS Explorer environment.

About This Guide

The *IRIS Explorer Module Writer's Guide* describes how to design and build IRIS Explorer modules using the Module Builder.

It provides information about using the built–in IRIS Explorer data types and about defining new data types for your particular needs. There are also sections on how to customize your build environment and set up special subdirectories for module development.

This guide is written primarily for programmers who support computational scientists and engineers in the fields of fluid dynamics, chemistry, meteorology, cosmology, physics, and mathematics. It addresses users who will write module suites for other people to use in the Map Editor.

However, any users who wish to extend the functionality of IRIS Explorer can write their own modules using the Module Builder.

The guide assumes you:

- can program in either C or Fortran
- understand the UNIX operating system and file structure
- are familiar with the X Window System™ and Motif™

In general, the user interface of the Module Builder adheres closely to the Motif standard.

Contents of This Guide

The *IRIS Explorer Module Writer's Guide* has ten chapters and two appendices. They cover the following topics:

- **Chapter 1, "Building a Module"**, steps the module writer through the module–building process in tutorial form.
- **Chapter 2, "Understanding the Module Builder"**, explains how to use the Module Builder to create the module resource file.
- **Chapter 3, "Using the Lattice Data Type"**, describes the lattice data type and explains how it is used in IRIS Explorer modules.
- **Chapter 4, "Using the Pyramid Data Type"**, describes the pyramid data type and explains how it is used in IRIS Explorer modules.
- **Chapter 5, "Using the Geometry Data Type"**, describes the geometry data type and explains how it is used in IRIS Explorer modules.
- **Chapter 6, "Using the Pick Data Type"**, describes the pick data type and explains how it is used in IRIS

Explorer modules.

- **Chapter 7, "Using the Parameter Data Type"**, describes the parameter data type and explains how it is used in IRIS Explorer modules.
- **Chapter 8, "Creating User−defined Data Types"**, outlines the IRIS Explorer typing language (ETL) syntax and explains how to build user−defined data types.
- **Chapter 9, "Advanced Module Writing"**, provides information on topics such as hook functions, shared module executables, and reference counting.
- **Chapter 10, "Module Prototyping with Shape"**, outlines the Shape language syntax and explains how to write Shape scripts for interpreter modules.
- **Appendix A, "Using Makefiles"**, describes how to create user Makefiles.
- **Appendix B, "The Firing Algorithm"**, gives an overview of the IRIS Explorer execution model.

All the chapters contain sample code and refer the reader to source directories in which modules that use the sample code reside. All example code may be found in directories below */usr/explorer/src/MWGcode.* You can copy this code to your own work area, build the modules and run them in the Map Editor.

The companion volume to this guide is the *IRIS Explorer User's Guide*, which describes the Map Editor and how you launch modules and create maps and applications in it. In addition, documentation for all IRIS Explorer modules is provided on−line as *man* and *help* pages for each module. Similarly, the specification for each of the IRIS Explorer Application Programming Interface (API) routines is available via *man*. More information on this is provided in the on−line *IRIS Explorer Reference Pages.*

All of the on−line documentation is available in hard−copy form as the three volumes:

- *IRIS Explorer User's Guide*,
- *IRIS Explorer Module Writer's Guide*,
- *IRIS Explorer Reference Pages*, which contains all the information from the *man* pages for the modules and API routines,

Contact your IRIS Explorer supplier for more details.

Chapter 1

Building a Module

This chapter takes you step by step through the process of creating a simple module in the Module Builder. The proposition is to build a module that acts as a channel selector. This module will take in an array containing a vector of values per sample, and output an array with a scalar at each sample. The user manipulates a widget to select which scalar, or channel, is output.

The example shows how to build a module with a user function written either in C or in Fortran. The process is identical for both, but some menus differ, depending on the language used. These differences are called out clearly in the text.

The information on building a module spans two chapters:

- Chapter 1 takes you through the required steps with some basic explanation.
- Chapter 2 provides conceptual and background information for each step.

If you want more information than is given in Chapter 1 about a particular step, refer to the parallel section in Chapter 2.

1.1. Introduction to Module building

The IRIS Explorer product uses modules to carry out operations on data in order to visualize the data in a way that has meaning to the observer. The modules supplied with IRIS Explorer offer a range of functions, but many users will want to construct their own modules, providing a more specific function or greater power than already exists.

The Module Builder is a tool that lets you build your own IRIS Explorer modules, either by modifying and renaming existing IRIS Explorer modules or by creating new ones. The great virtue of the Module Builder is its graphical user interface, which lets you build a basic module with no programming beyond that needed to write the computational function.

The module−building process has three main stages:

- Defining the internal structure, or "engine":
  - Creating a user function or subroutine
  - Defining the input and output ports
  - Defining the function arguments and their calling sequence
  - Defining the relationships between the inputs and outputs and the function arguments
- Defining the external structure, or user interface:
  - Laying out a module control panel
  - Associating input parameters with *widgets*, or control mechanisms
  - Creating menu bar items
- Building and installing the module in IRIS Explorer:
  - Setting environment variables to define the build directory
  - Selecting build options
  - Writing hook functions (if required)
  - Running the *build* command

Once the module is built, you can wire it into a map in the Map Editor and use it like any IRIS Explorer module.

1.2. Creating the User Function

The user function is the computational function or subroutine that acts on the data fed into the module through its input ports and produces new or changed data on the output ports. It provides the core functionality of the module.

1.2.1. Designing the Example Module

You are designing a module that will select one data variable, or channel, from an array that has a number of data variables. Arrays in IRIS Explorer are called lattices. The module is a simple selector with no limits on the range or data type. Here is a brief description of the structure you need in the new module, which you will call *ChannelSelector*.

It should take in all the data variables from the input lattice and put out values for only one data variable. It should have a widget for selecting a particular data variable. It needs an input port, an output port, some defined variables for the lattice data and coordinate values, and a computational function to select out the data values for a specific channel.

1.2.2. Writing the User Function

The example shows how you can create a channel selector module with a C function or a Fortran subroutine. The module accepts input data in the form of an IRIS Explorer lattice. You need not be concerned with lattices now, except to note that as a matter of convention, the variable names in this function correspond closely with the variable names in the *cxLattice* data type structure.

To find out more about lattices, read **Chapter 3, "Using the Lattice Data Type"**.

Example Code and Modules

Working example code for this module can be found in */usr/explorer/src/MWGcode/Tutorial/C/channel.c* and */usr/explorer/src/MWGcode/Tutorial/Fortran/channelselect.f*. There are also complete module resources for this code, called *ChannelSelector.mres* in the same directories. You can open the respective files to see how yours should look.

In the following sections you are taken through the steps required to construct a module using existing code, while imposing as few restrictions on the module as is feasible. This situation will occur frequently if you reuse existing and well tested code, for example from a library. Some steps might well be different if you decide to write code specifically for the module. This is indicated in the text.

The C User Function

The function called *chan* in the file *channel.c* serves as the user function in this module. It is written in C. The code shows the source of the function arguments used later in this chapter.

Note that the code is written for *float* data only. Through the specification in the module resources file, IRIS Explorer can ensure that this code may also be used for other data of a different type. To find out more about automatic type coercion see **"Automatic Type Coercion of Arrays"** in Chapter 2.

```
/* User function to select a single data channel
 * from a lattice of type float
 *
 * The Iris Explorer wrapper code will ensure that lattice data
 * of a different type will be converted to float before being
 * passed to this function, and the output lattice converted from
 * float to the type of the input lattice.
 *
 * This feature enables you to write a much more general module that
 * can deal with all data types, not just float. This will be of
 * particular use when you need to utilise already existing code.
 */
long sizes(long nDim, long *dims);

void chan(long nDim, long *dims, long *nDataVar, long which,
      float *dataIn, float *dataOut)
{
```

```
  int i, numElems, sel;

  /* Number of data channels in the input lattice */
  numElems = sizes(nDim, dims);

  /* Use a legal range for the data */
  sel = which - 1;
  if (sel < 0)
    sel = 0;
  else if (sel >= (*nDataVar))
    sel = (*nDataVar) - 1;

  /* Copy the data to the output lattice */
  for (i = 0; i < numElems; i++)
    dataOut[i] = dataIn[i*(*nDataVar)+sel];
  *nDataVar = 1;
  return;
}

long sizes(long nDim,long *dims)
{
  long foo, i;

  foo = 1;
  for (i = 0; i < nDim; i++)
    foo *= dims[i];
  return(foo);
}
```

The Fortran User Function

The Fortran subroutine called *chan* in the file *channelSelect.f* is the user function in this module. This code fragment shows the source of the function arguments used later in this chapter

Footnote: The Fortran examples in this document were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

.

Note that the code is written for REAL data only. Through the specification in the module resources file, IRIS Explorer can ensure that this code may also be used for other data of a different type. To find out more about automatic type coercion see **"Automatic Type Coercion of Arrays"** in Chapter 2.

```
C      Subroutine to read in a two-dimensional array
C      and output a one-dimensional array, both of type REAL
C
C      The IRIS Explorer wrapper code will ensure that lattice data
C      of a different type will be converted to REAL before being
C      passed to this function, and the output lattice converted from
C      REAL to the type of the input lattice.
C
C      This feature enables you to write a much more general module that
C      can deal with all data types, not just REAL. This will be of
C      particular use when you need to utilise already existing code.
C
       SUBROUTINE CHAN(NDIM,DIMS,NVARS,WHICH,DATAIN,DATOUT)
C      .. Scalar Arguments ..
       INTEGER         NDIM, NVARS, WHICH
```

```
C     .. Array Arguments ..
      REAL            DATAIN(NVARS,1), DATOUT(1)
      INTEGER         DIMS(NDIM)
C     .. Local Scalars ..
      INTEGER         I, J, NUM
C     .. External Functions ..
      INTEGER         SIZES
      EXTERNAL        SIZES
C     .. Intrinsic Functions ..
      INTRINSIC       MAX, MIN
C     .. Executable Statements ..
C
C     Number of elements that needs to be copied
C
      NUM = SIZES(NDIM,DIMS)
C
C     Data channel that needs to be copied
C
      J = MIN(MAX(1,WHICH),NVARS)
C
C     Copy the elements into the output array
C
      DO 20 I = 1, NUM
         DATOUT(I) = DATAIN(J,I)
   20 CONTINUE
C
C     Number of data channels in the output lattice
C
      NVARS = 1
C
      RETURN
      END
C     Subroutine to determine the number of elements in a lattice
C
      INTEGER FUNCTION SIZES(NDIMS,DIMS)
C     .. Scalar Arguments ..
      INTEGER                NDIMS
C     .. Array Arguments ..
      INTEGER                DIMS(NDIMS)
C     .. Local Scalars ..
      INTEGER                I
C     .. Executable Statements ..
C
      SIZES = 1
      DO 20 I = 1, NDIMS
         SIZES = SIZES*DIMS(I)
   20 CONTINUE
C
      RETURN
      END
```

Since *cxLattice* is a C data structure, the Fortran code contains pointers to the *cxLattice* data type (see **Chapter 3, "Using the Lattice Data Type"**).)

1.3. Setting up your Build Environment

Before you start to build this module, you need to set up a test directory and copy the example code into it.

1. Make a subdirectory in your home directory as follows:

```
cd ~
mkdir testMod
```

2. Change to the new subdirectory and copy the example code from */usr/explorer/src/MWGcode/Tutorial/** into *testMod*:

```
cd testMod
cp /usr/explorer/src/MWGcode/Tutorial/C/channel.c . (C)
cp /usr/explorer/src/MWGcode/Tutorial/Fortran/channelselect.f . (Fortran)
```

You can now build your module using the Module Builder and run it in the Map Editor. Just follow the steps in the next sections.

1.3.1. C or Fortran?

The method for building modules is the same for both languages, although some of the information you enter varies according to whether the user function is written in C or Fortran. Because the similarities are so great, this tutorial presents the process for users of both languages together. Where the activities for the two language diverge, the differences are clearly marked and parallel instructions are given for each language.

1.3.2. Invoking the Module Builder

Make sure you are in the *testMod* directory, then start up the Module Builder:

```
mbuilder
```

The Module Builder main window appears (see **Figure 1–1** and **Figure 1–2**). Click in the Module Name slot to select it and delete the default, then type in the module name *ChannelSelector*. In the next slot, enter the name of the file that contains the user function:

C Users: *channel.c*

Fortran Users: *channelselect.f*

You can also specify include files, additional libraries or your own *Makefiles*, but none are required for building the *ChannelSelector* module. To learn more about these options, read through **"The Include Files"** and **"User Makefiles"** in Chapter 2.

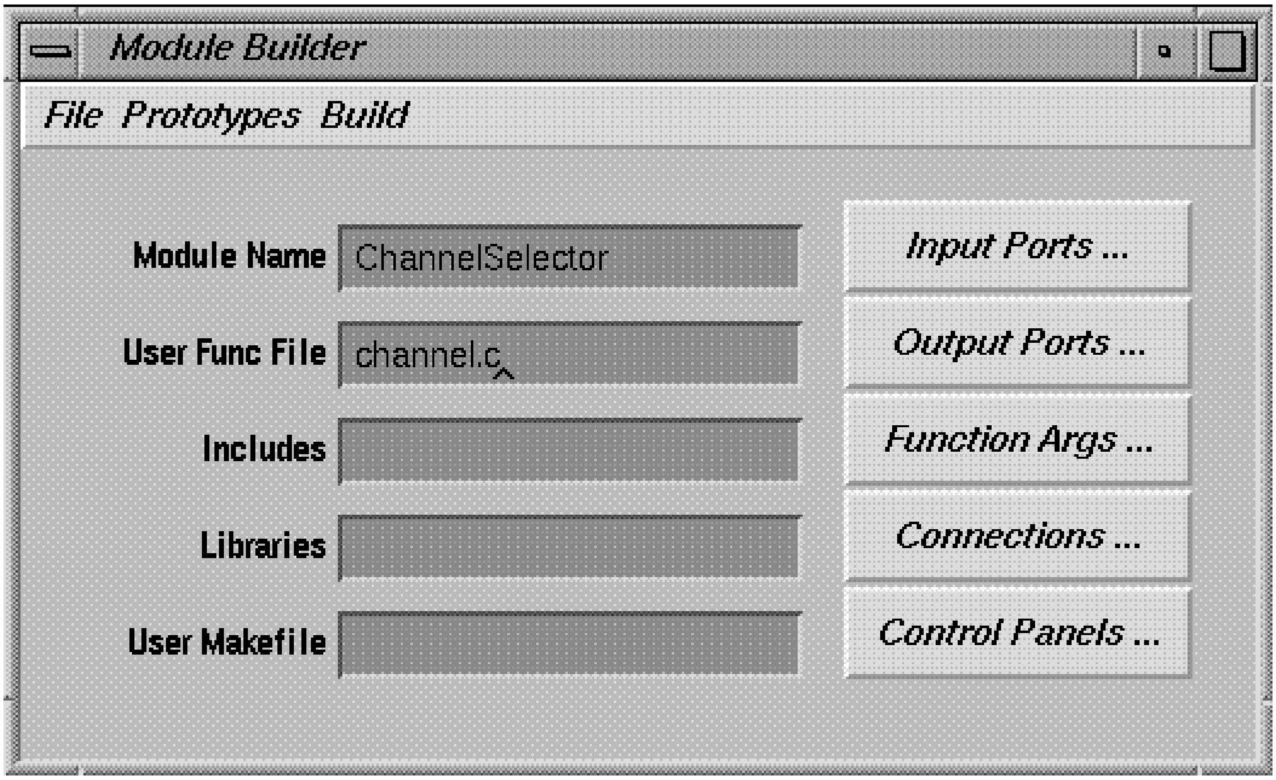


Figure 1–1 The Module Builder Window for C Users

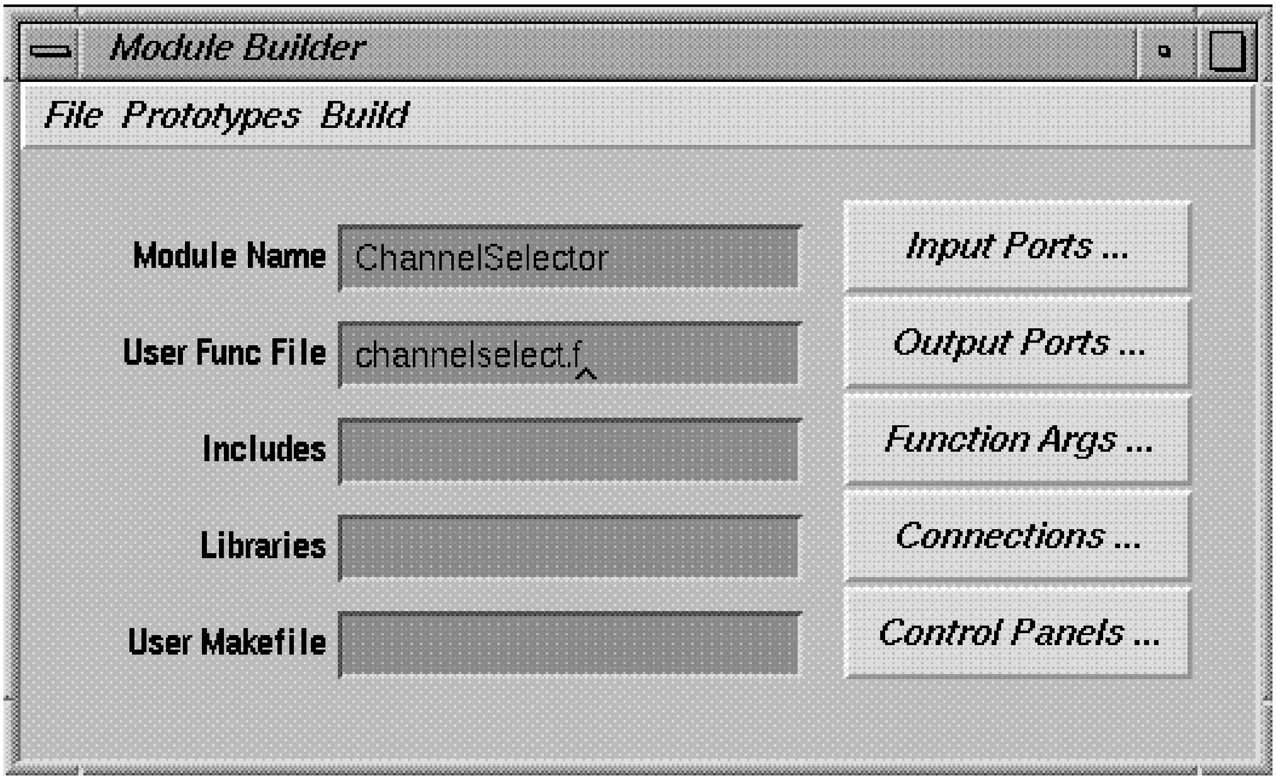


Figure 1–2 The Module Builder Window for Fortran Users

1.3.3. Saving the Module Resources

The Module Builder creates a number of files as you go through the construction process. The Module Builder automatically saves all the module characteristics, or resources, that you have just specified in the module resource (*.mres*) file before it builds the module. This is a binary file and cannot be accessed directly.

You can use "Save Resources" on the File menu to save your work at any time during the module construction phase. It is a good idea to save your work periodically if you expect to be interrupted or if you are building a very complex module.

1.3.4. Quitting the Module Builder

To quit the Module Builder at any time once it has been invoked, click on the *File* menu on the main window and select "Quit" (see **Figure 1–3**).

Figure 1–3 Module Builder File Menu

1.4. Defining the Internal Structure

The inputs, outputs, function arguments, and connections among different variables constitute the internal structure of the module, and you define this structure in the Module Builder windows. You can have only one window open at a time, other than the main window.

1.4.1. Creating an Input Port

Click on the first button, *Input Ports*, to bring up the Input Ports window (see **Figure 1–4**).

Note: The window is initially displayed with only one text slot. Click on the slot to highlight it and press **`Enter`** to produce a new text slot.

The module requires an input port that will accept data in the form of the lattice data type.

1. Click in the first text slot to highlight it, then type the name "Input" in the first text slot to create an input port.
2. Click on the button on the option menu in the middle column to choose the data type menu and select *cxLattice.*

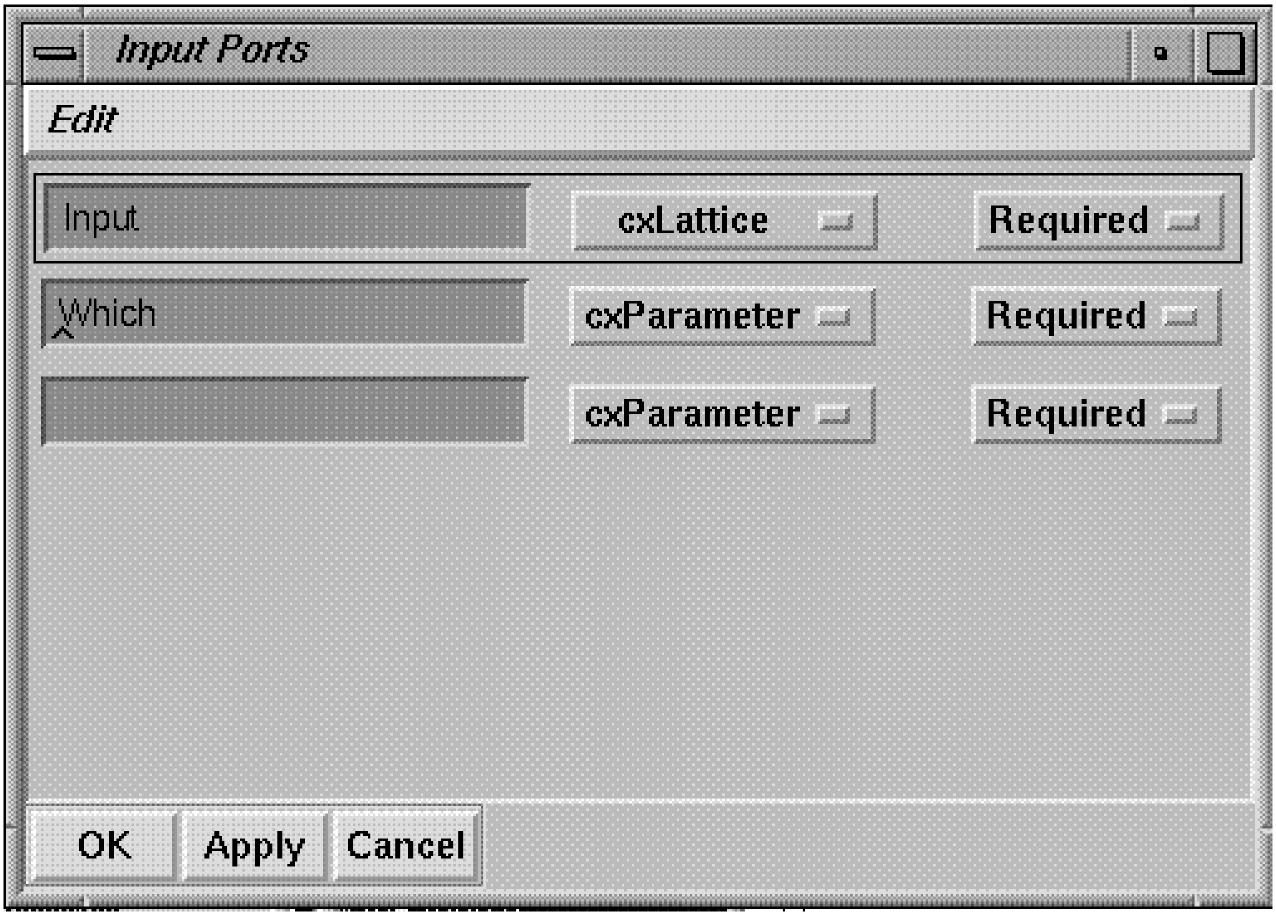


Figure 1–4 Input Ports Window

Defining the Data Type

When you select *cxLattice* from the data type option menu, the Lattice Constraints window appears (see **Figure 1–5**). To define the characteristics of the lattices the input port will accept:

1. Click off the yellow buttons to get the results shown in **Figure 1–5**.

   The input port called "Input" will now accept a lattice of any dimension and data type, containing any number of data variables and with any number of coordinates per node. Note that it is good practice to write modules with as few restrictions as possible. As the user functions have restrictions on the data types, the generality is in this case achieved through automatic type coercion by IRIS Explorer wrapper code. In general, you need not change existing code to widen its application range; IRIS Explorer can do this for you.

2. Data values are required, but coordinate values need not be given, so *Data Structure* is marked "Required" and *Coord Structure* is "Optional".

3. Click *OK* to save the setting and close the window.

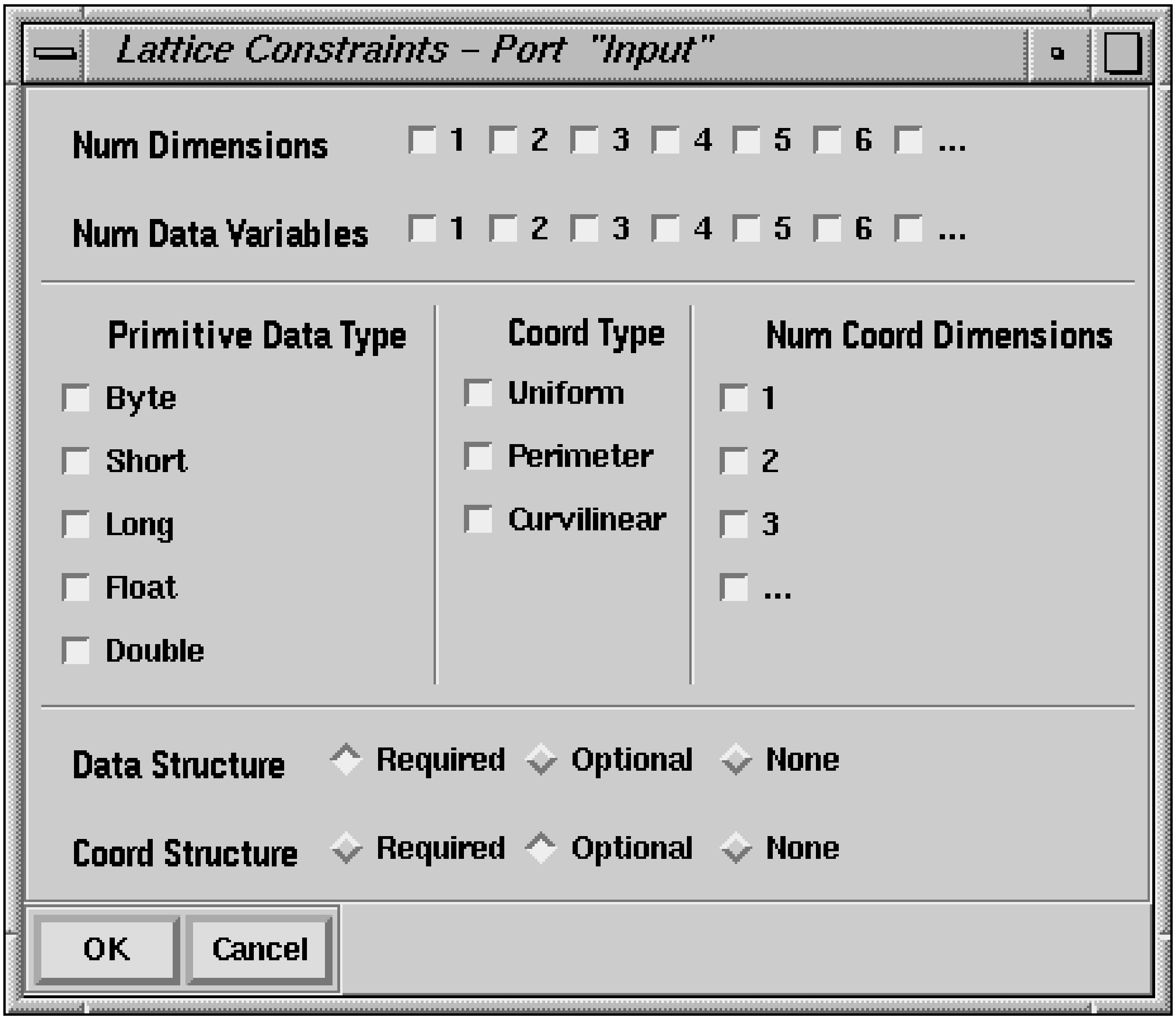


Figure 1–5 The Lattice Constraints Window

Defining the Parameters

The next step is to define a parameter for the input port in the Input Ports window (see **Figure 1–4**).

1. Click the second slot in the Input Ports window and enter the parameter name "Which".
2. Select *cxParameter* from the data type option menu.
3. Select *Required* from the option buttons in the third column for both ports.
4. Click *OK* to save the setting and close the window.

The module must receive data on both its input ports before it will fire, so each port is made a "Required" port.

1.4.2. Creating an Output Port

Click the *Output Ports* button to bring up the Output Ports window (see **Figure 1–6**). The module needs an output port so that it can pass the data from the selected channel downstream to the next module.

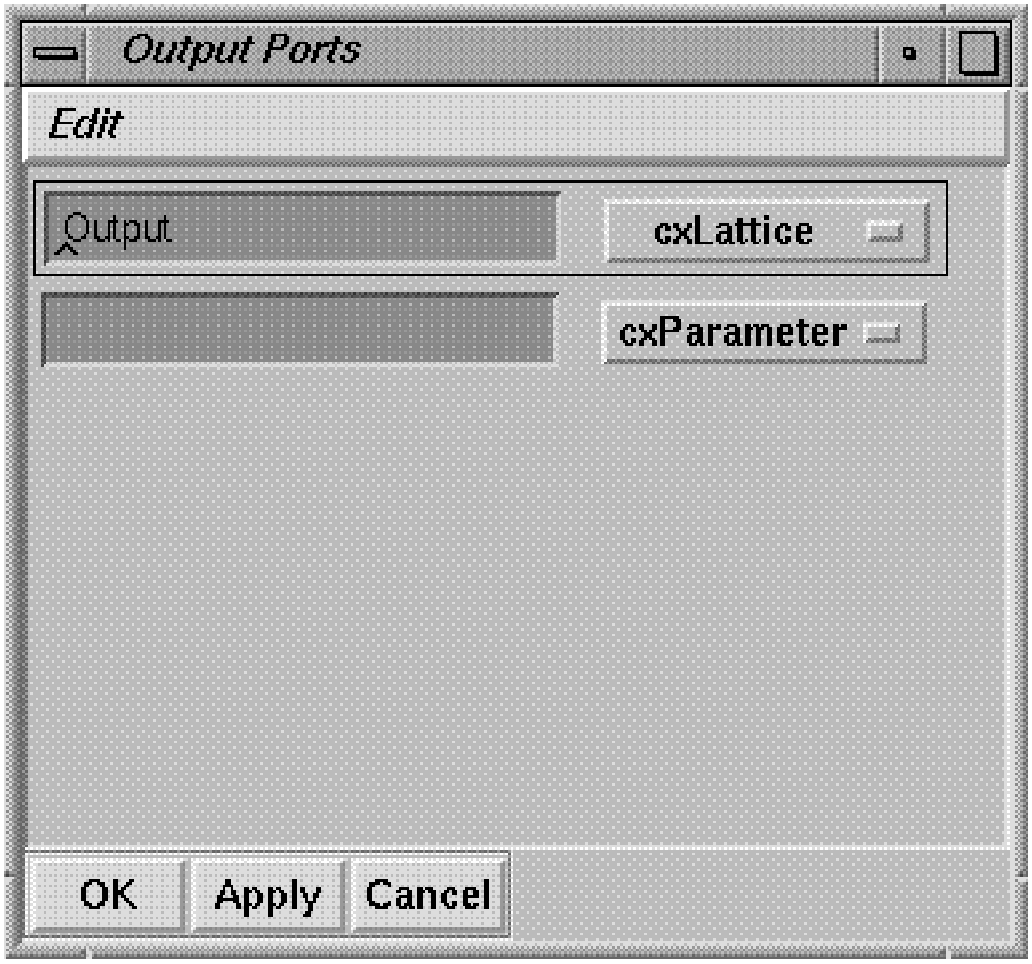


Figure 1–6 The Output Ports Window

To create the output port:

1. Enter the name of the output port, "Output", in the text slot.
2. Select *cxLattice* from the data type option menu.

   When the Lattice Constraints window appears (see **Figure 1–7**), set the constraints. All the constraints, except the *Num Data Variables*, are the same as those for the input lattice, however, the *Num Data Variables* is set to 1 in the output lattice because only one data channel is being output.

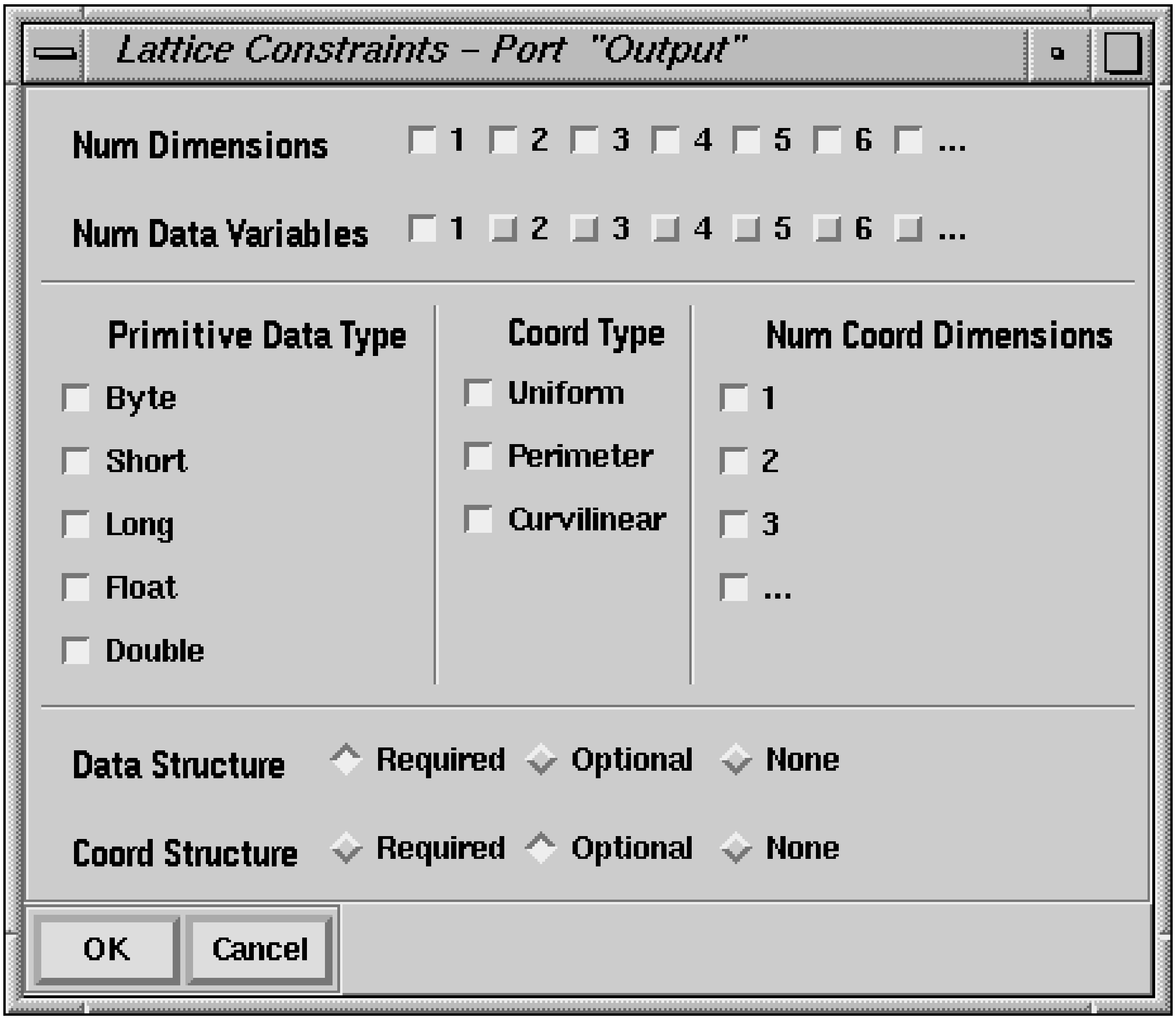


Figure 1–7 Lattice Constraints for the Output Port

3. Click *OK* to save the constraints setting and close the window.
4. Click *OK* to save the output port setting and close the Output Ports window.

For more information about input and output ports, read **"Creating Ports"** in Chapter 2.

1.4.3. Defining the Function Arguments

The Function Arguments window defines each function argument in the user function. Each function argument must be connected to an input or output item. You list the function arguments in their calling sequence, as shown in **Figure 1–8** and **Figure 1–9**.

To define the function arguments:

1. Click the *Function Args* button to display the Function Arguments window. When the window first appears, all of its fields are blank.
2. Enter the name of the function, *chan*, in the *Func Name* slot. This is the name of the function contained in the *channel.c* or *channelselect.f* file.
3. Set the language option.

   If you are using the C function, select "C" from the language option menu.

   If you are using the Fortran subroutine, select "Fortran" from the language option menu.
4. Enter the first function argument for the user function in the text slot. Press `Enter` after typing each function argument name to generate a new text slot.

   The function arguments for the C function are shown in **Table 1–1** and **Figure 1–8**.

   The function arguments for the Fortran subroutine are shown in **Table 1–2** and **Figure 1–9**.
5. Set the corresponding values for the argument type and reference by clicking each option menu button in turn. The C values differ from the Fortran values, as you will see from the function argument tables.
6. When you have entered all of the arguments, click *Apply* to have them accepted but keep the window open.
7. At the bottom of the window, you will see the function name displayed with all its arguments listed. Click *OK* to close the window and return to the Module Builder main window.

For more information about the data types and reference mechanisms, read **"Defining Function Arguments"** in Chapter 2.

These are the function arguments for the C function.

Argument Name	Type	References
nDim	int	Scalar
dims	int	Array
nDataVar	int	&Scalar
which	int	Scalar
dataIn	float	Array
dataOut	float	Array

Table 1–1 Function Arguments for the C Function

These are the function arguments for the Fortran subroutine.

Argument Name	Type	References
nDim	integer	Scalar
dims	integer	Array
nDataVar	integer	Scalar
which	integer	Scalar
dataIn	real	Array
dataOut	real	Array

Table 1–2 Function Arguments for the Fortran Subroutine

Figure 1–8 shows the argument names, data types, and references you enter if you are building your module around the C user function.

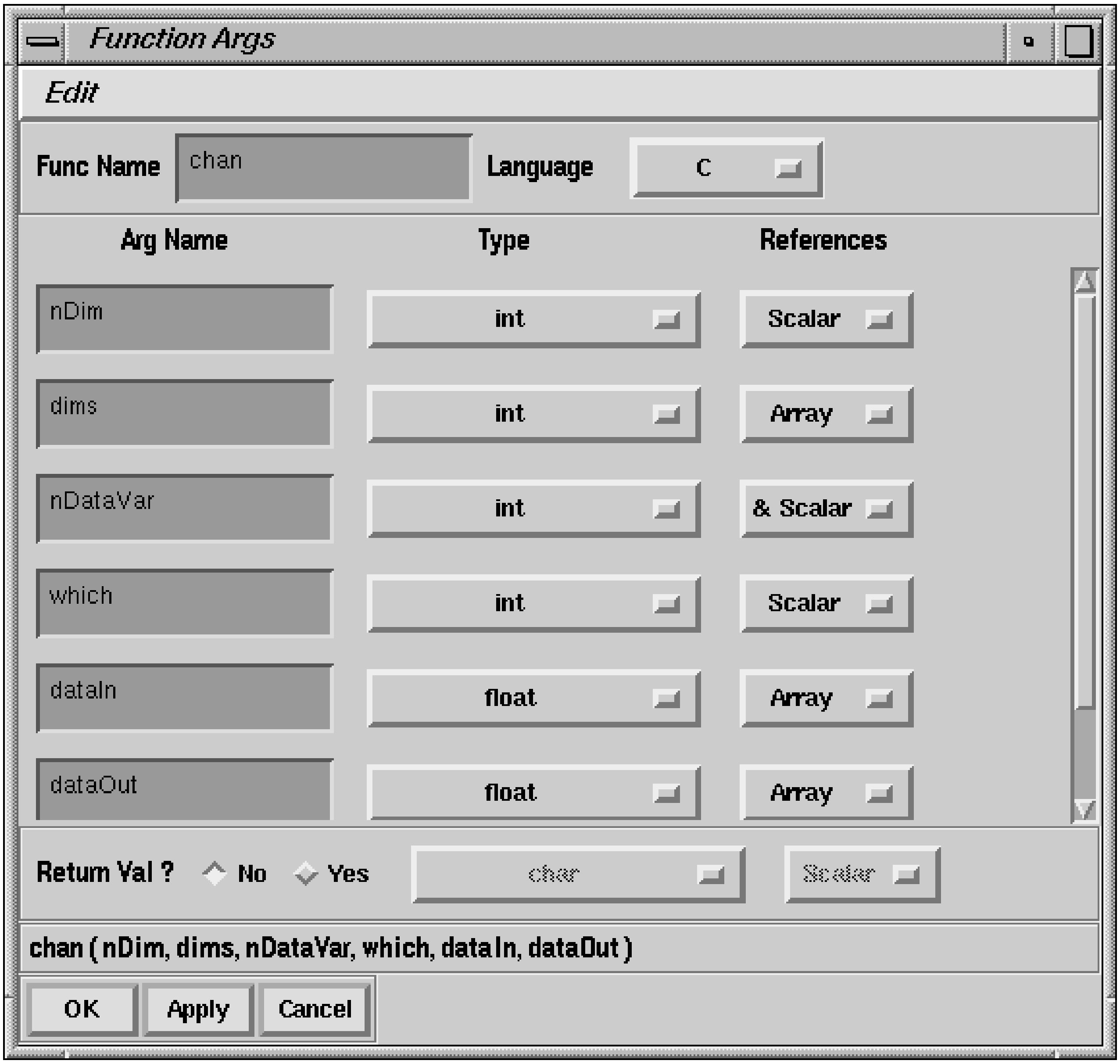


Figure 1–8 The Function Args Window for C Users

Figure 1–9 shows the argument names, data types, and references you enter if you are building your module around the Fortran user function.

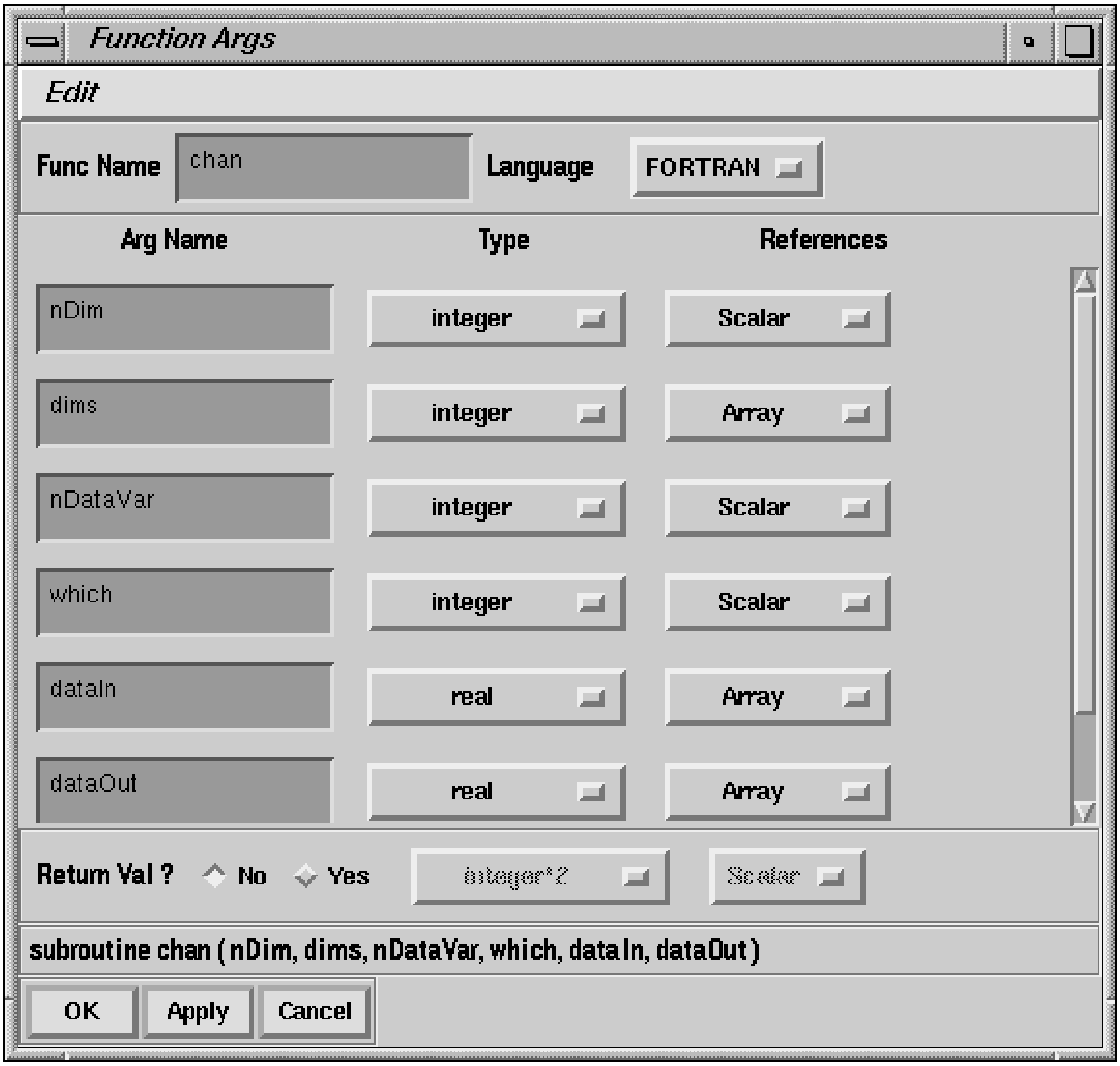


Figure 1–9 The Function Args Window for Fortran Users

1.4.4. Connecting Ports and Function Arguments

Each port must be connected with one or more function arguments, so that the incoming data can be routed to the correct function argument and the outgoing data can be directed to the correct output port.

The connections are identical for C and Fortran user functions.

1. Click on the *Connections* button to invoke the Connections window (see **Figure 1–10**). When it is first displayed, no connecting wires exist. These appear as you wire ports and function arguments together.
2. To connect an input port to a function argument or output port, click with the right mouse button on the input port name bar. A popup menu appears, listing the data structures associated with the port.

For example, the input port "Input" accepts lattice data, and its popup menu (see **Figure 1−11**) lists the lattice substructures. They contain data arriving on the module's input port from upstream. The connections you make from the data structures on the input ports to the function arguments determine which data are recognized and processed, and which data are ignored.

3. Select the structure member you want, for example, Dimensions Array. The function arguments and ports that can accept a connection are highlighted in green (output ports are darker green).

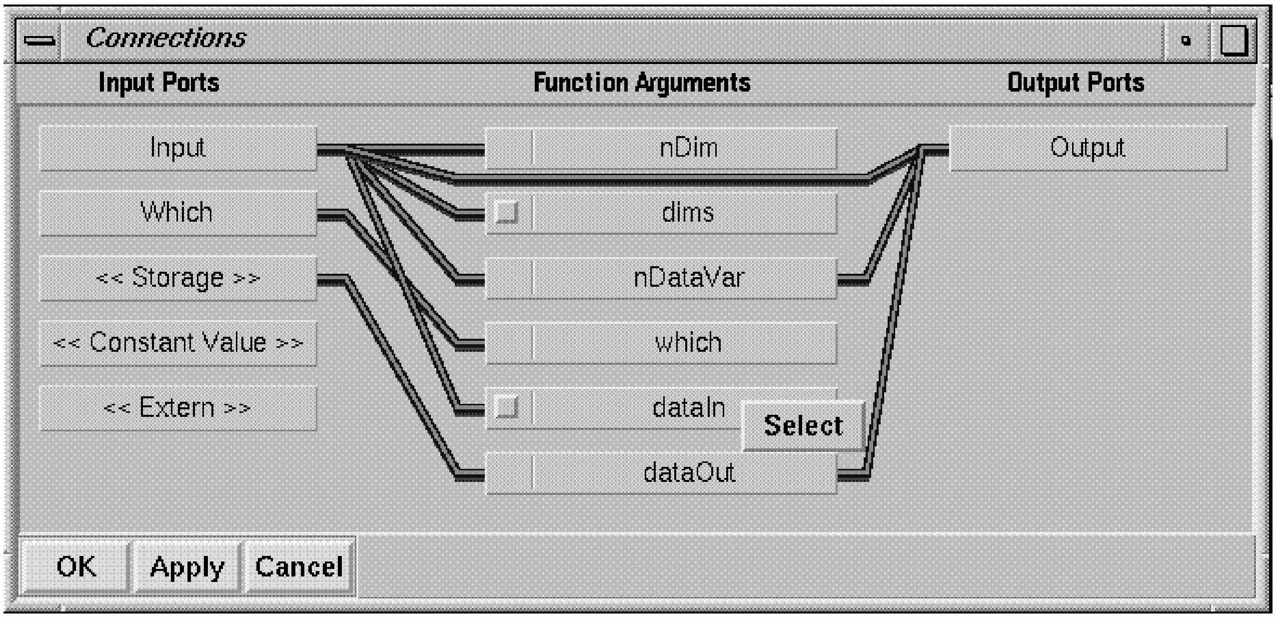


Figure 1−10 The Connections Window

4. Then click on a function argument, for example, *dims*, with the right mouse button. A small *Select* button appears (see **Figure 1−10**).

   When you click *Select*, a blue wire links the input port with the function argument.

5. Follow the same procedure to link input ports or function arguments to the output port. The output port menu items are printed in blue if they require a connection for the integrity of the data structure (see **"Connecting Arguments to Ports"**, in Chapter 2).

To break a connection, simply click on both ends of the connection, as in the Map Editor.

The popup menu in **Figure 1−11** shows one connection from the input port (top−level Lattice Structure) to the output port.

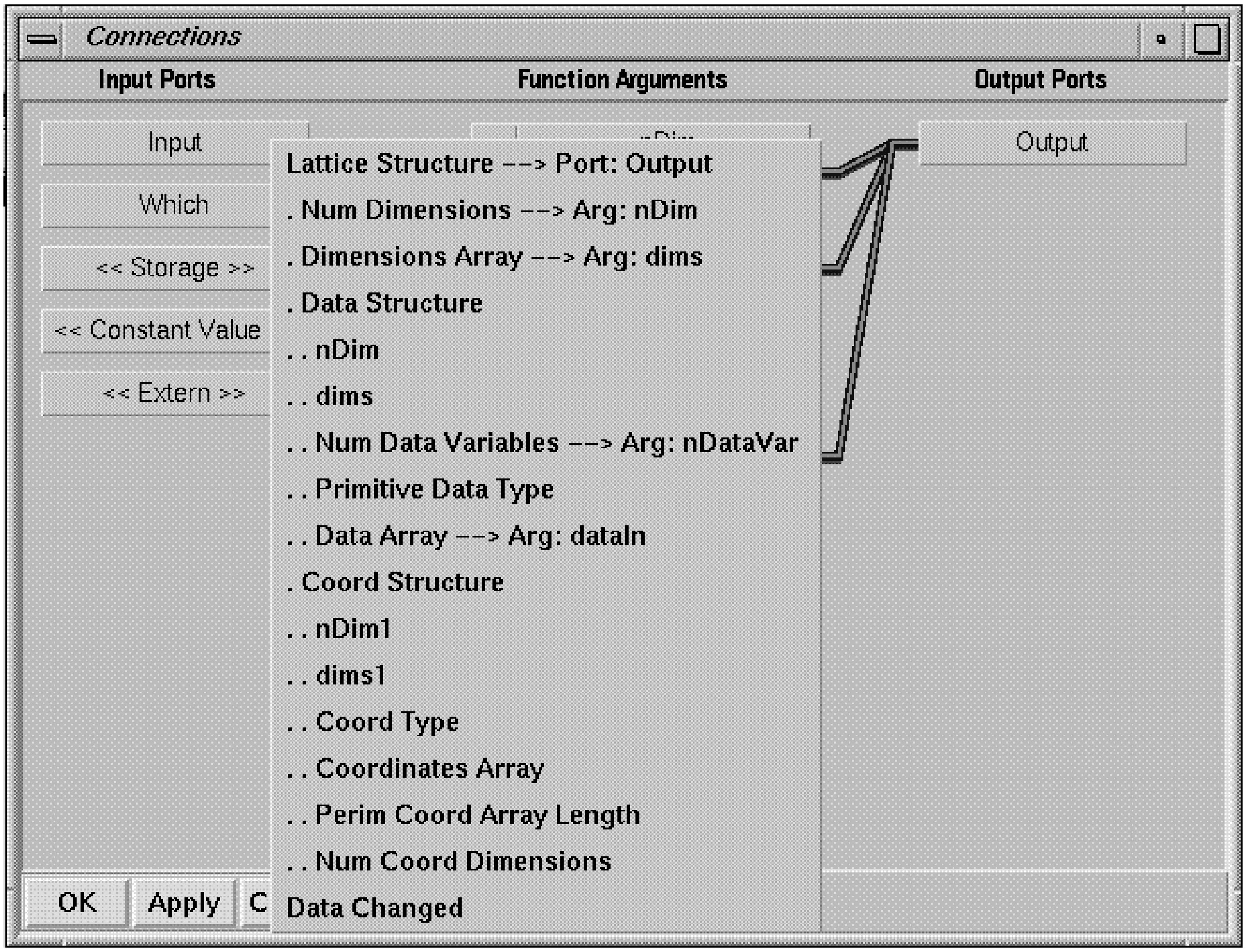


Figure 1–11 Input Port Popup Menu

It also shows four connections from lattice components, such as the number of dimensions (Num Dimensions), to function arguments. The Output port has three connections passing into it, although the port menu is not shown. The details of the data type structures are explained in Chapters 3 through 7.

Making the Right Connections

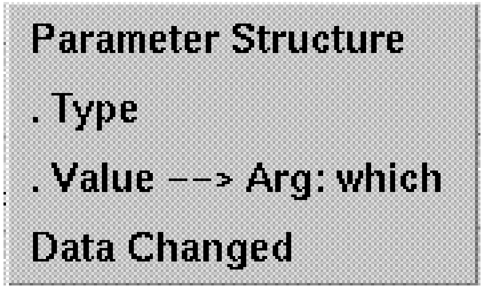
The correct connections are critical to the proper operation of the module. In this example, the values affected by the function arguments are passed separately to the lattice structure on the output port. The values for all members of the lattice data type not explicitly linked to function arguments are passed directly from the input port to the output port as default values.

For more about connecting ports and function arguments, read **"Connecting Arguments to Ports"** in Chapter 2.

To build the *ChannelSelector* module correctly, you must make these connections. They are the same for the C and the Fortran modules.

Input	*LatticeStructure* to port *Output: Lattice Structure*
	Num Dimensions to argument *nDim*
	Dimensions Array to argument *dims* (switch off *Copy* by pressing the yellow button)
	Num Data Variables to argument *nDataVar*
	Data Array to argument*dataIn*

Which *Value* to argument *which* (see the **option menu**)



<<Storage>> "Select" to argument *dataOut,* then set the storage value (see **Figure 1−12** or **Figure 1−13**). Note that storage for arrays must be allocated in this way for Fortran programs, as in general there is no Fortran equivalent of the C malloc function. C programmers might prefer to allocate storage for arrays in the usual way with malloc inside their programs, rather than using the IRIS Explorer <<Storage>> connection.

nDataVar "Select" to port *Output*: *Num Data Variables*

dataOut "Select" to port *Output*: *Data Array*

Using the Pseudoports

<<Storage>>, *<<Constant Value*>> and <<*Extern*>> are "pseudo−ports". When you connect one to a function argument, you use the second item on the **function argument popup menu**) to define the value for the pseudo−port. In this example, you need to define the storage value. The value is different for C and Fortran modules.

To set the storage size for *dataOut*, click the menu item "Set storage size" and enter the text shown in **Figure 1−12** and **Figure 1−13** for the C and Fortran modules respectively. There is a default setting which you should delete first.

This text is expected by the code that the Module Builder generates when the module is built. In this example, a routine is called to compute the number of floating (C) or real (Fortran) values that must be allocated for the *dataOut* function argument.

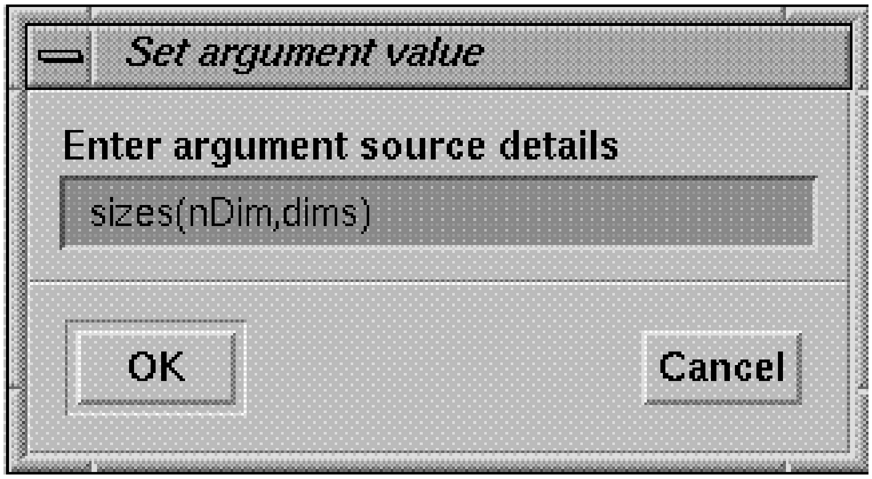


Figure 1–12 Setting Storage Size for C Users

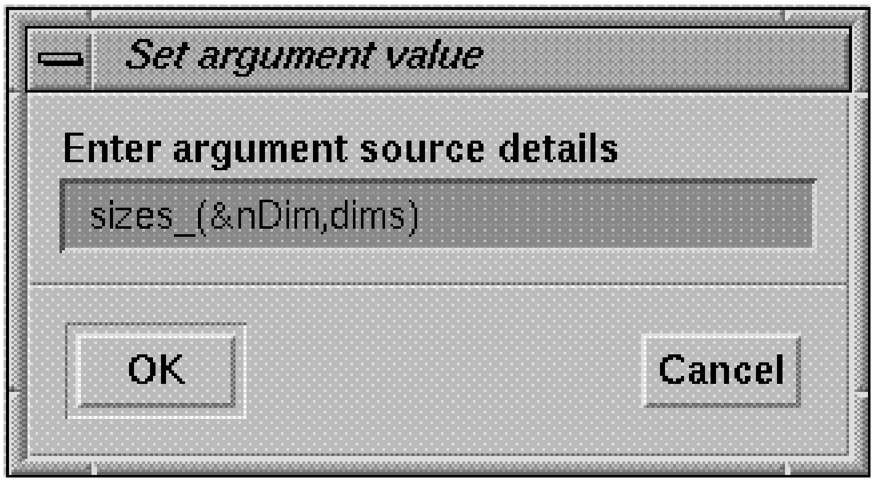


Figure 1–13 Setting Storage Size for Fortran Users

You can enter comments after the values if you wish, to remind you and explain to others why you chose this value.

When you have filled in all the popup windows and made all the connections you need, click *OK* to save the configuration and close the window.

For more information about the use of pseudo ports, read **"Creating Ports"**, in Chapter 2.

1.5. Defining the User Interface

The module has an interface that allows it to be controlled by the user. This interface is the module control panel. See Chapter 4, "Editing Control Panels and Functions" in the *IRIS Explorer User's Guide* for more details on control panel design and widget use.

The module control panel holds widgets, such as dials and sliders, that let you control the values of input port parameters interactively.

Click the *Control Panels* button to invoke the Control Panel Editor. The Control Panel Editor and a prototype control panel appear (see **Figure 1–14**).

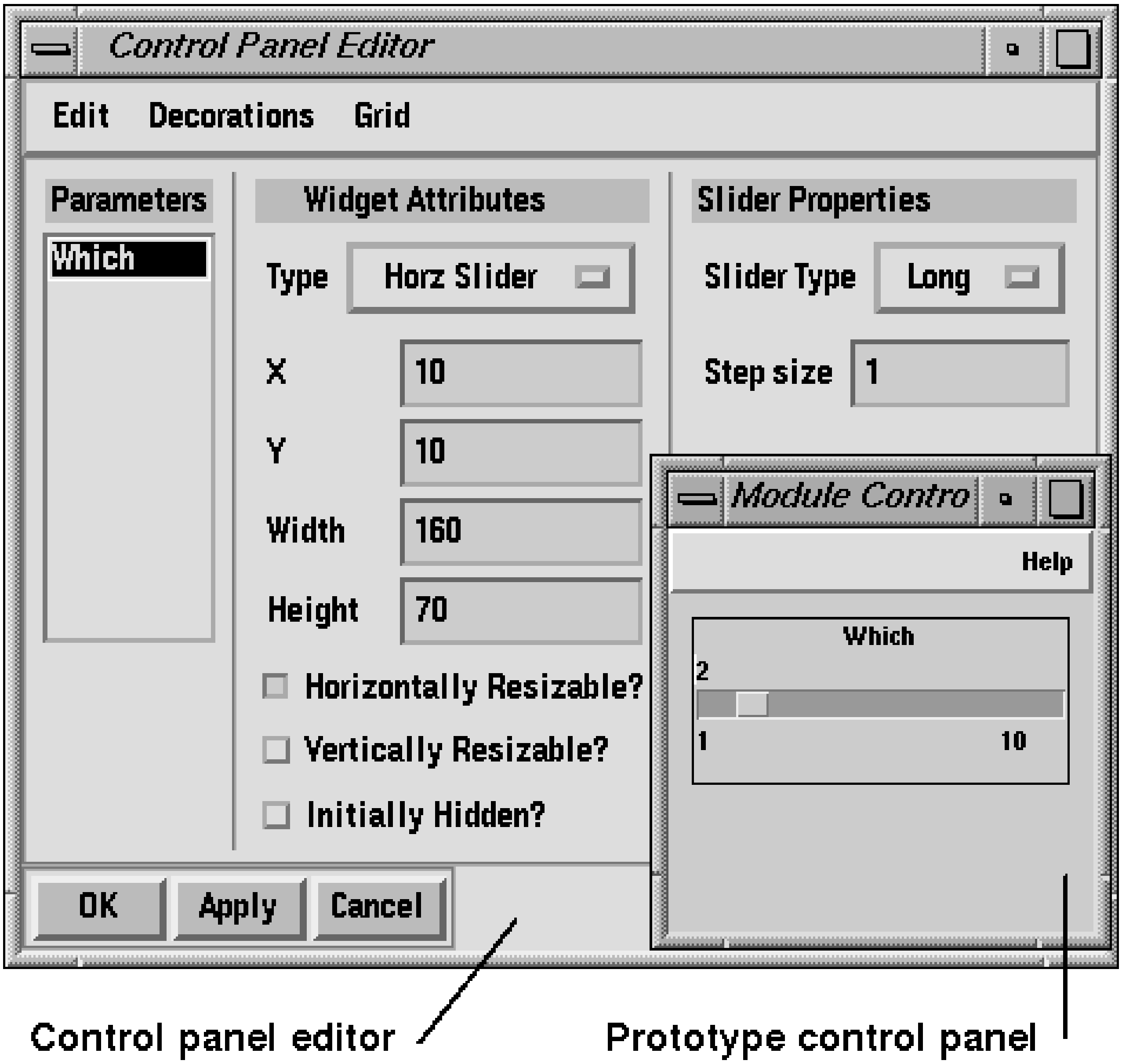


Figure 1–14 Control Panel Editor and Control Panel

All ports of type *cxParameter* appear in the "Parameters" list in the Control Panel Editor. In this example, "Which" is a parameter port (see **Figure 1–14**). It is characterized in the control panel as a slider by default because its values are floating or real.

To change the widget for "Which" to another kind, click the left mouse button on it to select it, then click the Type option menu under Widget Attributes. You can select any widget type that is not grayed out.

To change the widget's position in the control panel, click it to highlight it, then hold down the left mouse button and drag it into a new position.

To change the widget's size, use the type–in slots to enter a new set of dimensions, or drag on any of the eight black handles around the widget.

To set the lower limit for the slider, double–click on the left–hand text slot beneath the slider and type **`1`**, then hit **`Enter`**. The module input port can accept a lattice with any number of data variables, and so the upper limit for the slider will be unknown in general, but you can still set a sensible value for this number. So, type **`5`** in the right–hand slot and hit **`Enter`** (see **Figure 1–14**). Finally, you can enter a current value for the parameter in the text field above the slider.

1.5.1. Creating a Menu Item

You can create your own menu bar options for the module control panel. For this module, you will create a File menu with a "Reset" option.

1. Select "Menu Bar" from the Edit menu on the Control Panel Editor. The Menu Bar Editor appears (see **Figure 1–15**).
2. Type **`File`** in the first slot to name the menu.

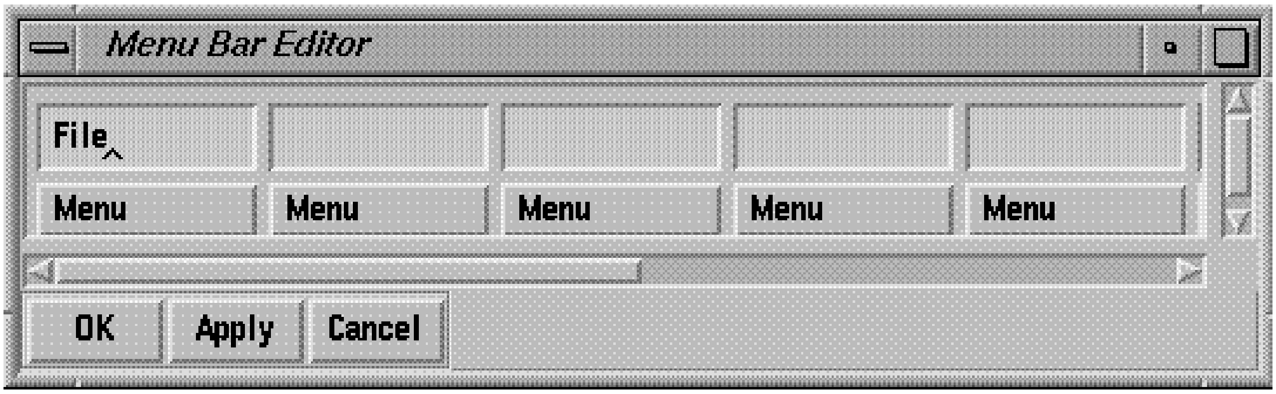


Figure 1–15 Menu Bar Editor

3. Click the *Menu* button underneath the slot. The Menu Editor appears (see **Figure 1–16**).
4. Type `Reset` in the first menu item slot to name the action.
5. Click the option menu button next to the first slot and select the action "Set param" from the option menu.

   A new option menu button appears on the right, together with a text slot.
6. Click the option menu button and select the parameter which is to be set (in this case, there is only one parameter, "Which").
7. Type `1` in the text slot to specify the parameter value.
8. Click *OK* to apply the selection and close the window.

   You can close each open window individually, or you can select *OK* on the Control Panel Editor, which closes all the subsidiary windows and applies the changes you have made.

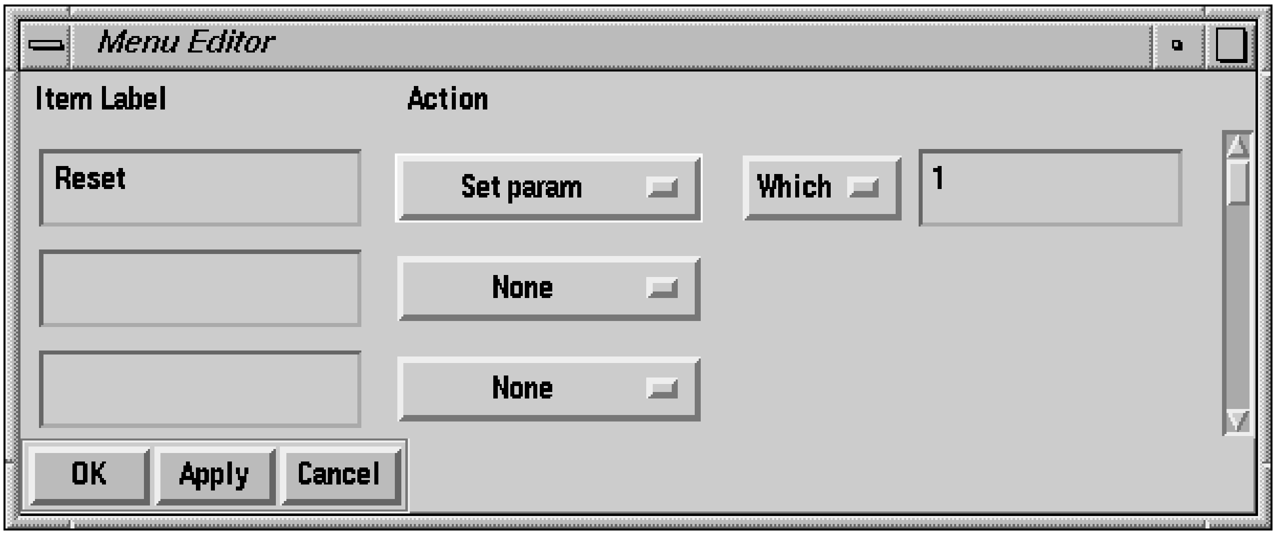


Figure 1–16 Menu Editor

1.5.2. Testing the Menu Bar

You can click on the *Control Panels* button to bring up the prototype control panel again. You will see that it now has a File menu on its menu bar. Click File to see the "Reset" option.

This completes the construction of the module. The next stage is to build the module and install it in IRIS Explorer so that you can use it in a map.

1.6. Building the Module

You have a blueprint for a module that must now be turned into an executable program. The code is linked and compiled during the build process.

1.6.1. Selecting Build Options

The Module Builder Build menu provides lets you set defaults and build the module. To check the build options for the Channel module, select "Options" from the Build menu. The Build Options window appears (see **Figure 1–17**).

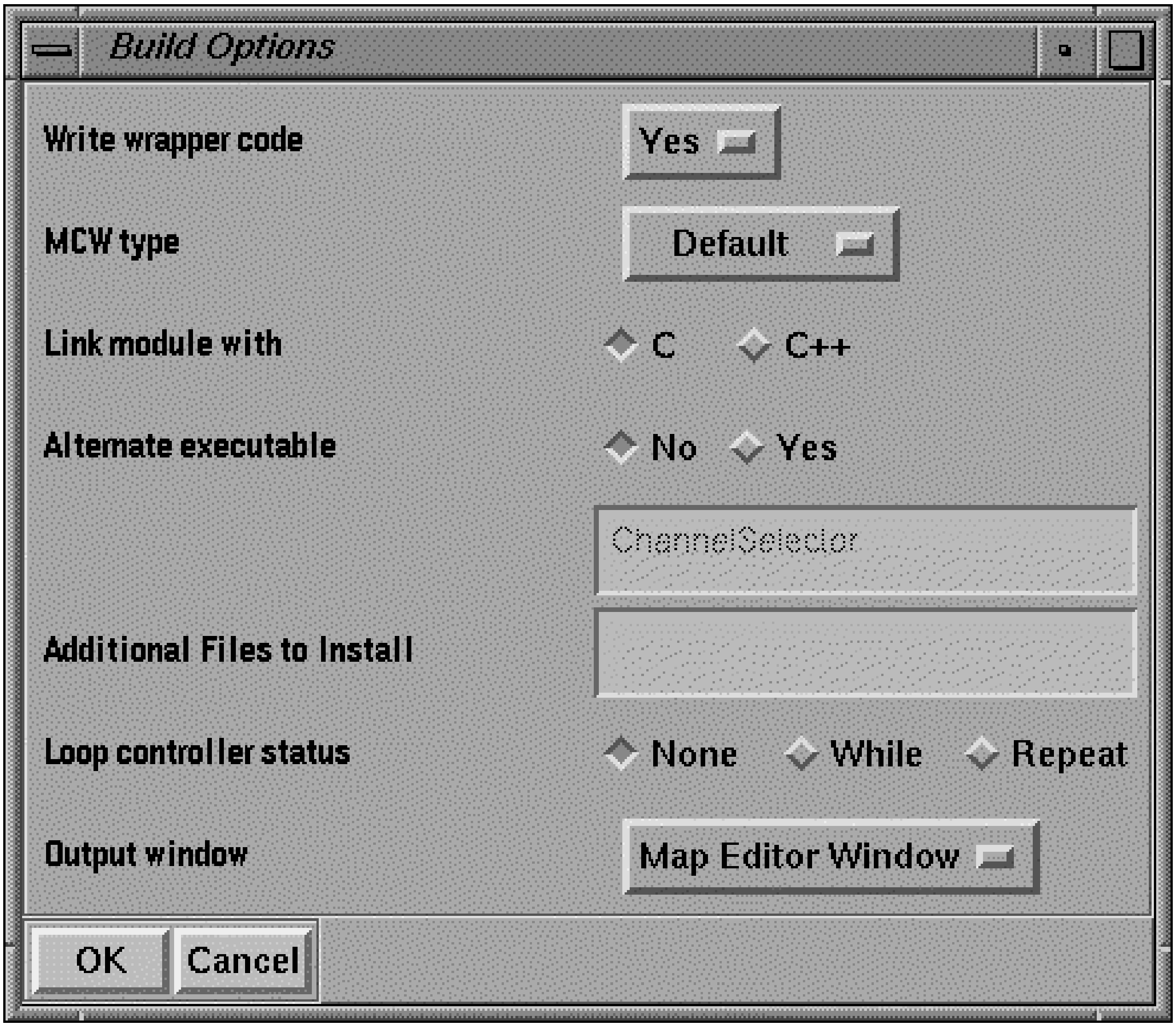


Figure 1–17 Build Options Menu

You want to accept all the current defaults. You want the Module Builder to:

- write wrapper code, called the Module Data Wrapper, which interfaces between the user function and the rest of the module
- create a default Module Control Wrapper (MCW) for managing other module functions
- link the module to C because the user function is not written in C++
- use the stipulated executable program file
- install no related files when it installs the module
- use the Map Editor as the text output window for the module

Click *OK* to accept the default options and close the window.

For more information about build options, read **"Using the Build Menu"**, in Chapter 2.

1.6.2. Running the Build Command

The Build command is on the Build menu (see **Figure 1–18**).

1. Click on the Build menu and select "Build". This command builds the module in the current directory, which is *~/testMod* in this case.

   If you receive this message:

   ```
   compile, link or install MAY have failed,
   ```

   check to see that you are running the Module Builder from your current directory.

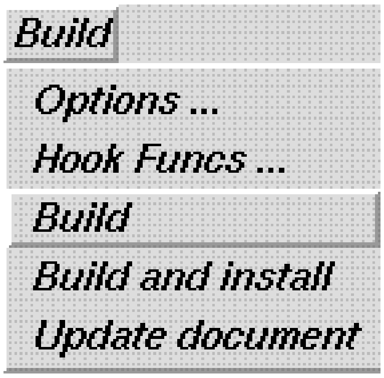


Figure 1–18 Build Menu Options

2. Select "Quit" from the File menu to exit the Module Builder.

You can now run the Map Editor by typing **`explorer`** at the shell prompt, and launch the *ChannelSelector* module from the Module Librarian.

To see how it works, connect these modules: *ReadImg* to *ChannelSelector* to *DisplayImg.* Read in the image file */usr/explorer/data/image/flowers2.rgb* and select each of the color channels, 1 to 3, in turn. You may also reset the channel to 1 by clicking on the "File" option on the menu bar and selecting "Reset".

Chapter 2

Understanding the Module Builder

This chapter explains in detail how to use the Module Builder to construct a module. It describes how you:

- create a user function for the module
- define include files and libraries for the module
- create your own *Makefiles*
- define data input and output ports
- specify the function arguments and link them to the ports
- configure the build environment
- use the build options to build and install the completed module

It also includes some examples of user functions for simple modules.

Advanced topics are marked with ✰. As an ordinary user, you can usually skip these passages on a first reading.

2.1. Creating a User Function

The user function or subroutine is a program that details what you want the module to do. You can write it yourself or use a program that already exists. The user function file contains the algorithm that actually performs the data transformation. It also contains any API subroutines you are using in the module.

You can write the program in C++, C, or Fortran 77; however, all user functions must export a C−style user interface. Writers of C++ user functions must qualify their routines accordingly. For example, you can qualify a routine called `funcName()` as

```
extern "C" funcName()
```

Note: In the IRIS Explorer environment, all files containing C++ code have the suffix *.C*.

You can write the user function at any time, but you cannot build the module without it. You can set up other parts of the module, such as the input and output ports and connections, save the module resources, and then pull up the half−finished module and build it later, when the user function is complete. You can also generate a prototype and build the module later.

2.1.1. Using the IRIS Explorer API

The IRIS Explorer application programming interface (API) is a set of functions in C and corresponding subroutines in Fortran that give the module writer greater control over module behavior. For example, instead of the Module Builder generating code to connect the user function to the ports, the writer can use the API to do so. The writer can also build more complex modules by using the API to make specific calls to other parts of IRIS Explorer. The API is described fully in the *IRIS Explorer Reference Pages*.

Note: If your user function is written in C++, use the C API. Since C++ is a superset of C, the C API can be called directly by C++.

✰ Geometry Modules in Fortran

If you try to build a geometry module in Fortran, your only access to the Geometry library is through the IRIS Explorer API subroutines. If you want to use the Open Inventor geometry library directly you must build geometry modules in C or C++, you will also need a Developer’s Licence for Open Inventor on your platform.

☆ **The Scripting API**

If you are writing a module, you can use *cxScriptCommand* to issue scripting commands from within the module. When you call *cxScriptCommand*, a message transmission and several context switches result, so performance is improved if you can bundle expressions in a single string.

For more information on syntax of *cxScriptCommand* and the Skm language, refer to Appendix B in the *IRIS Explorer User's Guide.*

2.2. Running the Module Builder

You use the Module Builder to create the module resources file and the control panel for a module and to build and install the module so that it can be launched in the Map Editor.

In general, you should run the Module Builder from the directory in which you want the completed module to reside; otherwise, the module may not build. See **"Building and Installing Modules"** for more information.

2.2.1. Opening Modules

At the shell prompt (%), type:

```
mbuilder
```

The Module Builder main window appears on the screen. If you open the Module Builder without a filename, the main window comes up with `default_module_name` in the name slot. In this window, you set the variables for the module resources file, a binary file called *<modulename>.mres*, which contains all the information required to build the module.

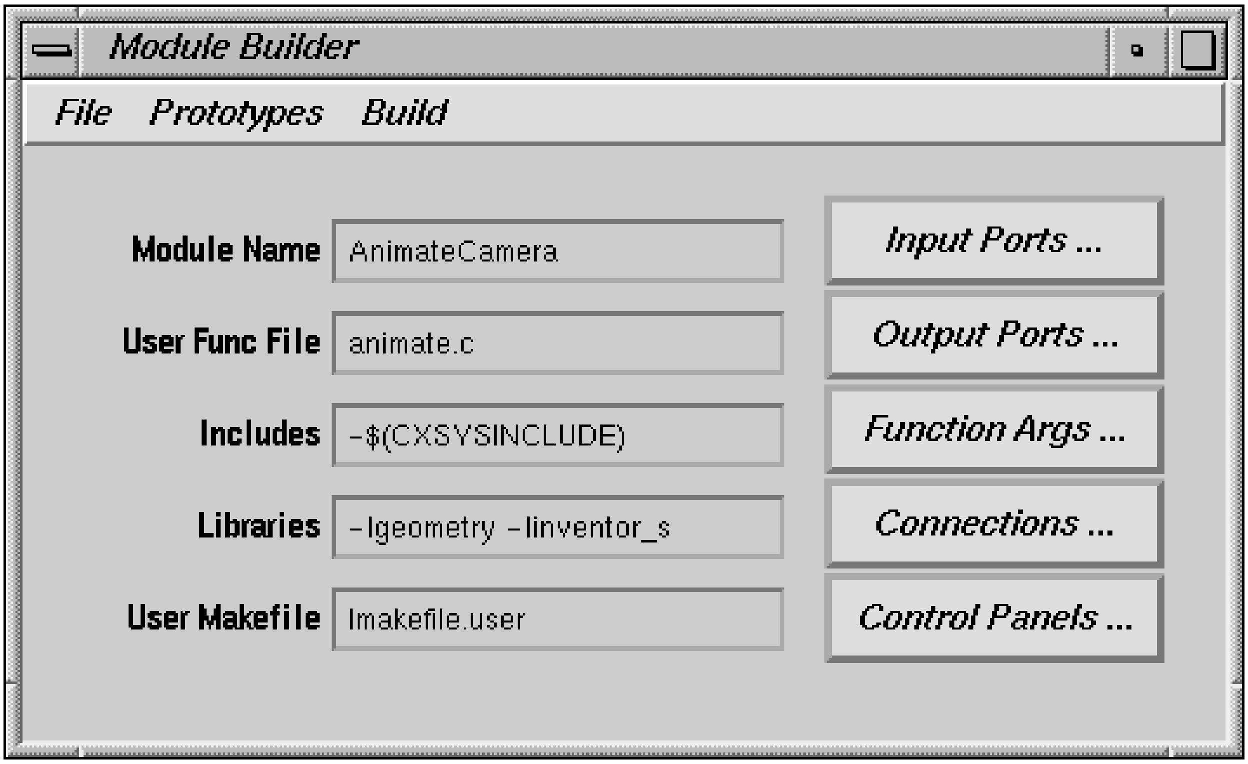


Figure 2–1 Module Builder Main Window

The *.mres* file template is created as you apply the information from each of the Module Builder windows (see **"Moving Around the Windows"**) and is saved when you first select *Save Resources* from the File menu.

To open the Module Builder and a module at the same time, change to the directory in which the module resides. For example, to look at the structure of an existing IRIS Explorer module, such as *GenLat*, type:

```
cd /usr/explorer/modules
mbuilder GenLat
```

The Module Builder opens with the information pertinent to the module displayed in its windows.

2.2.2. Moving Around the Windows

Five windows can be opened from the main Module Builder window, but only one besides the main window can be active at a time. If you try to open more than one, a dialogue box appears and tells you which window is already open. The active window may be buried under other windows or it may be iconified. To re–display the open window, click on the corresponding button in the main Module Builder window.

To select a text type–in slot in any of the Module Builder windows, click on the slot. The active slot is outlined in black and has a cursor in it. You can type into the active slot.

When you have entered information in a window, you can use one of the three buttons at the bottom of the window:

Cancel	Cancels all the information before it can be saved
Apply	Saves the information and leaves the window open
OK	Saves the information and closes the window

Changes are not incorporated into the module template until you click on *Apply* or *OK*.

To save the template as a *modulename.mres* file, you must use *Save Resources* on the File menu. You should do this often, to avoid losing your work.

2.3. The Main Window

Filling out the Module Builder main window is equivalent to setting the variables required by the language compiler and link editor.

2.3.1. The Module Name

The *<modulename>* can be:

- the name of an existing module that you want to examine. For example, the command

  ```
  mbuilder LatToGeom
  ```

  opens the Module Builder with *LatToGeom*. As you move through the Module Builder windows, you have access to the module's internal structure, called the module resources.
- the name for a new module that does not yet exist, but that you are about to create. The Module Builder main window appears with the new name in the Module Name slot, but all other windows are empty.

If you want to look at a different module resource file or to create another new module, use the File menu in the Module Builder main window. To select another *modulename.mres* file, click *Open* and select the file from the file browser that appears. To create a new module, select *New*.

Naming Modules

Enter the name of a module in the Module Name slot (see **Figure 2–1**). To enter or change the name of a module in the slot, click the text slot to highlight it and type a new name.

Each module name must begin with an alphanumeric character. Since they are UNIX files, you should follow UNIX file−naming conventions.

Modifying Existing Modules

If you are modifying an existing module to create a new one, you should save the modified resource file with a new module name and leave the original *modulename.mres* files intact.

Note: Because you cannot change the source code of an existing IRIS Explorer module, the modifications you can make to these modules are very slight; however, you can modify your own modules endlessly, if you wish.

Saving Modules

By default, IRIS Explorer saves newly created modules in your working directory. It saves the module in the form of a *modulename.mres* file when you select:

- *Save Resources* from the File menu (saves the contents of the windows as module resources), or
- *Build* or *Build & Install* from the Build menu (saves the resources, compiles and links the source code)

2.3.2. The User Function File

Enter the name of the file or files that contain your computational, or user, function in the User Func File text slot. For example, the *Gradient3D* module has one user function file, called *Gradient3D.f.*

☆ Your module may use an alternate executable, that is, a user function connected with another module (see **"Using the Build Menu"**). If so, don't enter the name of the user function in the User Func File slot; it appears on the Build Options menu.

Using Several Function Files

If the user function is large and complex, you can break it down into smaller sub−functions, each of which is in its own file. In the User Func File slot, list two or more files with a space between each name.

You can mix languages (C++, C, and Fortran), source files (*filename.C*, *filename.c*, or *filename.f*), and object files (*filename.o*) in this slot. For example, *Render*, which is a very complex module, uses *render.C*, *event.C*, *init.C*, *SoSceneViewer.C*, *SvManipList.C*, *cxGeoObject.C*, and several other files.

Using Makefiles

If you use library (*.a*) files in the User Func File slot, you need to supply your own *Makefile* in the User Makefile slot (see **"User Makefiles"**). If you use source or object files, the Module Builder uses standard *make* rules to determine file dependencies and to build the necessary object files, and you need not supply a Makefile unless you have a specific reason for doing so.

The *Makefile* generated by the Module Builder has default rules for generating an object file from C, C++, and Fortran source files.

☆ Writing Fortran User Functions

In IRIS Explorer, **.inc* files have no *cpp* directives in them at all. To use `#include` syntax, name your source file **.fi*, and the IRIS Explorer *Makefiles* will filter it for you. If your Fortran compiler naturally supports *cpp*, you can still use **.f.* To use `include` statement syntax, name your source file **.f.*

Use of certain data types requires that you explicitly include certain files. If you use *cxLattice*, you may need to include *Typedefs.inc*. If you use *cxPyramid*, you must include *cxLattice.inc*. If you use *DataTypes.inc*, however, it includes *Typedefs.inc* explicitly.

2.3.3. The Include Files

The Module Builder automatically passes the "include" directories such as */usr/explorer/include* and */usr/include* to the compiler when compiling the module's source code; however, you may want to include directories or files from other places for a particular module.

Use the Includes slot to do this.

Enter the name of each included file as you would list it in a compile command. Use the form **`-I<directory>`**; for example, **`-I../myInclude`**. Thus, to build the *AnimateCamera* module required the following include command:

```
-I$(CXSYSINCLUDE)
```

which appears in the Module Builder **Includes slot** for the module; here, **`CXSYSINCLUDE`** is an environment variable, set when the module was built. This contains the name of a directory where the required include files for the module are to be found.

If you include a system–defined variable by itself, it takes the form *${VARIABLE}*; for example, *${GEOMETRYINCS}.*

☆ You can also use the **`-D`** option for variables; for example, the *Render* module uses **`-DEXPLORER`**. For more information on these options, refer to the UNIX *cc(1)* command.

2.3.4. Associated Libraries

Some modules, such as those handling graphics, geometry, or mathematical procedures, and those written in Fortran, must be linked to related libraries. IRIS Explorer automatically adds certain default libraries into the module*Imakefile* for Fortran and geometry modules if you have:

- selected Fortran as the language for the user function (see **"Defining Function Arguments"**). The system–defined variable is *${FORTRANLIBS}.*

  Note: Even if you have a C user function, IRIS Explorer gives you *$(FORTRANLIBS)* if you have a *<filename>.f* or *<filename>.fi* file in your User Func File list.

- specified the *cxGeometry* data type on an input or output port. For example, *Render* has a Geometry input port. The system–defined variable is *${GEOMETRYLIBS}*.

The system variables *${FORTRANLIBS}* and *${GEOMETRYLIBS}* include any default libraries that are machine–dependent.

You may wish to:
- add libraries in addition to the default libraries
- use the libraries defined by *${FORTRANLIBS}* and *${GEOMETRYLIBS}* in a module that does not automatically include them
- add your own libraries to a module

Use the Libraries slot to do this.

Enter the name of each library as you would list it in a link command. Use the form **`-l<library>`** for individual libraries. For example, to build a module that uses the Graphics Library, enter:

```
-lgl_s
```

to associate the Graphics Library with the module. For more information on the **`-l`** and **`-L`** options, refer to the UNIX *ld(1)* command.

☆ If you are creating a module with a user–defined data type (UDT) on an input or output port, IRIS Explorer places any associated libraries in the module's *Imakefile* in the form **`-l<UDTname>`**; however, if you use the UDT inside a module that has no UDT port, to create a modified lattice, for example, you must specify the associated library in the Libraries slot. User–defined data types are described in **Chapter 8, "Creating User–defined Data Types"**.

2.3.5. User Makefiles

In general, the Module Builder uses standard *make* rules to determine the relationships among individual module files and to determine which files to recompile when a change is made to some aspect of the module.

✰ However, you can create and use your own *Makefile* or *Imakefile* if:

- you have listed object files in the User Func File slot
- you want a specific file dependence in operation. For example, *AnimateCamera* uses its own *Imakefile.user* file (see **Figure 2–1**).
- you have changed the value of an environmental variable. For more details on this topic, see **"Creating a Makefile"**.

Give your *Makefiles* names such as *MyMakefile* or *Makefile.user.* The names *Makefile* and *Imakefile* are reserved for use by the system and you should avoid them.

Note: The *Imakefile* extends the functionality of *make* by providing "include" facilities. It deals with X Window™ features.

For details on creating your own *Makefiles*, refer to **Appendix A, "Using Makefiles"**.

2.4. Creating Ports

The module ports allow you to get data into and out of the module. They constitute the interface between the module itself and other modules, the *external* interface. You can define up to 40 input or output ports for a module, each of which must have a unique name. If the Module builder detects a duplicate name, it renames the second port and issues a message.

Note: IRIS Explorer automatically adds new syncronisation ports called "Fire" and "Firing Done" to all modules when they are placed in the Map Editor (see Section 2.6.2, "Using the Synchronization Ports" in the *IRIS Explorer User's Guide*). In addition, if the module is a loop controller, it also adds a "Loop Ended" synchronisation port. To prevent confusion, you should therefore avoid using these names for the ports which you define when you build the module.

2.4.1. Defining Input Ports

Figure 2–2 shows the Input Ports window for *VectorGen.* This module has two ports, "Input" and "Scale Factor". When you define an input port, you give the Module Builder the information that it needs to allow legal connections between this port and output ports on other modules.

Input Port Characteristics

Three fields need to be defined: the port name, its data type, and its status.

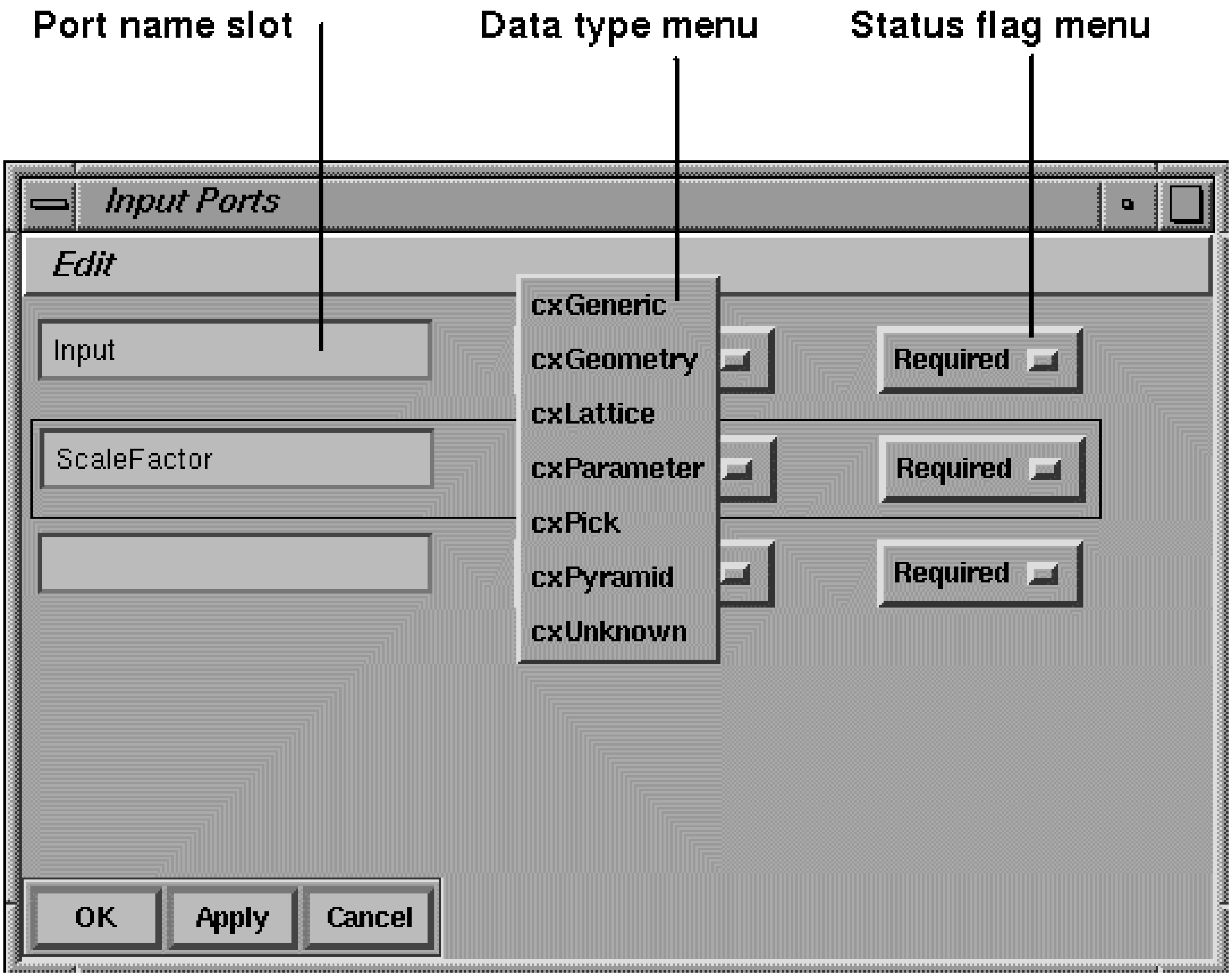


Figure 2–2 Input Ports Window

The *port name* goes in the text type–in slot. You can give a port any name you wish, as long as it is unique within the module. Descriptive names, such as "Color Input", make the module easier to use than, say, "In1". The name appears on the Input Ports menu of the module control panel in the Map Editor.

The *data type* on the port provides crucial information about data structures. It denotes the form in which incoming data must be cast for the port to accept it. The center column of option menus lists the available data types, of which you select one.

For more details on IRIS Explorer data types, see **Chapter 3, "Using the Lattice Data Type"** through **Chapter 8, "Creating User–defined Data Types"**.

IRIS Explorer has seven built–in data types. These are listed on the *data type* pop–up menu shown in **Figure 2–2**. In addition, you can create your own data types (see **Chapter 8, "Creating User–defined Data Types"**) and if any of these have been added, they will also appear on the *data type* pop–up menu for both the Input and Output Ports. For example, the pop–up menu for the Output Ports window in **Figure 2–6** shows an additional user–defined data type called *tstLattice*.

Users can connect only ports with the same data types, except for *cxGeneric*

Footnote: *cxGeneric* is a special case, a data type which can be wired to any other data type. It is described in Chapter 7.

. For example, if "Input" has the *cxPyramid* data type on its port, it can receive only a connection from a pyramid output port on another module. IRIS Explorer data types are all pointers to C structures, so they can be linked to Fortran user subroutines because Fortran can accept variables passed as pointers.

If you select the:

- *cxLattice* data type for a port, the Lattice Constraints window appears (see theLattice Constraints window, **Figure 2−3**).
- *cxPyramid* data type for a port, the Pyramid Constraints window appears (see Pyramid Constraints window, **Figure 2−5**).

The Map Editor uses the constraint options for type checking when modules are connected to one another in a map. If the incoming connection passes a lattice that is incompatible with the options defined on the receiving port, a warning message appears and the connection is not made.

The *status flag* menu lets you specify whether the port is optional or required. If an input port is:

Required	It must be wired up and have data on the port before the module will fire. This implies that it must also have a connection from an upstream module. (This is the default.)
Optional	It need not be wired up for the module to fire. However, once it *is* wired up, it behaves like a required port and it must have data on the port before the module will fire.

☆ Firing Sequences

The actual firing sequence of a module is controlled by the firing algorithm. For more detailed information on the passage of data into and out of ports, refer to **Appendix B, "The Firing Algorithm"**.

2.4.2. Setting Lattice Constraints

The *cxLattice* data type is flexible enough to accommodate a variety of lattice types. You can more narrowly define the kind of data that a single lattice port can accept by setting options in the Lattice Constraints window (see**Figure 2−3**).

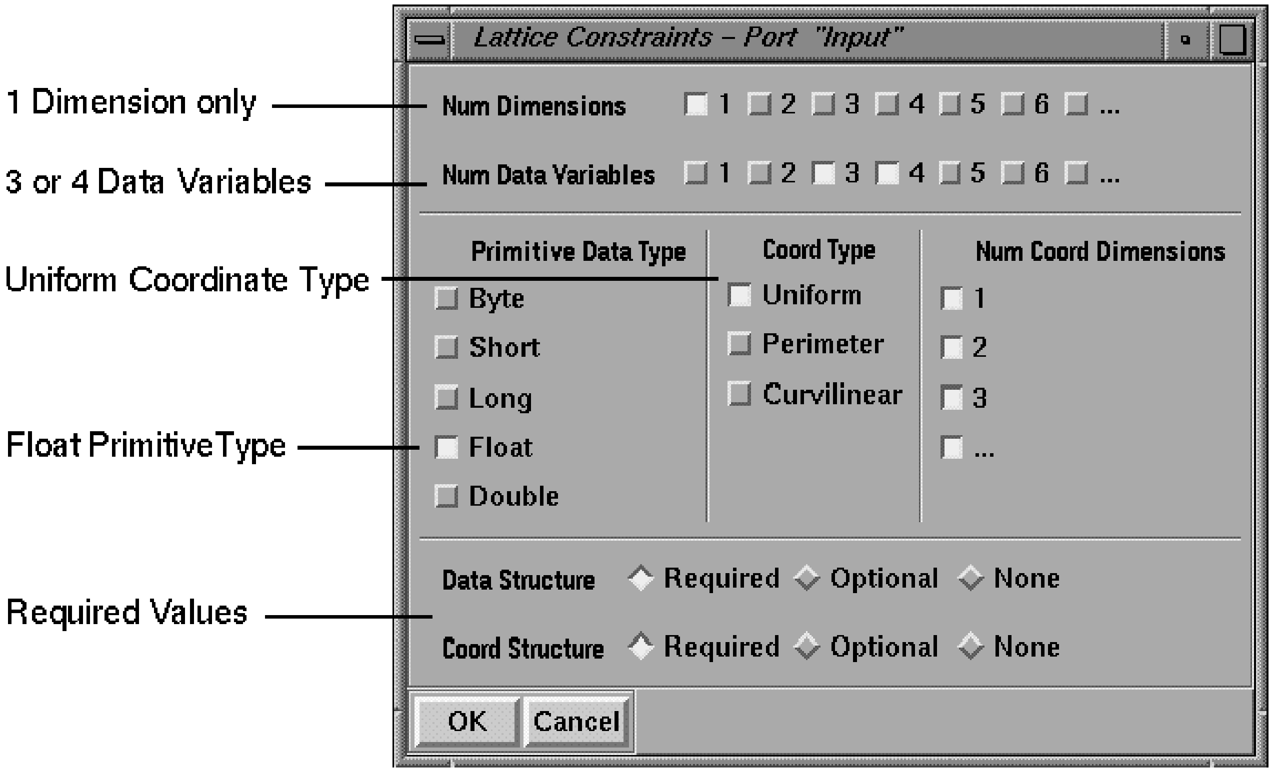


Figure 2–3 Lattice Constraints Window

You can set any restrictions that make sense for your lattice port. This applies to both input and output ports which accept or produce lattices.

For example, **Figure 2–3** shows the lattice definition for a port called "Input", which is expecting a colormap. The colormap lattice is a 1–D lattice with three (RGB) or four (RGBA) floating point data variables per node. It has uniform coordinates and it must contain both data and coordinate values.

Table 2–1 shows how these specification are applied in the Lattice Constraints window.

Colormap Lattice Specs	Button Positions
A 1–D lattice	Button #1 in the *Num Dimensions* field is ON
Three or four data variables	Buttons #3 and #4 in the *Num Data Variables* field are ON
Float data values	The *Float* button in the *Primitive Data Type* list is ON
Uniform coordinates	The *Uniform* button in the*Coord Type* list is ON
Curvilinear coordinate variables per node	Buttons 1 through 3 in the *Num Coord Dimensions* list are ON
Required values	The *Required* button is ON for both the *Data Structure* and *Coord Structure* fields

Table 2–1 Colormap Lattice Constraints

Defining Lattice Constraint Fields

Each field shown in the Lattice Constraints window defines a part of the lattice data type structure. The limits you set in this window should be consistent with the way the lattice is defined in the user function.

For more details on the lattice data type, see **Chapter 3, "Using the Lattice Data Type".**

Click the radio buttons to activate the options. The light−colored, concave form indicates that the option is allowed.

Num Dimensions

Your port can accept lattices ranging from 1D upward. You can restrict it to accept just one set of dimensions, such as 3D only, or you can let it accept, for example, both 2−D and 3−D lattices, depending on how specific or how complex the user function is.

Num Data Variables

You can specify as many data variables as the user function can handle for example, temperature, pressure, and altitude.

Primitive Data Type

You can specify any or all of the primitive data types. You can allow the port to accept lattice data of all primitive types, even though the user function accepts only one type of data such as bytes, for example.

In such cases, the Module Builder performs type coercion on the incoming data to convert it to the primitive type that the user function argument requires (see **"Automatic Type Coercion of Arrays"**).

Note: The lattice primitive data types are C data types. If your user function is written in Fortran, select the C equivalents of the Fortran data types the user function expects (see **Table 3−1** in Chapter 3).

Coord Type

You can specify any or all of the coordinate types. They are described in detail in **Chapter 3, "Using the Lattice Data Type"**.

Num Coord Dimensions

If the module accepts or produces a curvilinear lattice, you must specify whether the lattice can have one, two, or three coordinates per node of the lattice.

Data Structure and *Coord Structure*

These options let you refine the port status, which is determined in the Input or Output Ports window (Required or Optional). You can stipulate whether the lattice must have both data and coordinate values (the default) or whether one or the other may be optional. For example, a module such as *Interpolate* is concerned only with coordinate values, not with data, so you can set the *Data Structure* on the output port lattice to Optional.

✰ You can use the *Data Optional/Coords Optional* setting to allow the module to accept a lattice that has coordinate values but no data values. This lets you describe the geometry of a shape (such as a sphere or a pyramid mesh) by using coordinate values only. You can then feed in a lattice containing data values only that show, for example, the variation in stress levels over the pyramid surface.

✰ Automatic Type Coercion of Arrays

If you are working with lattice data and coordinate arrays, you can use the automatic type coercion facility. You can write a module's user function to operate on a single data type, such as a float, but allow the module to accept any primitive type as input on a specific port. The Module Builder then generates code to coerce data on that input port from its native primitive type to the primitive type required by the function argument to which the port is connected.

The Module Builder applies its coercion rules automatically during port and argument specification, based on the selections made in the Lattice Constraints window and the function arguments.

At run−time, the data is converted into the desired type, copied into memory, then passed to the computational function. The coercion proceeds as a series of individual coercions. **Figure 2−4** shows the traversal path and the coercion pairs.

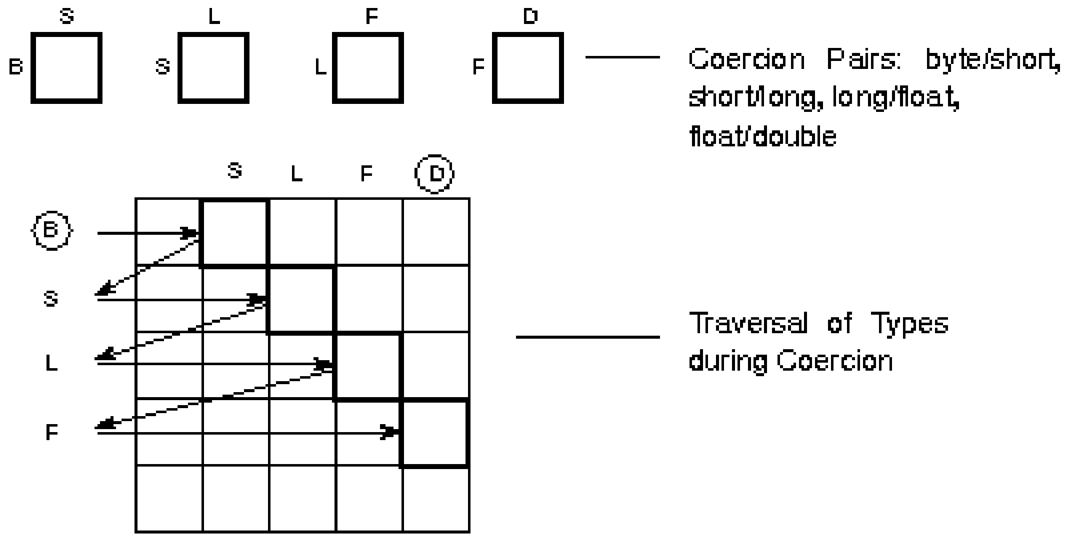


Figure 2–4 Sequence of Type Coercions

The C language process is:

D → F → LI → SI → B, or `double` to `float` to `long int` to `short int` to `byte`.

The Fortran language process is:

double precision → real → integer → integer*2 → character*1.

Out–of–range data is clipped to the acceptable range. Such coercions as

B → SI → LI → F → D cause no loss of precision.

The following paragraphs describe the coercion process:

Double to Float

Although converting from double to single precision involves a significant loss of precision, it is used when single and double versions of internal module code are not both available.

Float to Long Int

Float to Long Int coercion clips to the range of signed long integers, using the integer representation of the host computer and rounded to the nearest representable integer.

Long Int to Short Int

The short int type stores integers between 0 and 65535. The translation maps [0,65535] to [0,65535] and clips integers outside the range to 0 or 65535.

Short Int to Byte

The byte type stores integers from 0 to 255. Bit data and encodings of other data can also use this type, but in type coercion, a byte is used in integer form. The translation from integer to byte maps [0,255] to [0,255] and clips integers outside the range to 0 or 255.

Note: This feature makes module writing and integration much easier for the module writer, but it also requires time and a great deal of memory on the part of IRIS Explorer. It may be more efficient to write a module so that it accepts the correct data type without any coercion. This way, you will avoid the considerable "memory bloat" that is the cost of coercion.

Returning to the Port Window

Click *OK* to have your constraints accepted and to close the window. You are returned to the Input Ports window. Click *OK* to accept the port specifications and to return to the Module Builder main window.

2.4.3. Setting Pyramid Constraints

You can use the *cxPyramid* data type in a variety of ways. Some pyramids are built with every layer specified, and others are constructed in compressed form, with reference made to a standard structure in a pyramid dictionary for the compressed layers. The Pyramid Constraints window (see **Figure 2–5**) lets you specify whether the port can accept compressed or uncompressed pyramids and also establishes the level at which compression can take place.

For details on pyramid compression, see **Chapter 4, "Using the Pyramid Data Type"**.

Defining Pyramid Constraint Fields

Click on the radio buttons to make your selections.

Num Layers — You can specify a pyramid with any number of layers. For example, if you select 2 and 3, the port will accept 2–D and 3–D pyramids only.

Base Lattice — The base lattice is the lattice at the base of the pyramid. You can stipulate that the base lattice be required, optional, or absent.

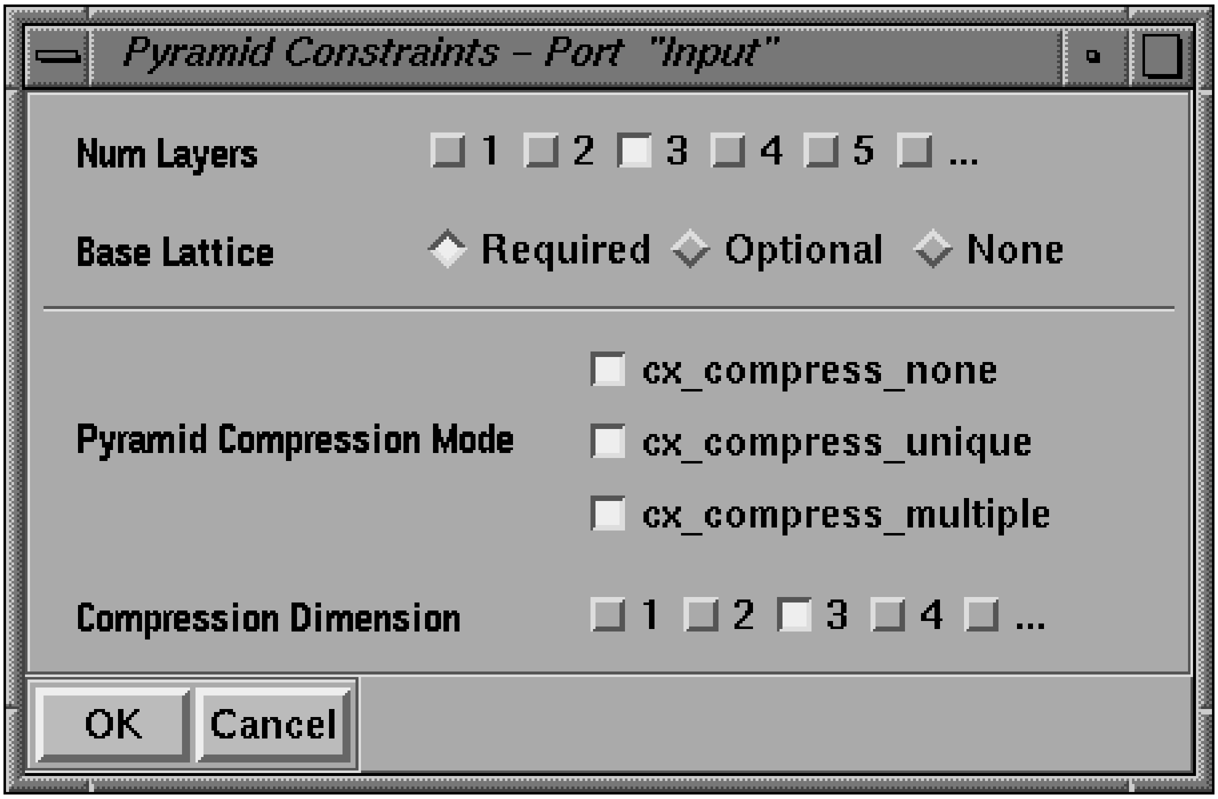


Figure 2–5 The Pyramid Constraints Window

Pyramid Compression mode
You can select one or more compression modes. You can use *cx_compress_none* ON and the other two options OFF to prevent the module from accepting or producing any compressed pyramids. The *cx_compress_unique* option accepts compressed pyramids based on a single type of element in a pyramid dictionary, such as a brick, and *cx_compress_multiple* accepts those made

Compression Dimension

Indicates the level at which compression takes place, for example, at the 1–D, 2–D, or 3–D level. The levels below this are specified only indirectly by reference to a pyramid dictionary.

Figure 2–5 shows these default settings for pyramid constraints:

- The pyramid must have four layers of lattices with connectivity information, that is three sets of lattice data and connections, and the base lattice.
- The base lattice, which is the 0th–level lattice, is required.
- The port can accept pyramids in all three compression modes, i.e., uncompressed pyramids and compressed pyramids containing a variety of elements.
- The port can accept pyramids that have been compressed at the third level. That is, compressed pyramids must be 3–D structures.

Returning to the Port Window

Click *OK* to have your constraints accepted and to close the window. You are returned to the Input Ports window. Click *OK* to accept the port specifications and return to the Module Builder main window.

2.4.4. Defining Output Ports

To display the Output Ports window (see **Figure 2–6**), click the *Output Ports* button on the Module Builder main window. The Map Editor uses the output port definition to decide whether a legal connection can be made between it and a given input port on another module.

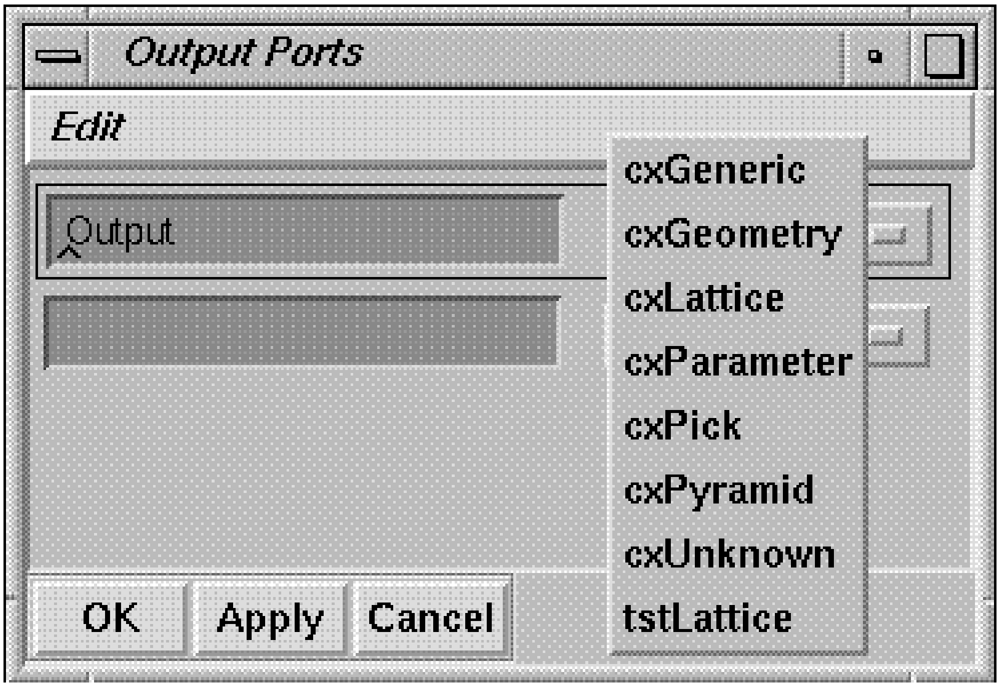


Figure 2–6 Output Port Window

Defining Port Characteristics

The Output Ports window resembles the Input Ports window, except that it has only two fields, the *port name* and the *data type.*

The example in **Figure 2–6** shows one output port with the port name "Output". The data type it produces is *cxGeometry*, this meaning that the output port must be connected to an input port that accepts geometry data. For more details on the Geometry data type, see **Chapter 5, "Using the Geometry Data Type"**.

If you are creating an output port that produces a lattice, you set the data type to be *cxLattice* and the Lattice Constraints window appears again. Set the constraints for the output lattice in exactly the same way as you would set those for the input lattice.

✰ Checking the Port Status

The Output Ports window does not require you to set the *status flag* because output ports are optional by definition; the module's firing pattern depends on the state of the input ports, not the output ports. A module that has the correct connections and data on its input ports will fire whether or not any output ports are wired.

You can write a smart module, that is, one that can track whether its outputs are wired and block certain computations if they are not connected. For example, the *Render* module has a camera output port for which data is calculated and output only if the port is wired.

You can write code for a module with two output ports to check which output is wired and compute the data for only that one.You can then write a ho*ok* function for "connect output" and call the API function *cxFireASAP* from that ho*ok* function so that when you wire up the previously unwired output, the module will fire and send the new data downstream.

For more details on hook functions see **Chapter 9, "Advanced Module Writing"**.

2.5. Defining Function Arguments

When you define the function arguments, you are specifying the interface between the module itself, defined here as the Module Control Wrapper (MCW), and the user function. The Module Builder uses this information to generate bridging code, which takes the form of the Module Data Wrapper (MDW). This is the module's *internal* interface.

You use the Build Options menu to specify whether or not you want the Module Builder to write the MDW for you (see **"Using the Build Menu"**.)

✰ You can take control of the interface and write your own interface code, in which case you do not need an MDW. If you write your own bridging code, you need not define the function arguments. Simply leave the Func Args and Connections windows blank.

For more details on writing wrapperless modules see **Chapter 9, "Advanced Module Writing"**.

You must enter the function name in the*Func Name* slot in this window (see **Figure 2–7** and **Figure 2–8**) even if the module has no MDW; otherwise, you will get an error message telling you that the user function name is missing.

2.5.1. Creating an Argument List

The Module Builder requires you to list each function argument and its correct type and referencing style so that the MDW can be written correctly. To display the Function Arguments (Func Args) window (see **Figure 2–7**), click the "Func Args" button on the Module Builder main window.

The examples in **Figure 2–7** and **Figure 2–8** show the argument lists for the *VectorGen* and *AtomicSurf* modules. The fields in the Func Args window are described below.

Func Name

The *Func Name* is the name of the user function or subroutine. It is repeated at the bottom of the window, along with the complete list of arguments. The Module Builder uses this field to create a call to the function.

Type the name in the text type–in slot exactly as it appears in the user function file.

Language

The *Language* button of the Function Args window allows you to select the language in which the user function or subroutine is written. The Module Builder uses this field to determine the calling conventions for the wrapper code (which is generated in C).

Select C, Fortran, or C++ from the Language pop–up menu.

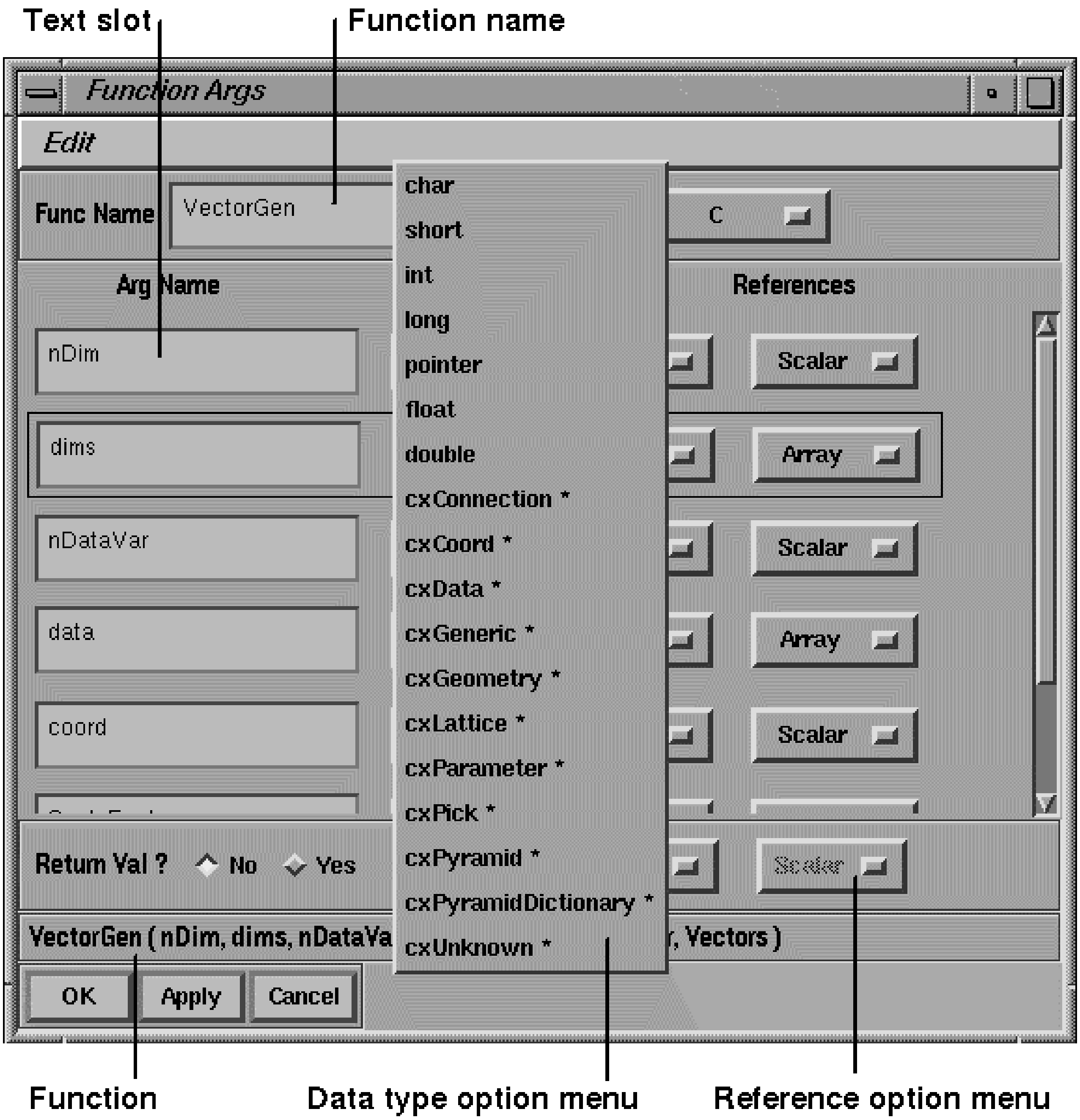


Figure 2–7 Function Arguments Window for a C Function

Arg Name

The *Arg Name* section of the Function Args window lists the function argument names, entered in the correct calling sequence. The Module Builder uses this list to establish the calling sequence of arguments for the function when it creates the Module Data Wrapper (MDW).

- Type the name of the first argument and press **`Enter`** to get a new slot.
- Use the Edit menu to copy or reposition function arguments in the list.
- Use *OK* or *Apply* to update the calling sequence when you have entered a new argument.
- To remove an argument from the calling sequence, simply delete the name of the argument from the list.

You can enter up to 45 function arguments per function. Each argument must have a unique name. The Module Builder renames duplicate arguments.

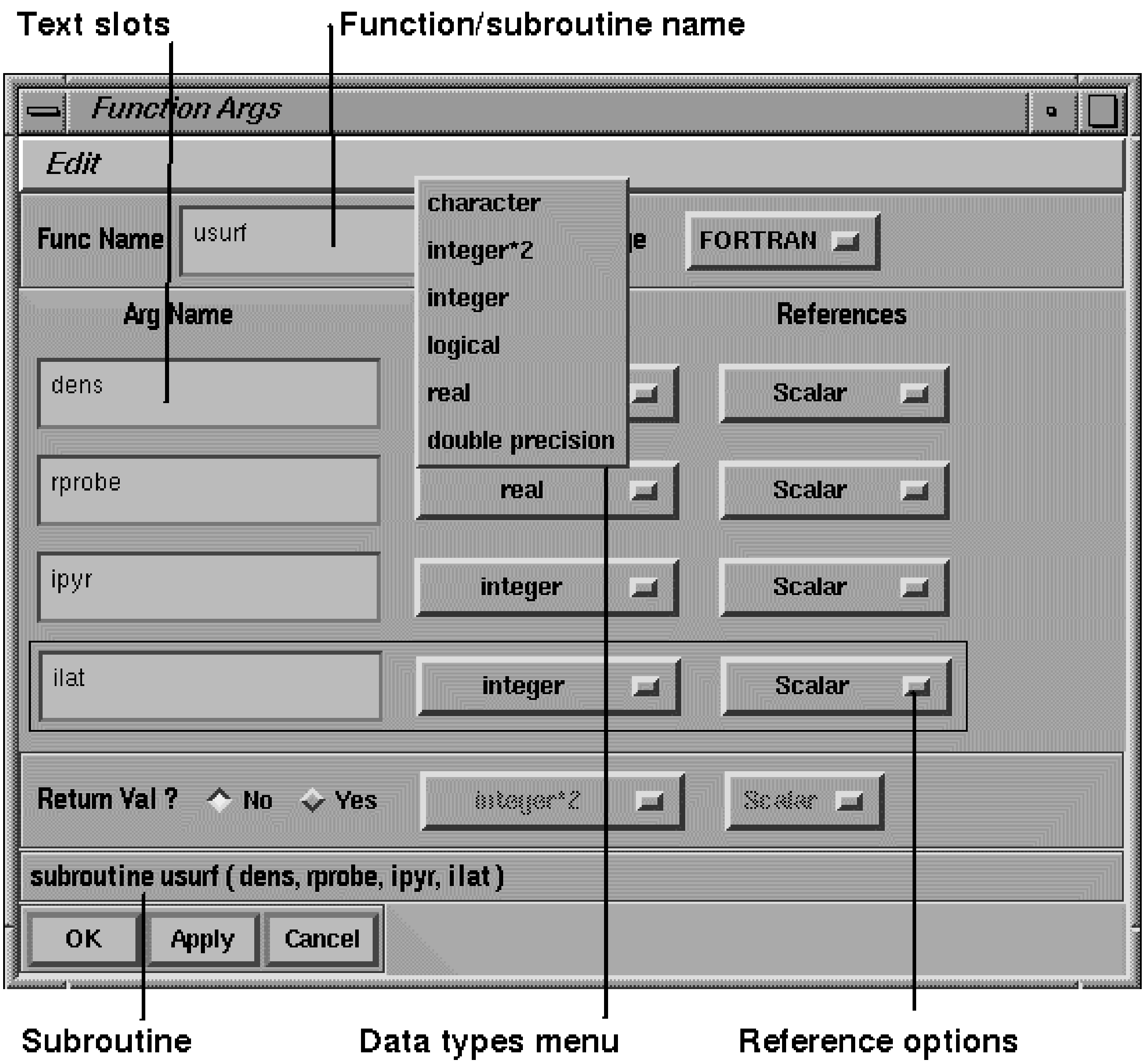


Figure 2–8 Function Arguments Window for a Fortran Subroutine

Type

The *Type* section of the Function Args window lets you select a data type for each argument variable. These differ for C/C++ and Fortran.

Figure 2–7 shows the data type options menu for a C function. The menu is the same for C++ functions. **Figure 2–8** shows the options for a Fortran subroutine.

Select one option from the menu for each argument.

Table 2–2 shows the relationship between the C and Fortran primitive data types that IRIS Explorer recognizes.

C and C++	Fortran
char (byte)	character*(1)
short	integer*2
int	integer
long	integer
int	logical
float	real
double	double precision
pointer	integer
cx*	integer

Table 2–2 Primitive Data Types in C and Fortran

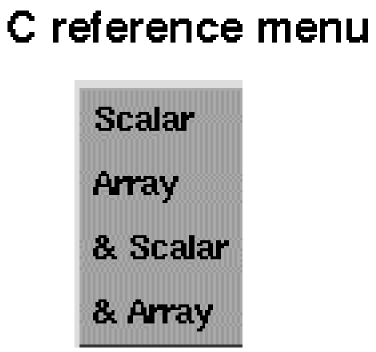
✰ In the C and C++ options menu, IRIS Explorer data types or subsidiary types (*cx<DatatypeName>*) are followed by an asterisk (*). They are automatically passed by reference because they are arrays written in C. In Fortran, you refer to the IRIS Explorer data types by using integers that contain a pointer to the C data structure. The C structures that are referenced can be manipulated through the Fortran API, but not directly in the syntax of the language. The pointers are uniformly treated as "opaque handles" that are passed around in integer variables.

References

The References section of the Function Args window lets you set the method whereby the argument variable is passed. **Figure 2–7** shows the data type options menu for a C function. The menu is the same for C++ functions. **Figure 2–8** shows the data type options menu for a Fortran function.

The choices are by value or reference in C or C++, by reference only in Fortran. **Table 2–3** shows the choices for C, C++, and Fortran.

Note: The term "references" is used here in the C sense, rather than the C++ sense, where it has a slightly different meaning.



C and C++	Fortran	Purpose
Scalar	Scalar	Used for passing in a single value. Scalars are passed by value in C and C++
Array	Array	Used for passing in or modifying a data array. Expects multiple values. Arrays are passed by reference in C and C++.
& Scalar	Footnote: Because Fortran always passes parameters by reference, there is only one option, *Scalar*, for passing Fortran scalar values.	Used for passing a value out by means of the argument list. Passes a pointer to a scalar.
& Array	Array Pointer	Used for passing out an array by passing a pointer to an array reference, such as an array allocated in the function. Also useful for returning a pointer to allocated storage.

Table 2–3 Function Argument References

Return Val?

The choices in the Return Val? section of the Function Args window indicate whether the user function or subroutine returns a value. If it does, you must select a type and reference as you did for each argument. The return value can be used when you set up connections from the function arguments to the output ports (see **"Connecting Arguments to Ports"** below).

When you have entered and defined all the function arguments in their correct calling sequence, click *OK* to save the argument list and close the window.

2.5.2. Connecting Arguments to Ports

The user function arguments, once defined, must be associated with the input and output ports so that incoming data goes to the correct function arguments for processing before being passed in turn to an output port.

To display the Connections window, click on the "Connections" button on the Module Builder main window. The Connections window displays graphically the links between inputs, function arguments, and outputs. When you first create a module, no connection wires are shown. The wires appear after you have used the mouse to make a link between an argument and a port.

Figure 2–9 shows the connections between the function arguments for *VectorGen* and its ports. There are two input ports, "Input" and "Scale Factor", and one output port, "Output". The function arguments are listed in the center column. The open port menu in the center of the screen is the connections menu for "Scale Factor", which passes the *parameter* data type.

There are also three "pseudo" input ports listed under Input Ports. These are described in **"Using Pseudo Input Ports"** .

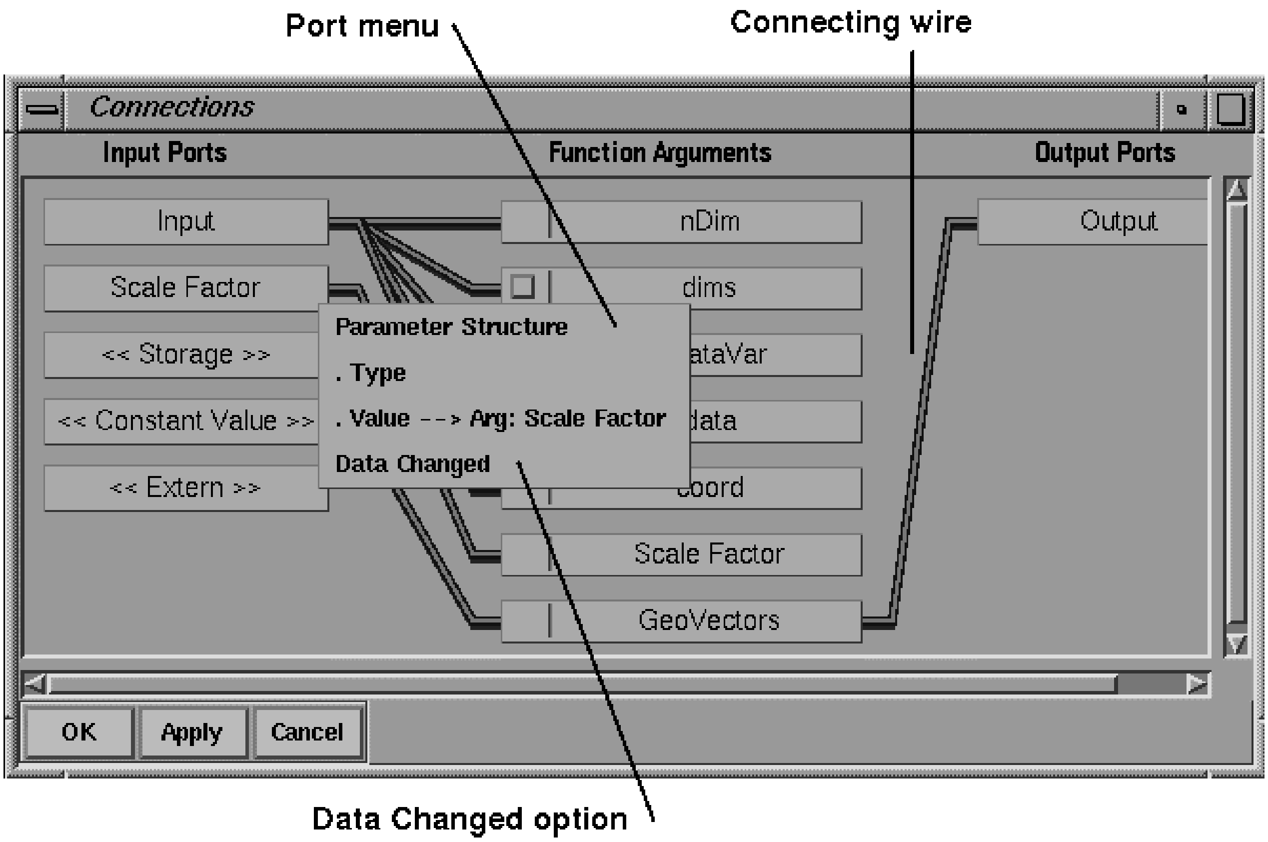


Figure 2–9 Connections Window

Understanding the Connections Menu

Each port has a connections menu that gives its data type structure and lists the constituent elements of the data type. The number of dots in front of each element name indicates how many levels below the top–level data structure you have penetrated.

For example, in **Figure 2–9** it is clear from the menu that a link has been made between the Value element in the Parameter structure and the *Scale Factor* function argument. This means that any data value passed from an upstream module to the "Scale Factor" input port of *VectorGen* is sent in turn to the *Scale Factor* function argument. The Value element has one dot in front of it; it is one level below the Parameter structure and on a par with the Type element.

Lattice, pyramid, and pick data types have several substructures that can be wired to or from function arguments. **Figure 2–10** shows the Connections window for *AtomicSurf*, which has an input port, "Molecule", that accepts the pyramid data type. Part of the connections menu for this port is shown. As you can see, *Pyramid Structure*, which includes the entire contents of the data type, is connected to the function argument *ipyr*.

The other items on the menu are the substructures of the pyramid data type (more accurately, of the base lattice of the pyramid. For more on the pyramid structure, see **Chapter 4, "Using the Pyramid Data Type"**). They include:

- individual scalars, such as the number of variables per node (*Num Data Variables*)
- arrays such as the *dims* vector (*Num Dimensions*) and the *data* and *coordinate* arrays (*Data Array* and *Coord Array*)
- C data and coordinate structures, such as *Data Structure* and *Coord Structure*, of which the scalars and arrays are constituents

You can write function arguments that accept data from any one of these substructures if you wish.

You can use a pointer to the C structure to pass a complete IRIS Explorer data type directly to a function argument if the user function is written in Fortran or the function argument is broken into smaller elements of the data type.

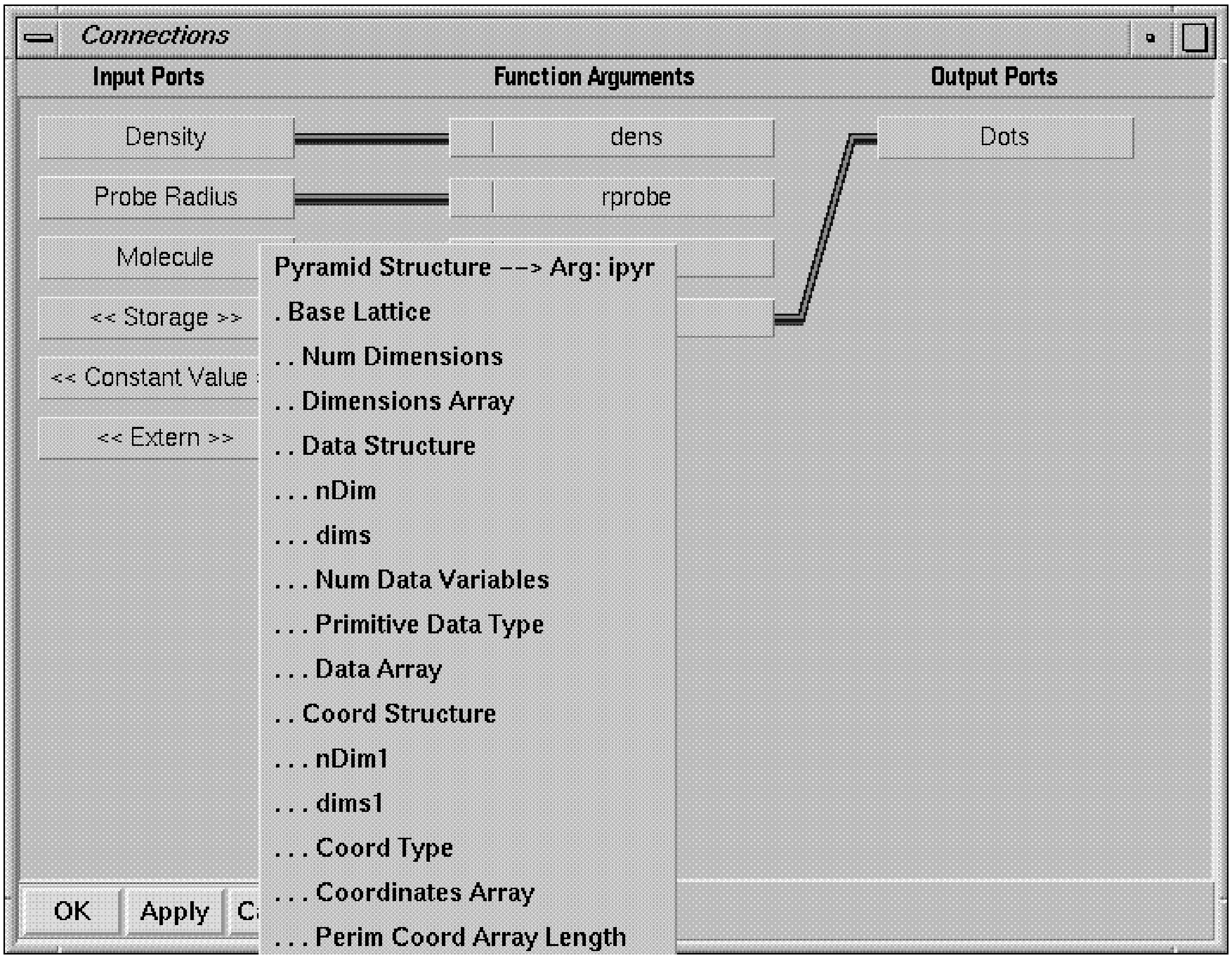


Figure 2–10 Connections Menu for the "Molecule" Port

Creating a Link

The physical mechanism of wiring ports and functions together and disconnecting them again is exactly the same as that used in the Map Editor. Click on an input port with the right mouse button, select the element to be wired, and then click on the function argument (or output port). The Connections menu for function arguments has only one item, *Select* . To break a connection, simply reverse the process.

The "-->" icons in the connections menus show which elements of an input or output port are associated with each other or with function arguments.

You can make only one connection to a given function argument from an input port, and only one connection from a given function argument to an output port; however, you can make several connections *from* an input port or *to* an output port. For example, the "Input" input port in **Figure 2–9** has four connections, one to each of four function arguments.

All function arguments must have a connection to either an input or an output port, but all ports do not necessarily have to be connected.

Highlighted Connections

Figure 2–11 shows how connection options are highlighted in the Connections window. There are two kinds of highlighting:

- When you select a port menu item or function argument as the first member of a connection, the selected item turns light green, and the available options from which you can select the second member turn darker green.

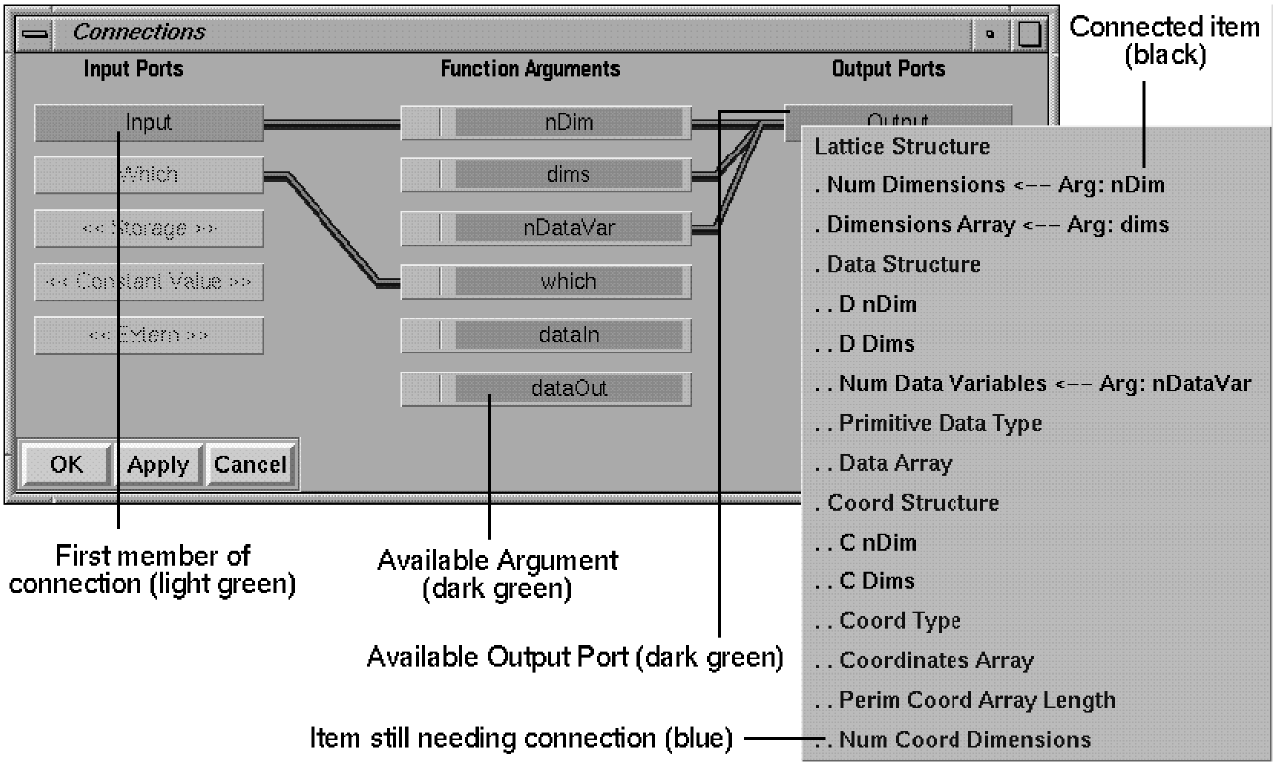


Figure 2–11 Highlights in the Connections Window

Unavailable items are grayed out (in **Figure 2−11**, these are the remaining input ports).

- When an output port lacks a connection required for the integrity of the data type on the port, the connections menu lists the relevant item in blue. When you first open the Connections window, no connections exist and the entire list of data structures is blue. As connections to the output port are made, the connected structure is listed in black.

  This color coding helps you to see whether the output port will receive all the data it needs to construct a complete data type.

Passing Default Values

In many cases, you can connect the top−level data structure on the input port directly to the top−level data structure on the output port in order to pass default values. For example, in a module that accepts a lattice on an input port and outputs a lattice on an output port, you can connect the *cxLattice* structure at the top of the input port menu to the *cxLattice* structure at the top of the output port menu.

In this way, all data that is not affected by a function argument will be passed by default from the input port to the output port. The members of *cxLattice* that were operated on by function arguments will acquire new data that is passed in turn to the output port to replace the old data. See, for example, the Connections window for the *ChannelSelector* module in **Figure 1−10**, where most of the parts of the input lattice are passed directly to the output lattice, while *nDataVar* and the data part of the output lattice are set explicitly via the function arguments.

Using Pseudo Input Ports

At the bottom of the Input Ports list are the three so−called "pseudo" input ports, <<Storage>>, <<Constant Value>>, and <<Extern>> (see **Figure 2−12**). They cannot be wired directly to output ports, unlike the regular input ports. In most circumstances you would not need to use the facilities of pseudo−ports, but they can be particularly useful if you do not wish to re−write or are unable to change your existing source code. For example, this would be the case if your existing source code has function parameters that have constant values in the context of your module. You would set these values using the <<Constant Value>> pseudo−port.

You can connect the pseudo−ports to function arguments to do these things:

- allocate storage for the user function: <<Storage>>
- set function arguments to constant values: <<Constant Value>>
- ✰ allow the user function to reference an external function: <<Extern>>

Once you have wired the <<Storage>> or <<Constant Value>> ports to a function argument, the argument's connection menu acquires a second item. To set the argument, click on it after the connection has been made.

The second item, *Set Storage* or *Set Constant Value*, when selected, invokes the "Set argument value" window. **Figure 2−12** shows the connection between the <<Constant Value>> input port and the *GeoVectors* function argument in *VectorGen*, the *GeoVectors* connections menu, and the value set (NULL, which is the default). See **"Allocating Storage Space"** for more information about temporary storage.

You can enter any legal C or Fortran expression in the text slot, including references to library or user−supplied routines that can be resolved at link time. It is a good idea to comment on your entries for future reference.

<<Extern>> can be wired to only a function argument whose variable name refers to the name of another function or subroutine. This function must be in a module source file and referenced in the User Function File slot in the Module Builder main window (see **Figure 2−1**).

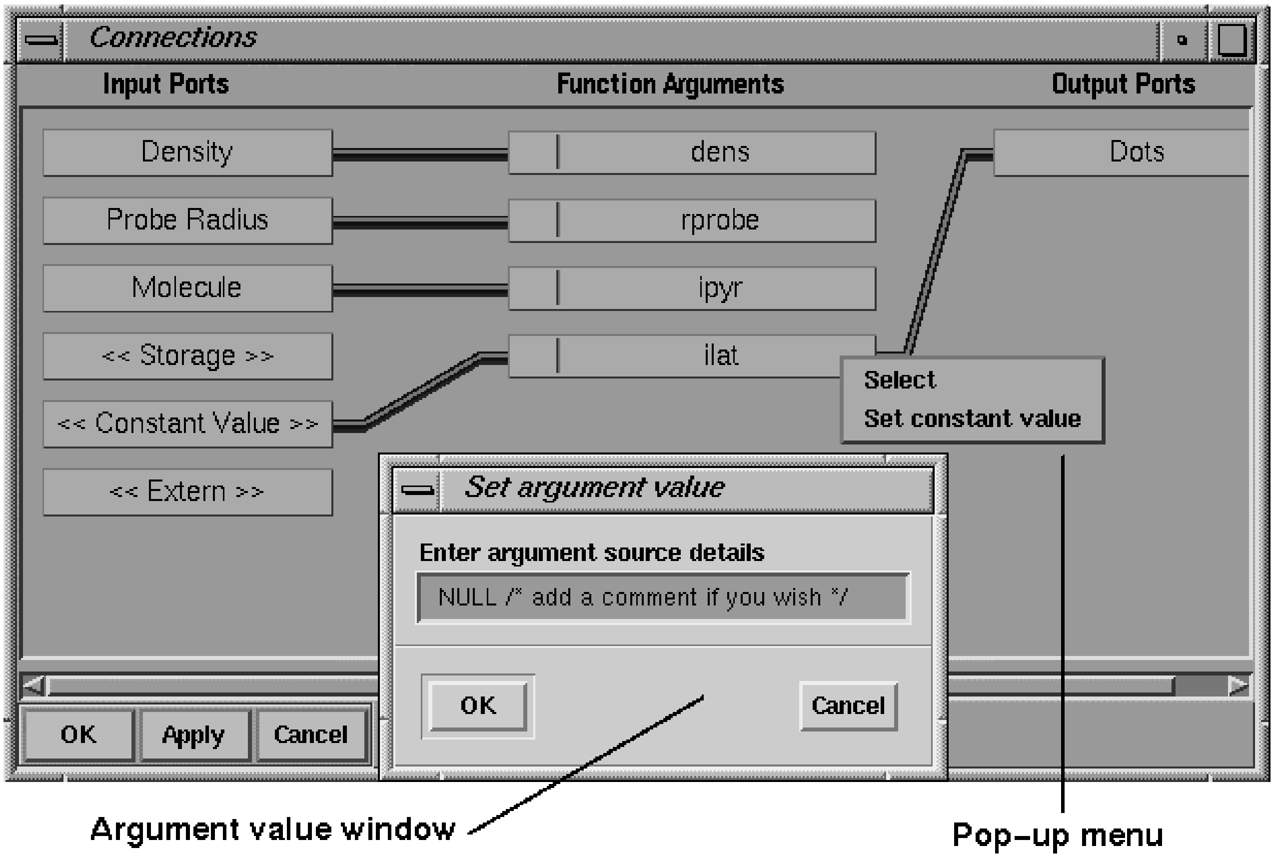


Figure 2–12 Setting a Constant Value for a Function Argument

You can use <<Extern>>:

- to reference numerical analysis routines. For example, you can specify the name of a routine that executes a particular action, such as a convergence test. The name of the function argument should be identical to the function name. The function argument is then the function returning a long scalar.
- in an integrator module, to pass in either a function to be integrated, or a derivative function.

Here is an example of how to use <<Extern>>. Suppose you have a function named *calcDerivative* and you want the user function to use this derivative calculating function in its global optimization calculation. You would need to:

- set a function argument name to *calcDerivative*
- wire <<Extern>> to the *calcDerivative* function argument
- treat that function argument as a function or subroutine in the user function.

Mapping Rules

These are the basic rules to follow when connecting ports and arguments:

1. The mapping to an output port must describe the contents of the output data structure fully. Every element in an output port should have a source, either a function argument or an input port; otherwise the Module Data Wrapper may not have enough information to build the data set and errors will appear (not immediately, where they can be picked up, but later, and apparently mysteriously).
2. On port menus, you can wire to the data type structure itself or to all its component elements. For example, instead of wiring to *Pyramid Structure* in **Figure 2–10**, you can wire to all the substructures listed as separate elements.

Similarly, you can wire to *Data Structure* or you can wire to *Num Data Variables*, *Primitive Data Type*, and *Data Array*, which are subelements of the *Data Structure* component.

3. Wire input ports to function arguments and/or output ports. Ports can have multiple connections (one to each part of the data structure on the port menu). Function arguments can have only one input and only one output (but need not have any outputs).
4. If you make more than one connection to a function argument or a port menu element, you may duplicate and/or overwrite data as it is passed, causing errors. In particular, if you wire more than one connection to an output port element, you run the risk of leaking memory.
5. Every function argument must be connected to an input port, a pseudo input port, or an output port. You can leave only one unconnected if you do not intend to use it.
6. You can wire an input lattice data type to an output data type, but the values of *nDim* (Num Dimensions) and *dims* (Dimensions Array) cannot be changed. This is a mistake easily made because the data and coordinate substructures (*cxData* and *cxCoord*) each have a copy of these values.

✰ Checking the State of Input Ports

At the bottom of each connections menu is the *Data Changed* item (see **Figure 2–9**). You can use it to record whether the module received new data on that particular port during the most recent firing. Depending on the answer, the user function may then act to collect new data from the port, or ignore the old data. *Data Changed* returns a long integer with the value 0 (to indicate old data) or 1 (to indicate that new data has arrived on the port).

You can wire *Data Changed* to any scalar function argument.

✰ Copying Arrays

Arrays listed as function arguments in the Connections window have *Copy* buttons that let you choose between copying pointers to arrays and the arrays themselves, depending on whether data is to be overwritten or not. The *Copy* button appears at the left side of the array function argument. When you click a Copy button, it is lit and the word "Copy" is displayed (see **Figure 2–13**).

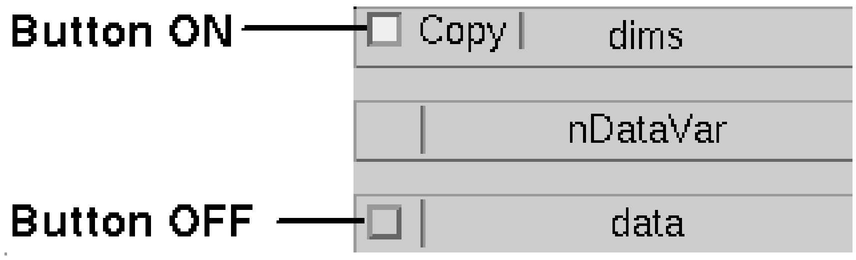


Figure 2–13 Copy Buttons

This copy facility is important if you intend to modify the contents of an argument whose original values come from an input port. Since data from an input port resides in shared memory, it cannot be written into without polluting the data space of other modules and causing a module in the map to fail.

When the *Copy* button is ON, it tells the Module Builder to make a working copy of the array so that you can modify it. This is useful even if you intend to use it only as a temporary working space, without writing it to an output port.

For example, in a module that manipulates the grid data of a lattice without changing the coordinates, data will be overwritten; therefore, the *Copy* button should be turned ON. If the *Copy* button is OFF, a pointer to the array is passed.

Scalars are always copied before being passed to the user function.

☆ Allocating Storage Space

You can use the <<Storage>> pseudo input port to allocate temporary storage space which may be used inside the user function. The space can be used as follows:

- For temporary storage in the user function. This space is allocated by a <<Storage>> connection to a function argument but not wired to an output port. This may be especially useful for Fortran programmers who wish to use dynamically allocated memory.
- To hold data that is entered in the user function and then assigned into an output port, where the storage allocation is not done explicitly inside the user function.

Two API routines are useful for computing the size of arrays for wiring to the storage allocator (see the*IRIS Explorer Reference Pages* or the on−line *man* pages for details):

- *cxDimsProd* computes the total number of data variables in a lattice.
- *cxDimsSum* computes the length of the perimeter coordinate vector for a lattice.

After the user function has been invoked, the allocated storage is freed. The Module Data Wrapper automatically frees working space that has been held temporarily by the user function. The Module Builder uses reference counting to maintain temporaries assigned to output ports as part of the output port list.

You must free any temporaries that you create inside the user function.

2.6. Creating File Prototypes

You can use the Prototypes menu in the Module Builder main window (see **Figure 2−14**) to create a prototype user function and a help file for your new module.

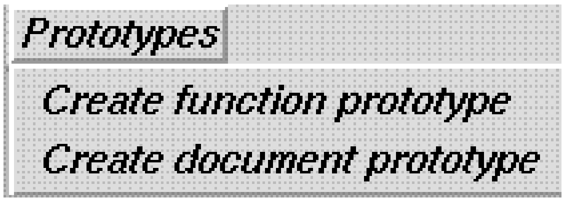


Figure 2−14 The Prototypes Menu

2.6.1. The User Function Prototype

The Module Builder will create a user function prototype for you if you have not already written it, or do not have existing code for it. You must already have named the function and laid out the calling sequence of the function arguments in the Func Args window. The Module Builder uses this information to write a stub function that is the user function as described, but without any action.

When you select *Create function prototype* from the Prototypes menu, a window appears, in which you enter the name you want for the new user function (see **Figure 2−15**). The file is created in your working subdirectory, along with the other module files. If a file with the name you have selected already exists, the prototype file is renamed < *filename>.use*r.

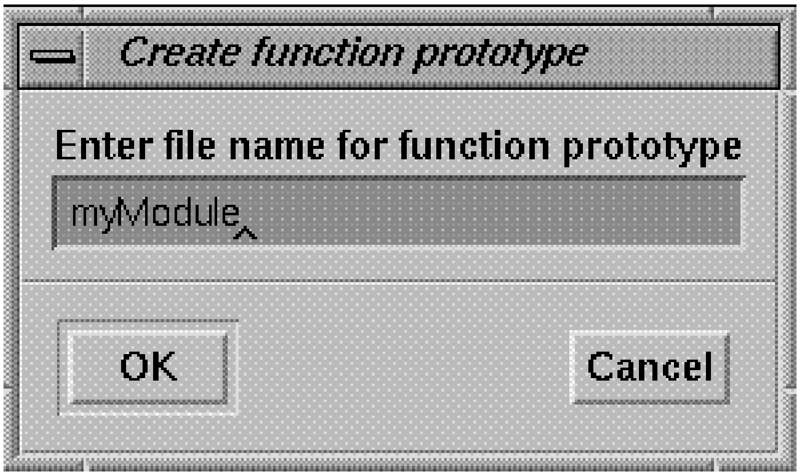


Figure 2–15 Function Prototype Window

This example shows a basic prototype for a user function *MyModule.c*. The module has lattice input and output ports and parameter input ports. The user function is written in C and has 12 function arguments.

```
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>

#ifdef __cplusplus
 extern "C" {
#endif

 void MyModulec (
 long    dim,
 long    dim,
 long    dim,
 long    *dims,
 long    nDataVar,
 double  *dataArray,
 long    nCoordVar,
 float   *coordArray,
 long    *nDimOut,
 long    *dimsOut,
 double  *dataOut,
 float   *coordOut );

#ifdef __cplusplus
}
#endif

/* ------------------------------------------- */
/* ------------------------------------------- */

 void MyModulec (
 long         Idim,
 long         Jdim,
 long         Kdim,
```

```
 long         *dims,
 long         nDataVar,
 double       *dataArray,
 long         nCoordVar,
 float        *coordArray,
 long         *nDimOut,
 long         *dimsOut,
 double       *dataOut,
 float        *coordOut )
{
}
```

Once you have the prototype file, you can open it and finish writing the user function at your leisure. If you create the prototype file before you have defined the function in the Module Builder, the file will not contain the function arguments.

2.6.2. Hook Function Prototypes

You can generate a function prototype for each hook function you want to write for a module. To create the hook function, use the "Hook Funcs" option on the Build menu (for details, refer to **"Hook Functions".** Then select *Create function prototype* from the Prototypes menu to create the hook function prototypes.

Here are examples of the two types of hook function prototype in Fortran.

```
C Hook function for IRIS Explorer module actions. /*Remove hook /*
      SUBROUTINE FREMOVE()
      print *,'Remove hook function called.'

      RETURN
      END

C Hook function for IRIS Explorer module actions. /* Connection /*
      SUBROUTINE FCONNIN( PORTNAME, LINKTAG)
      CHARACTER*(*) PORTNAME
      *INTEGER LINKTAG
      print *,'Connect input hook function called: ',
     1*PORTNAME, LINKTAG

      RETURN
      END
```

2.6.3. The Help File

If you want the Module Builder to create a help file template for you, select *Create document prototype* from the Prototypes menu on the Module. The help file is saved in your working directory with the name *<modulename>.doc.*

A skeleton file contains the port names, data types, and other variables that you have specified during module building. You should add any descriptive information you want users to read when they invoke help for your module. This information should include a description of the module's algorithm, ports, known problems or limitations, and related modules. Be careful not to disturb any line beginning with %%.

This example shows what the prototype help file for*MyModule.c*, called *MyModule.doc*, would look like.

```
MyModule
%%------------ DESCRIPTION ------------
<Replace this with your module description>
```

```
%%------------ INPUTS ------------
%PORT Input 1-D
%TYPE Lattice
%CONSTRAINT 1-D
%CONSTRAINT curvilinear
<Describe the purpose of the port here>
%PORT I dimension
%TYPE Parameter
<Describe the purpose of the port here>
%PORT J dimension
%TYPE Parameter
<Describe the purpose of the port here>
%PORT K dimension
%TYPE Parameter
<Describe the purpose of the port here>
%%------------ WIDGETS ------------
%%------------ OUTPUTS ------------
%PORT Output 3-D
%TYPE Lattice
%CONSTRAINT 3-D
%CONSTRAINT curvilinear
<Describe the purpose of the port here>
%%------------ PROBLEMS ------------
<Describe any known problems/limitations here>
%%------------ SEEALSO ------------
<List associated or related modules here>
```

If you are modifying an existing module and changing some aspect of its control panel (such as using different widgets or new ports), select *Update document* from the Build Options menu (see **Figure 2–16**) to update the existing *<modulename>.doc* file and preserve any existing annotation.

2.7. Using the Build Menu

The Build menu (see **Figure 2–16**) provides a way to set some defaults for the module and to choose the directory in which modules will be built and installed: in your working directory, or in another directory of your choice. All the menu options except "Hook Functions" are discussed below. Hook functions are discussed in **"Hook Functions".**

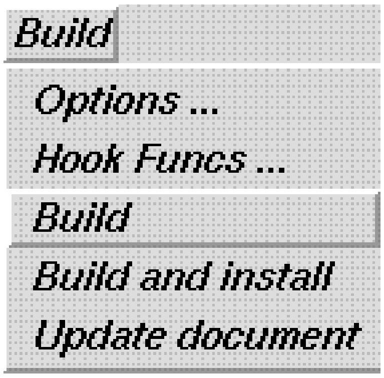


Figure 2–16 The Build Menu

2.7.1. Selecting Build Options

You can set a number of options in the Build Options menu (see **Figure 2–17**):

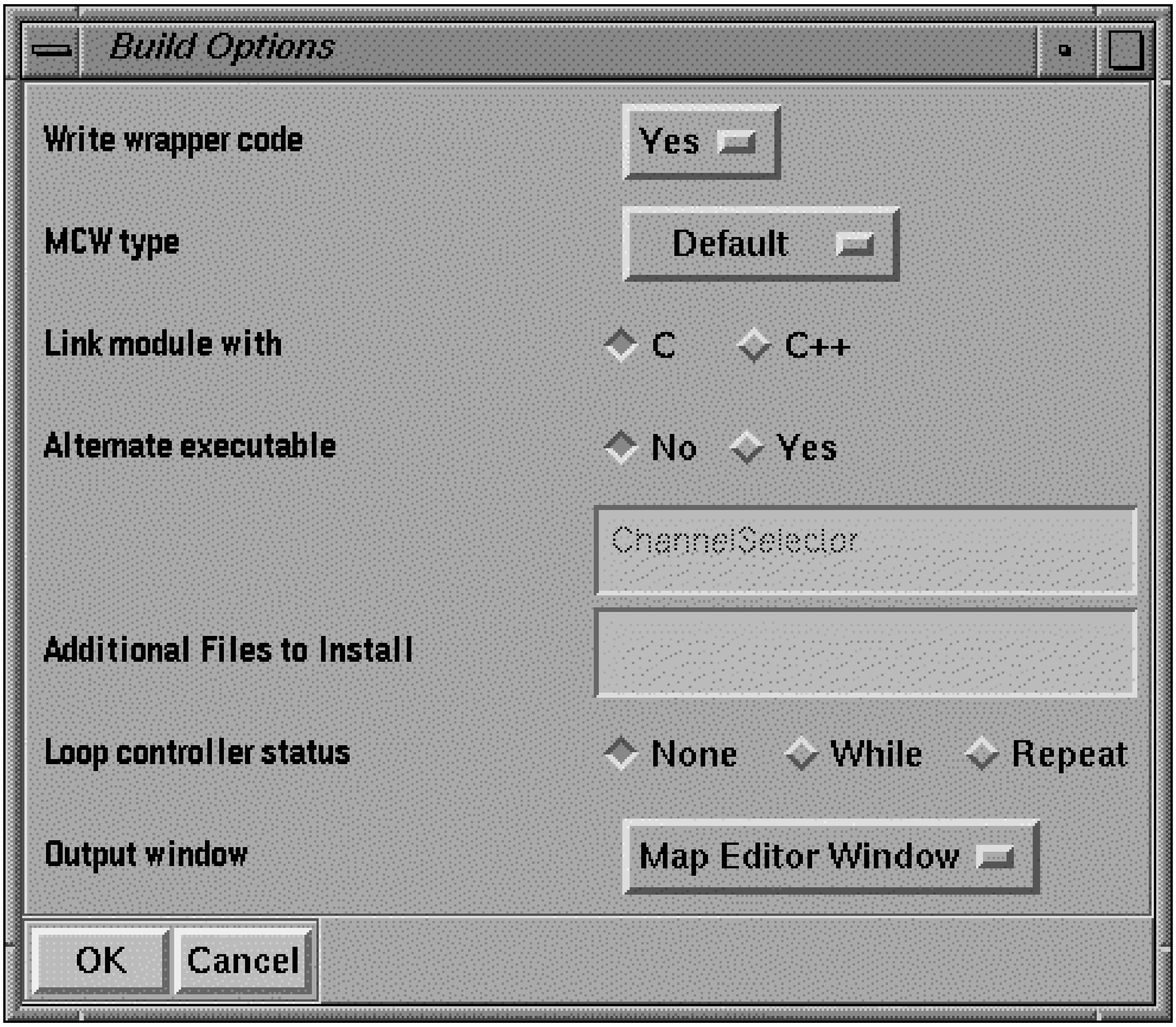


Figure 2–17 The Build Options Menu

Write wrapper code

Lets you select an automatically generated Module Data Wrapper. The default is *Yes.*

MCW type

Lets you select the type of Module Control Wrapper (MCW) for linking. Most modules link against the default MCW, but a few manipulate windows in the X Window System independently of the IRIS Explorer graphical user interface (see examples in **"Working with Module Wrappers"**). These modules should be linked against the X–MCW.

Link module with

Lets you link the module with either C or C++.

Alternate executable

Lets you specify whether your module has its own executable or whether it will use the same executable as some other modules in order to save disk space. For more information, refer to **"Sharing Module Executables".** The default is *No*.

Additional Files to Install

Lets you stipulate any script or program files you want installed with your module. Used when you are writing an interpreter such as a *LatFunction*–based module. For more information, see **"Building and Installing Modules"** and also **Chapter 10, "Module prototyping with Shape"**.

Loop controller status

Lets you determine whether or not the module is a loop controller, and if so, whether it controls a "while" loop or a "repeat" loop. For more information on loop controller modules, see **"Building Loop Controller Modules".** The default is *None*.

Output window

Lets you select the destination for standard output and error reporting from the module. By default, modules send printed output to the shell invoked by the Map Editor. Selecting *Console Window* directs the standard out and standard error streams to */dev/console* in the workstation's console window. This selection saves one file descriptor per module; it may allow more modules to be invoked. Do not use "Console Window" output with a module using the X–MCW.

2.7.2. Building and Installing Modules

The last three options on the Build menu do not have any menu choices. The *Build* and *Build and install* options simply initiate the compilation and installation process on the module. The *Update document* option tries to update the module*modulename.mres* file and produces an error message in the shell window if it fails.

Build builds the module in the current working directory.

Build and install installs a module in */usr/explorer/modules* by default, irrespective of where it is built. If you want a module to reside in a directory other than */usr/explorer/modules*, you need to set the *EXPLORERUSERHOME* environment variable before you start the Module Builder. This variable is described in **"Configuring the Build Environment"**.

It is often more convenient to use the *Build and install* command on the Build menu rather than *Build*, since this will install your completed module in the correct directory.

☆Some modules, particularly those that contain an interpreter that reads a script file, have auxiliary files that need to be installed when the module is installed. For instance, *LatFunction*–based modules have a script file (named in the "Program File" widget) that should be installed to go along with the module. This is automatically handled by the interpreter option when you specify an alternate executable, but you can use the Build Options menu to install other files as well if you wish (see **"Selecting Build Options".**)

2.7.3. Updating the Module Resources

The *Update document* option updates both the *modulename.mres* file and the document help file (*.doc*). You must have write permission on the module's *modulename.mres* file to do this.

The document file is overwritten and the original *.doc* file is saved as a *.bak* file. If you have information in the old file that you want retained in the new file, you must edit the new *.doc* file accordingly. These are ASCII files, so this is easily done.

The *modulename.mres* file is overwritten without any backup file being saved.

2.8. Configuring the Build Environment

The Module Builder handles the details of the module building and installation environment, but it does not handle creation of data types. These you must make yourself (see **Chapter 8, "Creating User–defined Data Types".**) When you do so, you may want to designate a different installation directory for your new data types and modules.

2.8.1. Defining EXPLORERUSERHOME

You must define one very important environment variable to indicate to the Module Builder where new modules and type files are to be installed:

EXPLORERUSERHOME
: name of the directory that is the root of the user's installation tree.

Typically, the value of this variable would be */usr/local/explorer* or perhaps *$HOME/explorer*. For example, in the C−shell you would type this command:

```
setenv EXPLORERUSERHOME /usr/local/explorer
```

When modules and types are built, the generated *Makefile* installs a number of items under *EXPLORERUSERHOME*, creating subdirectories as necessary. This is the directory structure created under *$EXPLORERUSERHOME*:

$EXPLORERUSERHOME/

- *modules/* — All the necessary components for modules, including the module resource (modulename.mres) file and the module executable.
- *types/* — The description files for user−defined types. These always have the filename suffix *.type*.
- *lib/* — The automatically generated API library for user−defined types.
- *include/cx/* — The C and Fortran header files for user−defined types.
- *man/*
  - *man1/* — The manual pages for user modules.
  - *man3/* — The manual pages for the user−defined types (UDT) library.

2.8.2. Setting Environment Variables

A few environment variables control alternatives when modules are built and installed. They need not have any particular value, and they take effect:

- *only* if defined
- after you rebuild the Makefile (see **"Creating a Makefile"**)

The variables are:

CXBUILDSHARED
: When modules are linked, they will be linked with the IRIS Explorer shared libraries. This results in a substantial savings in disk space and run−time memory usage. However, IRIS Explorer libraries are incompatible from release to release, so modules linked this way are guaranteed to be incompatible across releases.

CXINSTALLSYMBOLIC
: When you use the *make install* command, modules will be installed using symbolic links, rather than being copied. This can result in substantial disk space savings.

CXINSTALLSTRIPPED
: The symbol table will be removed from the module executable when it is installed. Although you gain disk space, you are no longer able to debug the module.

DEBUG
: Modules will be compiled with the debug option, *−g*. IRIS Explorer modules are compiled by default with the compiler's optimizer option, *−O*.
You must set the DEBUG option *before* you make the module *Makefile* if you want to be able to debug your module. For more information, see **"Debugging a Module"**.

2.8.3. Creating a Makefile

When you issue the *make install* command in a directory used for building modules or data types, the files are created and installed in the directories according to the environment variables you have set. If you ever change the value of these variables, you must recreate the *Makefile* in your directory to reflect the new value.

You can do this with the command:

```
make Makefile
```

If you do not already have a *Makefile*, you must use the command:

```
cxmkmf
```

This command makes you a new *Makefile*, which you can use to recreate your modules and type−related files.

Modules built in a directory are listed in a special file named *MODULES*; the Module Builder creates this file for you. New data types must be listed in a file named *TYPES*, which you must create by hand, since there is no mechanism for listing new data types automatically.

The *TYPES* file must contain the names of the data type files to be processed in the current directory, but without the *t* suffix. For example, if you have data type files named *NewType.t* and *MyType.t*, then you must create a *TYPES* file that contains the lines:

```
NewType
MyType
```

For more on *.t* files, see **Chapter 8, "Creating User−defined Data Types"**.

Both the *MODULES* and the *TYPES* files must be present before you run *cxmkmf*. Then *cxmkmf* creates the *Makefile* commands required to build and install code properly for these data types.

For more information on *Makefile*s, see **Appendix A, "Using Makefiles"**.

2.8.4. Debugging a Module

To compile a module with the *−g* option for debugging purposes, set the *DEBUG* environment variable before you issue the *cxmkmf* command.

More specfically, suppose the module which you wish to debug is called *MyModule*. When you are ready to make the module, type these commands:

```
setenv DEBUG 1
cxmkmf
make MyModule
```

You can then debug it. To do this, start IRIS Explorer and bring *MyModule* (along with any other modules that you may require) into the Map Editor. This starts the module running as a process called *MyModule* (you can look for it using the UNIX *ps(1)* command). Then, in a separate shell window, bring up your favorite debugger and attach it to the running process (your debugger may require the process id rather than its name; this can be obtained from the output of *ps(1)*). Using *dbx*, for example:

```
dbx −P MyModule
```

Upon attaching the debugger to the process, you will probably find that the process is sleeping within the *select* system call. In this state, the module is awaiting new input before firing. Set a breakpoint at a suitable point within your module (for example, the entry point to the user function), then tell the debugger to continue. Again, using *dbx*, for example:

```
Process 13832 (MyModule) stopped at [_select:12 +0x8,0xfabcfd8]
         Source (of select.s) not available for Process 13832
(dbx) stop in mymod
Process 13832: [3] stop in mymod
(dbx) cont
```

Now fire the module by passing it new input data. The debugger will stop execution at your breakpoint, and you can start to debug your code.

2.9. Examples of Simple Modules

Here are user functions for two simple modules, coded in C and in Fortran. The first one is a simple pass−through module, and the second is a channel−select module. The source code for these modules is in */usr/explorer/src/MWGcode/ModuleBuilder/C/** and */usr/explorer/src/MWGcode/ModuleBuilder/Fortran/**.

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

2.9.1. The Pass−through Module

This module copies all lattice input data to the output lattice, doubling the data values if they are of type float.

C Version:

```
/* Example module to show the use of some lattice API functions.
 *
 * Input lattice coordinates are copied to the output lattice
 * Input lattice float data values are doubled for the output lattice
 * Other input lattice data values are copied to the output lattice
 */

#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/cxLattice.api.h>
#include <cx/Typedefs.h>

void pass(cxLattice *inlat, cxLattice  **outlat)
{
  float          *outData;
  long            nDim, *dims, nDV, nCD;
  long            num_pts, i, hasData, hasCoord;
  cxData         *Data;
  cxPrimType      ptype;
  cxCoordType     ctype;

  /* Extract descriptive information about inlat */
  cxLatDescGet(inlat, &nDim, &dims, &hasData, &nDV, &ptype, &hasCoord,
        &nCD, &ctype);

  /* Duplicate the coordinates and data of inlat to outlat
   * if the input data are not of type float
   */
  if (ptype != cx_prim_float)
    *outlat = cxLatDup(inlat, 1, 1);
  else
    {
      /* Duplicate the coordinates of inlat to outlat
       * if the input data are of type float
       */
```

```
      *outlat = cxLatDup(inlat, 0, 1);

      /* Extract pointers to data structure and float data in inlat */
      cxLatPtrGet(inlat, &Data, NULL, NULL, NULL);

      /* Insert a data structure into outlat equal to that of inlat */
      cxLatPtrSet(*outlat, Data, NULL, NULL, NULL);

      /* Extract pointers to the float data in outlat */
      cxLatPtrGet(*outlat, NULL, (void **) &outData, NULL, NULL);

      /* Double all data elements for outlat */
      num_pts = cxDimsProd(nDim, dims, nDV);
      for (i = 0; i < num_pts; i++)
 *(outData + i) = 2.0 * (*(outData + i));
    }
}
```

Fortran Version:

```
C     Example module to show the use of some lattice API functions.
C
C     Input lattice coordinates are copied to the output lattice
C     Input lattice float data values are doubled for the output lattice
C     Other input lattice data values are copied to the output lattice
C
      SUBROUTINE PASS(INLAT,OUTLAT)
C
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
      INCLUDE '/usr/explorer/include/cx/Typedefs.inc'
C
C     .. Scalar Arguments ..
      INTEGER         INLAT, OUTLAT
C     .. Local Scalars ..
      INTEGER         CTYPE, DATPTR, HASCRD, HASDAT, I, IER, NCV,
     *                NDIM, NDV, NUMPTS, P0, PDIMS, POUTDT, PTYPE
C     .. Local Arrays ..
      REAL            OUTDT(1)
      INTEGER         DIMS(1)
C     .. External ..
      EXTERNAL        CXDIMSPROD, CXLATDESCGET, CXLATDUP, CXLATPTRGET,
     *                CXLATPTRSET
C     .. Pointers to Lattice Structures ..
      POINTER (POUTDT,OUTDT)
      POINTER (PDIMS,DIMS)
C     .. Executable Statements ..
C
C     Extract descriptive information about inlat
C
      IER = CXLATDESCGET(INLAT,NDIM,PDIMS,HASDAT,NDV,PTYPE,HASCRD,
     *      NCV,CTYPE)
C
C     Duplicate the coordinates and data of inlat to outlat
C     if the input data are not of type float
```

```
C
      IF (PTYPE.NE.CX_PRIM_FLOAT) THEN
         OUTLAT = CXLATDUP(INLAT,1,1)
      ELSE
C
C        Duplicate the coordinates of inlat to outlat
C        if the input data are of type float
C
         OUTLAT = CXLATDUP(INLAT,0,1)
C
C        Extract pointers to data structure and real data in inlat
C
         P0 = 0
         IER = CXLATPTRGET(INLAT,DATPTR,P0,P0,P0)
C
C        Insert a data structure into outlat equal to that of inlat
C
         P0 = 0
         IER = CXLATPTRSET(OUTLAT,DATPTR,%VAL(0),P0,%VAL(0))
C
C        Extract pointers to the real data in outlat
C
         P0 = 0
         IER = CXLATPTRGET(OUTLAT,P0,POUTDT,P0,P0)
C
C        Double all data elements for outlat
C
         NUMPTS = CXDIMSPROD(NDIM,DIMS,NDV)
         DO 20 I = 1, NUMPTS
            OUTDT(I) = 2.0*OUTDT(I)
   20    CONTINUE
      END IF
C
      RETURN
      END
```

2.9.2. The Channel−select Module

This module selects a channel.

In C:

```
/* Example module that does a channel select */

#include <cx/Typedefs.h>
#include <cx/DataTypes.h>
#include <cx/DataAccess.h>
#include <cx/UI.h>
#include <bstring.h>

void channel(long which, cxLattice *inLat, cxLattice **outLat)
{
  long xLo, xHi, i, nData, dataLen, w;
  long nDim,nDataVar,hasData,hasCoord,nCoordVar,*dims;
```

```
  cxCoord *coord;
  cxPrimType primType;
  cxCoordType coordtype;
  cxData *data;
  char *oldData, *dataArray;

  /* Get all the information about the input lattice */
  cxLatDescGet(inLat,&nDim,&dims,&hasData,&nDataVar,&primType,
        &hasCoord,&nCoordVar,&coordtype);
  cxLatPtrGet(inLat,&data, (void **)&oldData,&coord,NULL);

  /* Reset the range of the slider
   * based on the number of input data variables
   */
  xLo = 1;
  xHi = nDataVar;
  cxInWdgtLongMinMaxSet( "Which Data", xLo, xHi);

  /* Reset range of widget if off-scale */
  w = which;
  if(w > xHi)
    {
    w = xHi;
    cxInWdgtLongSet("Which Data",xHi);
  }
  if(w < xLo)
    {
    w = xLo;
    cxInWdgtLongSet("Which Data",xLo);
  }

  /* Make a new lattice of the correct size */
  w -= 1;
  nDataVar = 1;
  *outLat = cxLatDataNew(nDim,dims,nDataVar,primType);

  /* Move the coordinate pointer from the old to the new lattice */
  cxLatPtrSet(*outLat,NULL,NULL,coord,NULL);

  /* Get the information about the new data array */
  cxLatPtrGet(*outLat,NULL,(void **)&dataArray,NULL,NULL);
  nData = cxDimsProd(nDim,dims,nDataVar);
  dataLen = cxDataPrimSize(data);

  /* Copy the data from the old array to the new array */
  oldData += dataLen * w;
  for(i=0;i<nData;i++)
    {
      bcopy(oldData,dataArray,dataLen);
      oldData += dataLen * xHi;
      dataArray += dataLen;
    }
}
```

Fortran Version:

```
C     Example module that does a channel select
C
      SUBROUTINE CHAN(WHICH,INLAT,OUTLAT)
C
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
C
C     .. Scalar Arguments ..
      INTEGER            INLAT, OUTLAT, WHICH
C     .. Local Scalars ..
      INTEGER            CTYPE, DLEN, HASCRD, HASDAT, I, IER,
     *                   J, K, L, NCV, NDATA, NDV, NDIM, P0,
     *                   PTYPE, W, XHI
C     .. Local Arrays ..
      INTEGER            DIMS(1)
      CHARACTER          DARRAY(1), OLDDAT(1)
C     .. External Subroutines ..
      EXTERNAL           CXINWDGTLONGMINMAXSET, CXINWDGTLONGSET
C     .. External Functions ..
      EXTERNAL           CXDATAPRIMSIZE, CXDIMSPROD, CXLATDATANEW,
     *                   CXLATDESCGET, CXLATPTRGET
C     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
C     .. Pointers to Lattice Structures ..
      POINTER (PDIMS,DIMS)
      POINTER (PDATA,DATA)
      POINTER (PCOORD,COORD)
      POINTER (POLDAT,OLDDAT)
      POINTER (PARRAY,DARRAY)
C     .. Executable Statements ..
C
C     Get all the information about the input lattice
C
      IER = CXLATDESCGET(INLAT,NDIM,PDIMS,HASDAT,NDV,PTYPE,HASCRD,
     *      NCV,CTYPE)
      P0 = 0
      IER = CXLATPTRGET(INLAT,PDATA,POLDAT,PCOORD,P0)
C
C     Reset the range of the slider
C     based on the number of input data variables
C     Reset range of widget to be inside the scale
C
      W = MIN(MAX(1,WHICH),NDV)
      CALL CXINWDGTLONGMINMAXSET('Which Data',1,NDV)
      CALL CXINWDGTLONGSET('Which Data',W)
C
C     Make a new lattice of the correct size
C
      W = W - 1
      XHI = NDV
      NDV = 1
      OUTLAT = CXLATDATANEW(NDIM,DIMS,NDV,PTYPE)
C
```

```
C     Move the coordinate pointer from the old to the new lattice
C
      P0 = 0
      IER = CXLATPTRSET(OUTLAT,P0,%VAL(0),PCOORD,%VAL(0))
C
C     Get the information about the new data array
C
      P0 = 0
      IER = CXLATPTRGET(OUTLAT,P0,PARRAY,P0,P0)
      NDATA = CXDIMSPROD(NDIM,DIMS,NDV)
      DLEN = CXDATAPRIMSIZE(PDATA)
C
C     Copy the data from the old array to the new array
C
      DO 40 I = 1, NDATA
         K = DLEN*(I-1)
         L = DLEN*((I-1)*XHI+W)
         DO 20 J = 1, DLEN
            DARRAY(K+J) = OLDDAT(L+J)
 20      CONTINUE
 40      CONTINUE
C
      RETURN
      END
```

Chapter 3

Using the Lattice Data Type

This chapter defines the lattice data type and how it is used in the IRIS Explorer environment. It describes the forms that lattices can take in one, two, and three dimensions and gives examples of how to create them.

The API (Application Programming Interface) routines are listed and examples of user function code for writing modules that manipulate lattices are included.

3.1. Understanding Lattices

The IRIS Explorer lattice data type, *cxLattice*, contains all the information IRIS Explorer requires for creating arrays. An array is a regular, structured matrix of points, which can be 1−D or multidimensional. Because it is extremely versatile, you can use the *cxLattice* data structure to represent a wide variety of array data. Thus, some modules may accept a *cxLattice* with any number of dimensions, containing any number of data values, of any type. Other modules may, for example, accept only 1−D lattices in byte format. In general, the form of the array data you plan to feed into a module will determine how narrowly you define the lattice specifications on the input port in the Module Builder.

The IRIS Explorer lattice data structure has two parts. One holds *data values* and the other holds *node coordinates.* A *node* is a point in a lattice defined by a unique coordinate or set of coordinates in Cartesian space, usually indicating the position of the data value (or values). The data and coordinate arrays are optional, however. You can create a lattice with an empty data structure, and node coordinates only, or one with data values and no coordinate values.

The two array structures in *cxLattice* are *cxData* and *cxCoord.* The data and coordinate arrays are defined in separate variables because:

- The Cartesian space in which the lattice exists may be irregular, for example, in a finite difference−based simulation program. Such programs may require complicated physical mappings even though the data is straightforward.
- Two different lattices, such as an input and an output lattice, can share the same data or coordinate array, thereby saving memory. For example, the lattices may have different data but use the same coordinate values.

Figure 3−1 illustrates the principle of sharing array values. The lattices on the module input port and on the module output port use the same coordinate values, which are passed directly from input to output, but the data values change from input to output as they are processed by the user function.

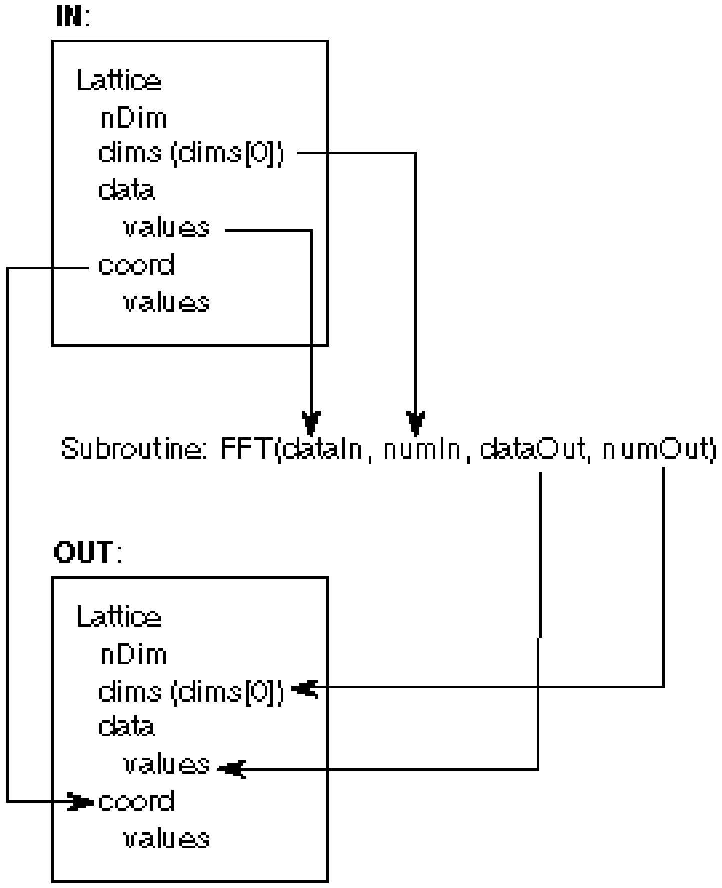


Figure 3–1 Data Flow Between Input and Output Lattices

3.1.1. The Lattice Data Type

The *cxLattice* data type is one of the root data types, which means it can be placed on module ports and will pass data into and out of modules. It has three main parts, or subsidiary data structures:

- the dimension variables, *nDim* and *dims*
- the subtype, *cxData*, which defines the number of data variables, the primitive type, and the data values
- the subtype, *cxCoord*, which defines the number of coordinates, their Cartesian mapping, and their values

This is the type definition for *cxLattice*:

```
shared root typedef struct {
  long                *nDim         "Num Dimensions";
  long*               *dims[nDim]   "Dimensions Array";
  cxData(nDim, dims)  *data         "Data Structure";
  cxCoord(cDim, dims) *coord        "Coord Structure";
} cxLattice;
```

The type definitions for *cxData* and *cxCoord* are given below. **Figure 3–2** shows a schematic representation of *cxLattice.*

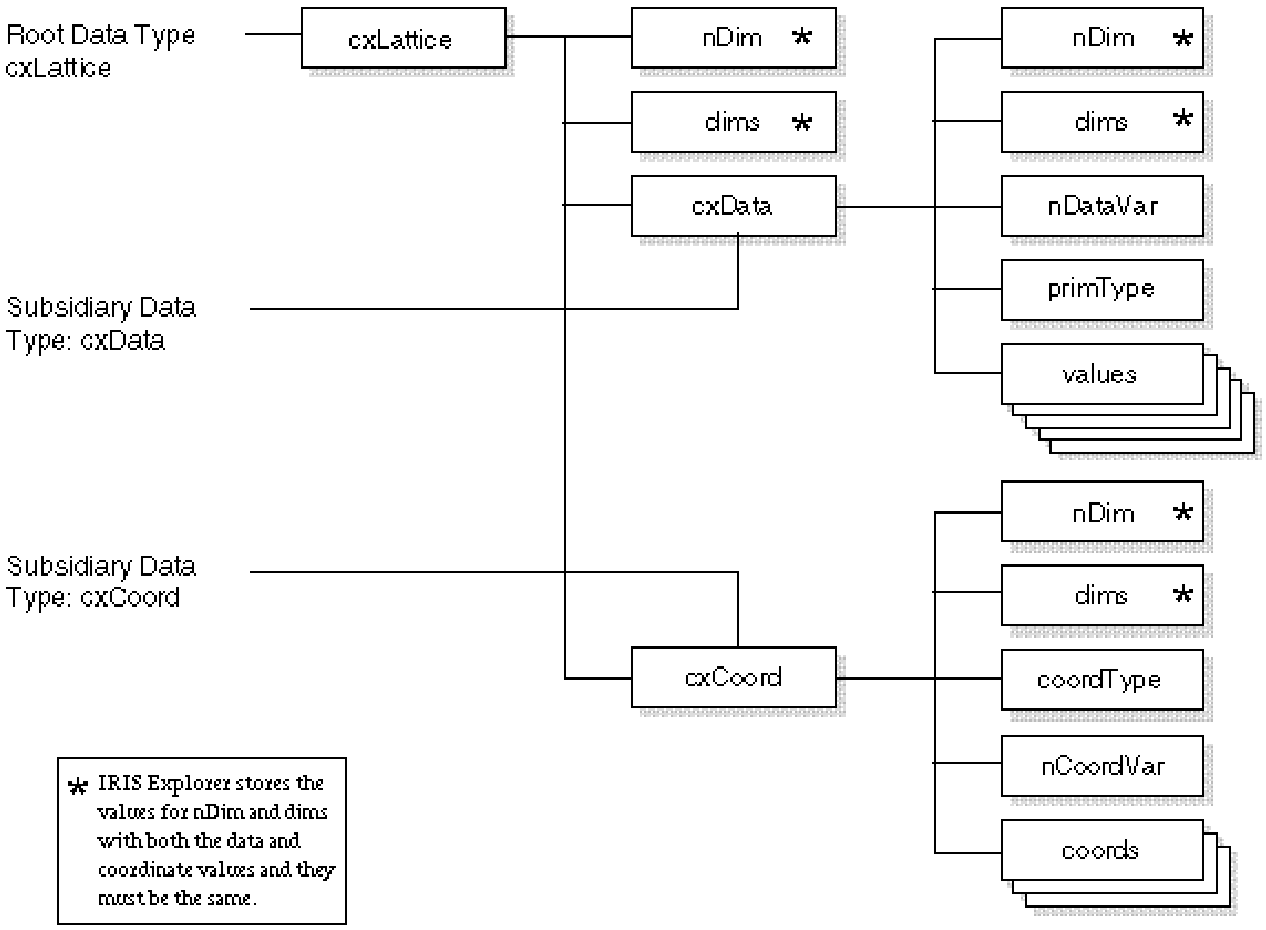


Figure 3–2 Schematic Structure of the Lattice Data Type

The Dimension Variables

The dimension variables in the *cxLattice* data type are:

nDim — The number of dimensions. It determines the dimensionality of the array that holds data, independent of its Cartesian mapping and the number of data variables. For curvilinear lattices it also determines the dimensionality of the array which holds the node coordinates.

dims — A vector of integer values specifying the number of nodes in each of the *nDim* dimensions.

The *nDim* variable indicates the number of dimensions of the lattice. The *dims* array specifies the number of nodes in each dimension – that is, the number of data values in each dimension.

From **Figure 3–2**, it can be seen that *nDim* and *dims* are stored with both *cxData* and *cxCoord*; the dimensions of these arrays are set by the values of *nDim* and *dims*. Thus, these values *must* be consistent throughout the *cxLattice* structure; otherwise, you may get some bizarre results. For example, if you define *nDim* as 3 and *dims* as [10,6,6] in *cxData*, you must also define them as 3 and [10,6,6] in *cxCoord*.

The *cxData* Structure

Data values go into the *cxData* type, which contains the value or values stored at each node of the lattice. Its elements include:

- the dimension variables, *nDim* and *dims*
- the number of data variables per node, *nDataVar*
- the type of primitive variable that can be used for data values, *primType*. There are five options (see **Table 3–1**).
- the array of data values

This is the data type definition for *cxData*:

```
shared typedef struct { /* IRIS Explorer Lattice's Data array */
  long       nDim;
  long       dims[nDim];
  long       nDataVar     "Num Data Variables";
  cxPrimType primType     "Primitive Data Type";
  switch     (primType) {
    case cx_prim_byte:
      char      values[nDataVar, dims] "Data Array";
    case cx_prim_short:
      short     values[nDataVar, dims] "Data Array";
    case cx_prim_long:
      long      values[nDataVar, dims] "Data Array";
    case cx_prim_float:
      float     values[nDataVar, dims] "Data Array";
    case cx_prim_double:
      double    values[nDataVar, dims] "Data Array";
  } d;
} cxData(nDim, dims);

typedef enum {
  cx_prim_byte,
  cx_prim_short,
  cx_prim_long,
  cx_prim_float,
  cx_prim_double
} cxPrimType;
```

The *cxCoord* Structure

The Cartesian coordinate values that define the position of the lattice nodes go into the *cxCoord* data type. These values map the lattice data to Cartesian space. Its elements include:

- the dimension variables, *nDim* and *dims*
- the type of physical mapping, *coordType*. There are three options: uniform, perimeter, and curvilinear mapping, described later in this chapter.
- The number of coordinates per node, *nCoordVar*
- the array of coordinate values

This is the data type definition for *cxCoord*:

```
shared typedef struct { /* IRIS Explorer Lattice's Coordinate array */
  long          nDim;
  long          dims[nDim];
  cxCoordType   coordType                  "Coord Type";
  switch        (coordType) {
    case          cx_coord_uniform:
      float         bBox[2, nDim]            "Coordinates Array";
    case          cx_coord_perimeter:
      long          sumCoord                 "Perim Coord Array Length";
      float         perimCoord[sumCoord]     "Coordinates Array";
```

```
    case         cx_coord_curvilinear:
      long         nCoordVar               "Num Coord Dimensions";
      float        values[nCoordVar, dims] "Coordinates Array";
  } c;
} cxCoord(nDim, dims);
```

How Variables Interact

Figure 3−3 shows the relationship of some lattice variables. This example depicts a 2−D lattice with seven nodes in each dimension and three data variables per node. Thus, *nDim* = 2, *dims*[0] = 7 (*x* direction), *dims*[1] = 7 (*y* direction), and *nDataVar* = 3.

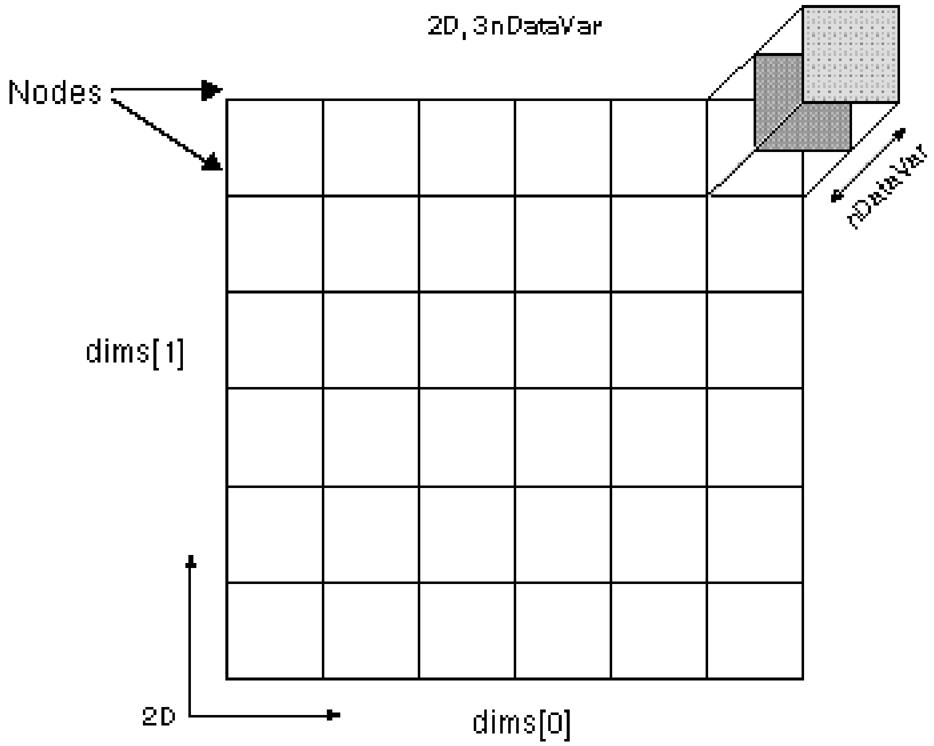


Figure 3−3 Lattice Variables

3.2. Manipulating Lattices

The lattice and coordinate data values are stored separately and in different formats. **Figure 3−4** shows how data from arrays in C and Fortran formats is stored in computer memory, and illustrates the difference between row−major (C) and column−major format (Fortran). The formats used by each data and coordinate lattice type in *cxLattice* format are described below.

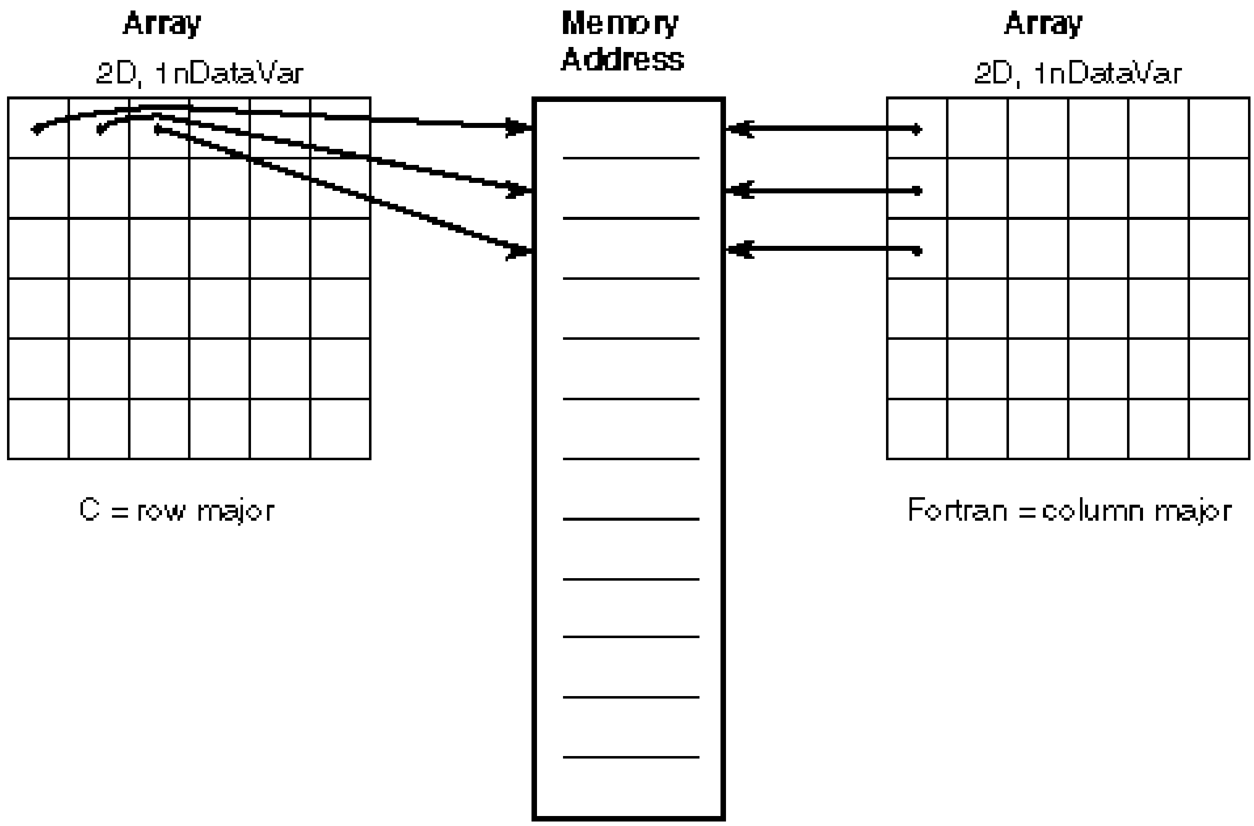


Figure 3−4 Storing Arrays in C and Fortran

3.2.1. Storing Data Values

Lattice data is located at the coordinate nodes. In a 1−D array, or vector, each node in the array has two neighbors (except the end points, which each have only one), as shown in **Figure 3−5**. In two dimensions, each internal node has four neighbors. A node internal to a 3−D array has six neighbors. An internal node in an *n*−D array has 2^n neighbors. This regular structure is the *computational* space of the lattice. This topology is completely determined by the value of *nDim* and the *dims* vector, and is to be distinguished from the *physical* space of the lattice, which is determined by its coordinates part.

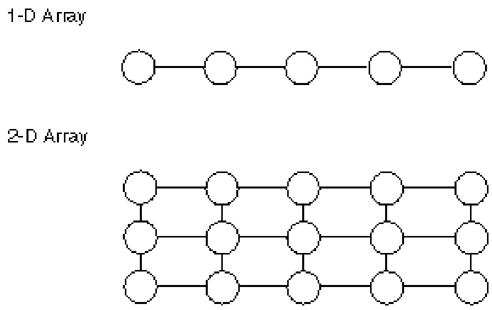


Figure 3–5 Array Neighbors

Lattice data is stored in the Fortran convention, using a column–major layout, in which the I direction of the array varies the fastest (see **Figure 3–6**). For all lattice types, the I direction corresponds to the X direction. Similarly, J corresponds to Y, and K to Z.

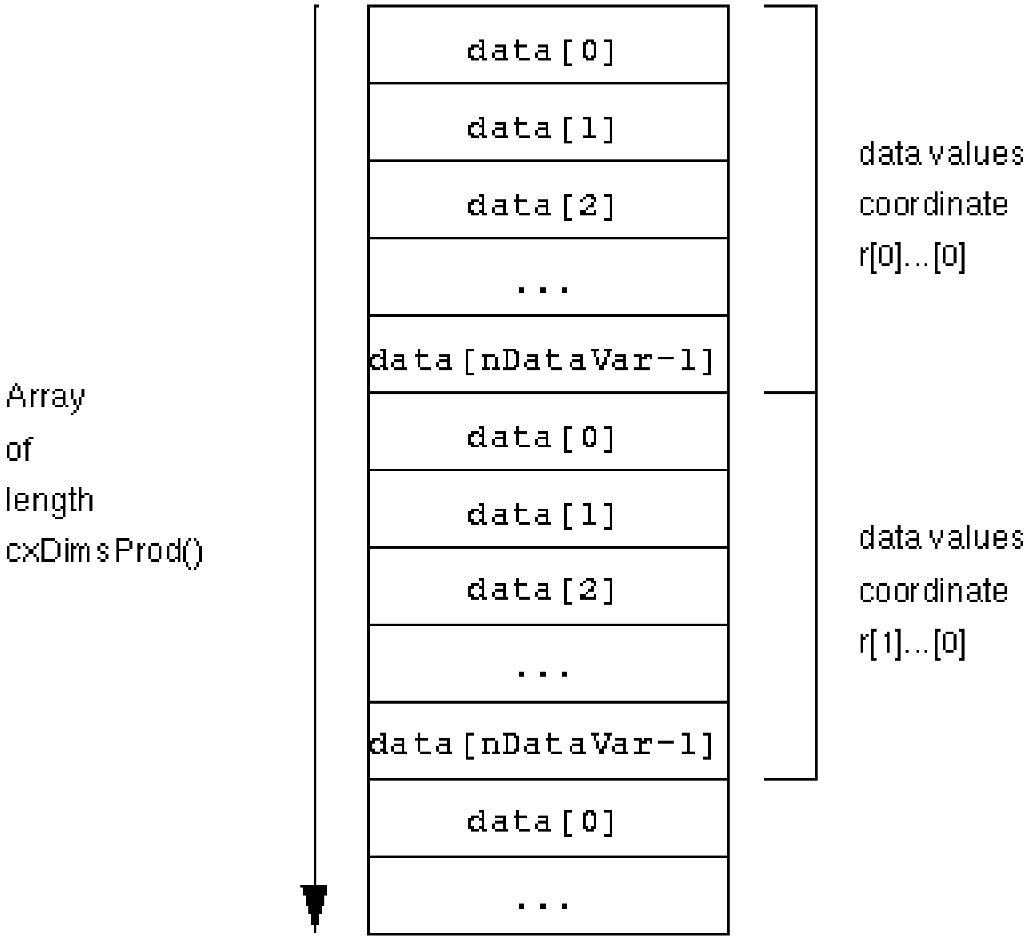


Figure 3–6 Storing Data Values

If a lattice has several data values at each node (that is, if *nDataVar > 1*), then the data is stored in interleaved format. In a color image, for example, the interleaving of RGB data looks like this:

```
R(node 1)
G(node 1)
B(node 1)
R(node 2)
G(node 2)
B(node 2)
R(node 3)
G(node 3)
B(node 3)...
```

☆To locate a particular node within an array, you use array indexing. For example, the node located at (i,j,k) is:

```
Array[k][j][i]       in C
Array(i,j,k)         in Fortran
```

You can compute the total number of data values in a lattice by calling the API function *cxDimsProd*. Since this number is the product of the number of nodes in the lattice and *nDataVar*, the number of data values at each node, you can use this result to calculate the number of nodes in the lattice. See the *IRIS Explorer Reference Pages* or the on–line *man* pages for details on the API routines.

3.2.2. Defining Primitive Values

The primitive data type is defined in terms of C types in the lattice data type. If you are programming in Fortran, choose the C primitive that is equivalent to the Fortran variable that your subroutine expects. **Table 3–1** lists the equivalences between the two.

C Data Types	Fortran Equivalents
byte (signed char)	character*1
short	integer*2
long	integer (integer*4)
float	real (real*4)
double	double precision (real*8)

Table 3–1 Lattice Primitive Types

3.2.3. Storing Coordinate Values

Coordinates are always stored in single–precision floating point (float) format. The lattice data type allows for three types of coordinate mapping to physical space: uniform, perimeter, and curvilinear. The interleaving of the coordinate storage varies from type to type. Each type is described in detail below.

Uniform Lattices

A lattice with uniform coordinates has a uniform cell size throughout (see **Figure 3–7**). Most generated data is in this format.

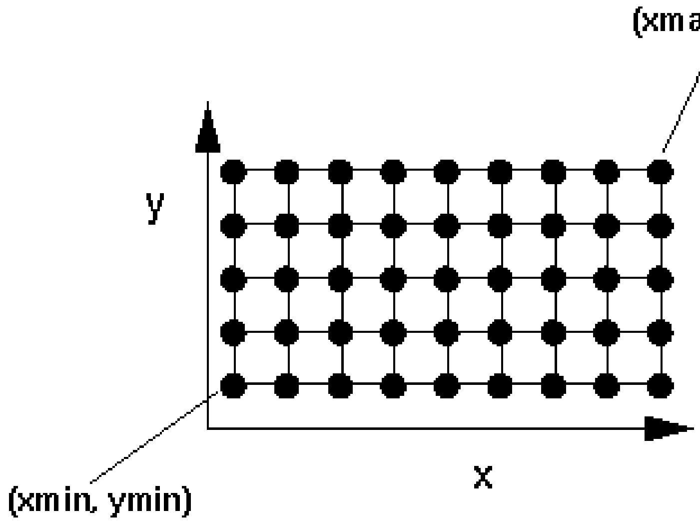


Figure 3–7 A 2–D Uniform Lattice Structure

The data structure for a uniform lattice is:

```
struct {
  float    *bBox; /* An array of length [2, ndim]*/
} cx_coord_uniform;
```

The coordinate values are stored in row–major format, in the C convention:

min[0]	max[0]	min[1]	max[1]	min[2]	max[2]

IRIS Explorer uses a bounding box to set the size and aspect ratio of uniform lattice coordinates. Bounding boxes are dimensioned as a constant and a scalar in the *cxCoord* data type. That is, *dims* is [2, *nDim*]. IRIS Explorer saves only the bounding box coordinates for a uniform lattice. It can construct the complete lattice coordinate set from these values.

For example, this is how *PrintLat* prints out the coordinate structure for the uniform lattice shown in **Figure 3–7**. It shows the values of *nDim* (in this case 2), *dims*, *coordType*, and the bounding box coordinates. Comparing this output with the variables of Figure 3–7, it can be seen that *xmin* = 0.0, *ymin* = 0.0, *xmax* = *dims[0]*–1.0 and *ymax* = *dims[1]* –1.0. In this example, the spacing between nodes in the *x* and *y* directions are both equal to 1.0. This can be altered by changing the coordinates of the bounding box (see **"Changing the Aspect Ratio"**).

```
coord
  nDim 2
  dims 9 5  /* Number of nodes in the x and y dimensions /*
  coordType cx_coord_uniform

  bBox
     0.000000e+00       8.0000000e+00  /* The values of xmin and xmax /*
     0.000000e+00       4.0000000e+00  /* The values of ymin and ymax /*
```

Figure 3–8 shows an example of a 3–D uniform lattice.

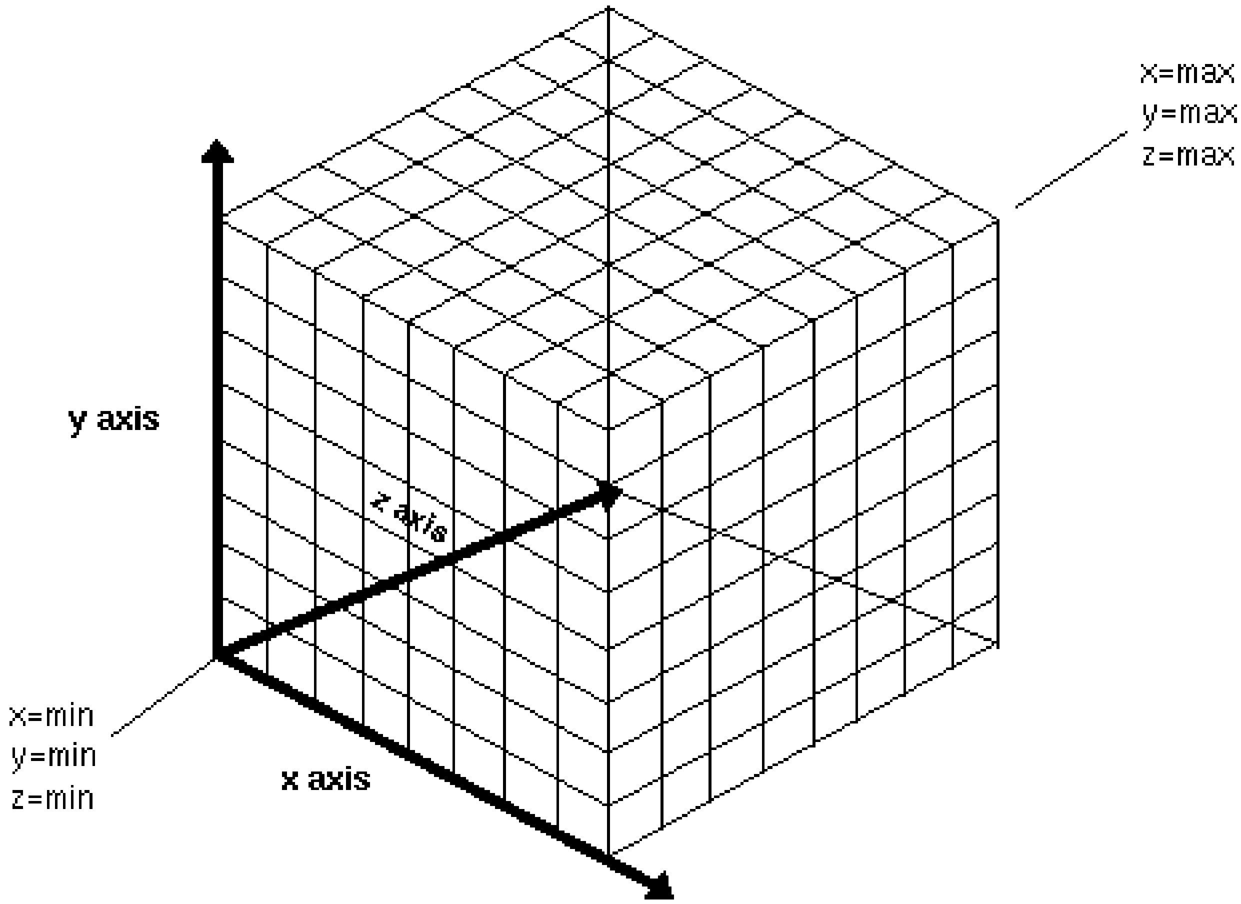


Figure 3−8 A 3−D Uniform Lattice Structure

Changing the Aspect Ratio

A 2−D image is an example of a uniform lattice; all the pixels in the image have the same size and aspect ratio. You can change the aspect ratio of a lattice by manipulating the bounding box coordinates.

For example, the lattice in **Figure 3−7** has nine nodes in the *x* direction and five nodes in the *y* direction. The default mapping provides a 1:1 aspect ratio. Since the bounding box for this lattice is [0.0, 8.0] by [0.0, 4.0], the lattice is mapped into a 9 by 5 grid to be displayed. However, if the bounding box coordinates were [−1.0, 1.0] by [−1.0, 1.0], the lattice would occupy a 2 by 2 space when mapped to the screen, with a pixel aspect ratio of 2:1. Uniform lattices can have this non−uniform aspect.

Perimeter Lattices

A perimeter lattice has a list of coordinate values sufficient to specify an irregularly spaced rectangular structure.

The data structure for a perimeter lattice is:

```
struct {
  long    sumCoord;  /* Total number of nodes in all dims */
  float   *perimCoord; /* Ordered list of all coordinates */
} cx_coord_perimeter;
```

Here, **perimCoord* is an array of length *sumCoord* containing an ordered list of all the coordinates for each dimension. You can compute *sumCoord* by calling the API subroutine *cxDimsSum*. See the *IRIS Explorer Reference Pages* or the on−line *man* pages for details on the API subroutines.

Figure 3−9 shows the data set for a 2−D perimeter lattice. The *x* and *y* perimeter vectors contain coordinate values that specify the layout of the lattice. It contains eight nodes in the *x* dimension and six nodes in the *y* dimension.

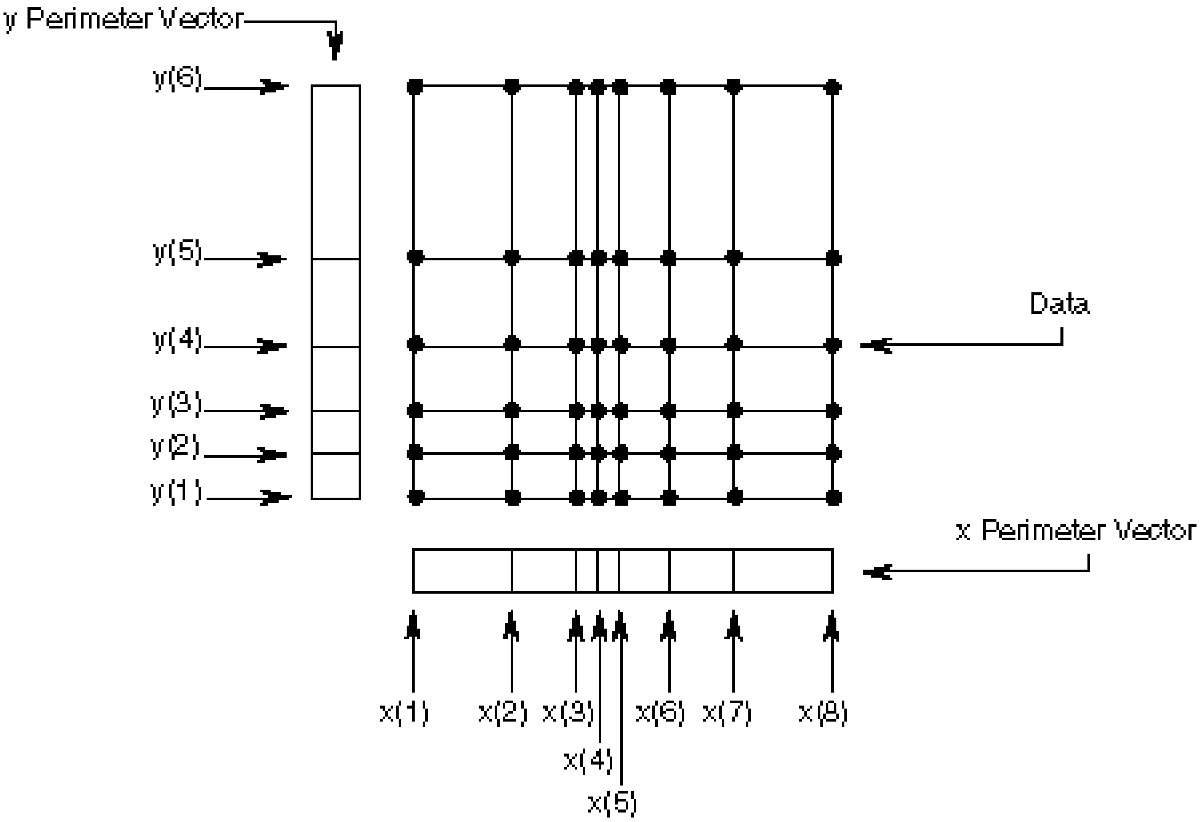


Figure 3−9 Data Set for a 2−D Perimeter Lattice

Coordinates for perimeter lattices are stored in row−major format in the C convention (see **Figure 3−10**).

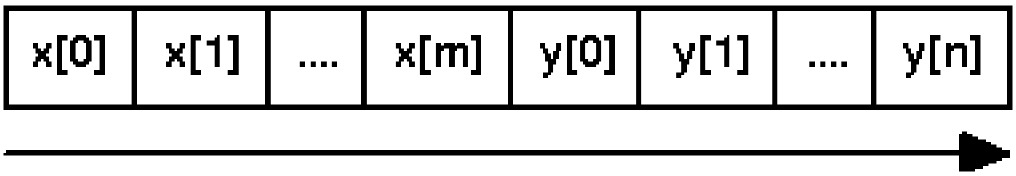


Figure 3−10 Storing Perimeter Coordinates

Figure 3−11 shows an example of a 3−D perimeter lattice. The *dims* values are the same for each of the perimeter vectors, because there are the same number of nodes in each dimension.

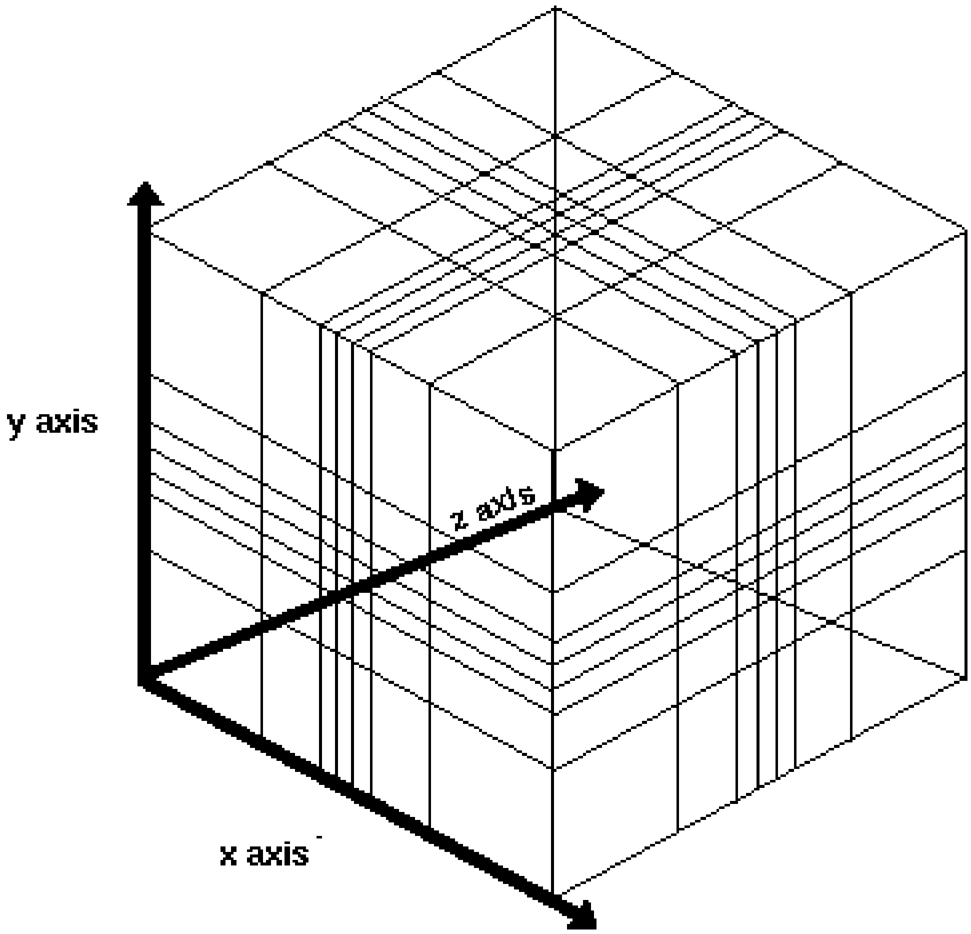


Figure 3−11 Example of a 3−D Perimeter Lattice

Curvilinear Lattices

Curvilinear lattices are used to store datasets where the data values at a node need to be associated explicitly with the Cartesian coordinates of the node. Examples of these include a collection of atoms in 3D space, points on the surface of a sphere and computational fluid dynamics data calculated in a body−fitted coordinate system.

The data structure for the coordinates part of a curvilinear lattice is:

```
struct {
  long       nCoordVar;    /* Number of physical dimensions */
  float      *values;  /* Array containing node coordinates */
} cx_coord_curvilinear;
```

You can use *cxDimsProd* to compute the total number of coordinate values as the product of the number of nodes in the lattice (i.e. the *dims* vector) and *nCoordVar.*

Coordinate values for curvilinear lattices are stored interlaced at the node level in the Fortran convention, with the I dimension varying the fastest. This is the same storage method used to store lattice data (see **Figure 3−6**) and is the reverse of the method used for storing uniform and perimeter coordinates.

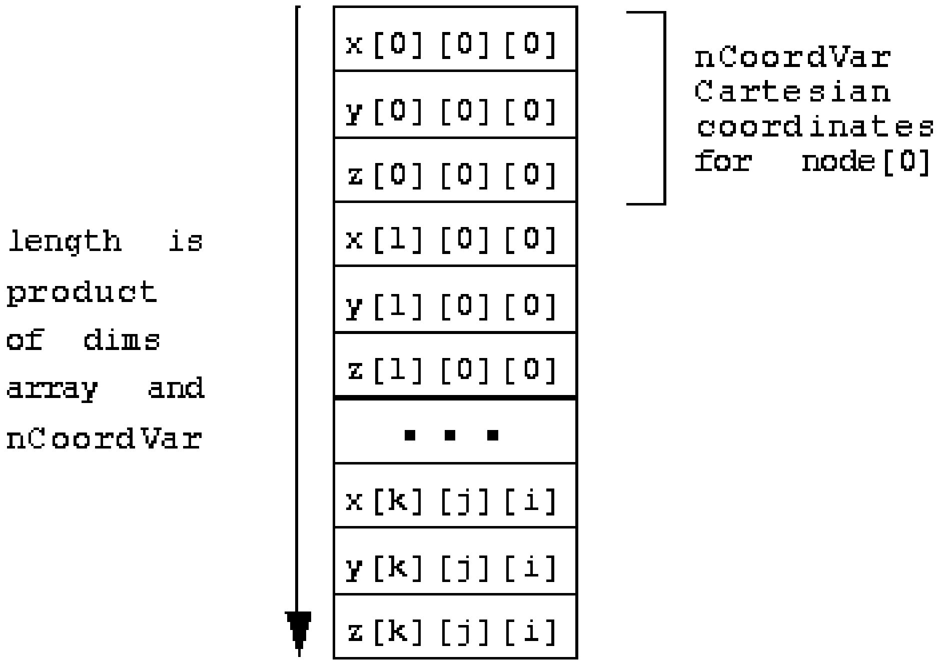
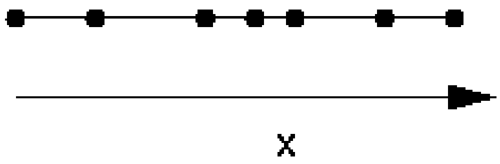
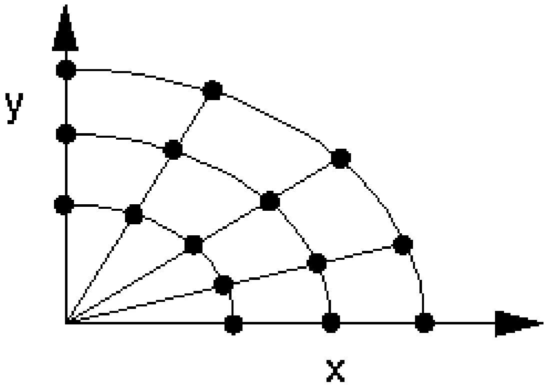


Figure 3–12 Storing Curvilinear Coordinates

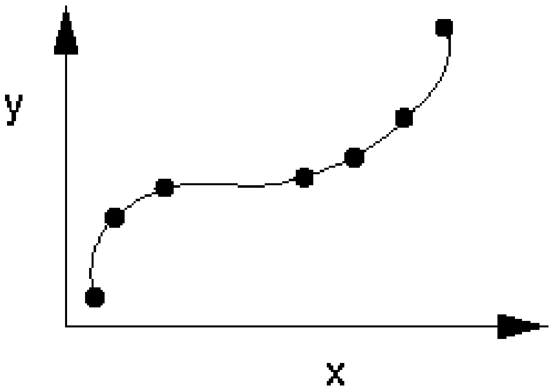
The number of computational dimensions for the lattice is defined by the variable *nDim* (see **Figure 3–5**). For uniform and perimeter lattices, this number is also equal to the number of physical dimensions for the lattice, but for curvilinear lattices, the number of physical dimensions is defined by the variable *nCoordVar*, the number of coordinate variables for each node. In principle, this can have any value, although since each node in an *nDim*–lattice usually requires at least this number of coordinates to locate it in physical space, useful lattices will have *nDim* <= *nCoordVar*. Some examples of curvilinear lattices with this property are shown in **Figure 3–13**.



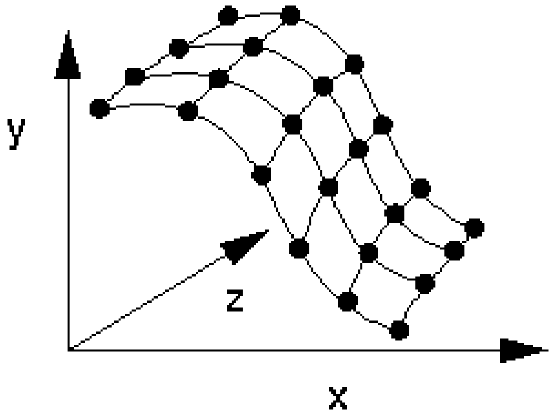
nDim = 1, nCoordVar = 1



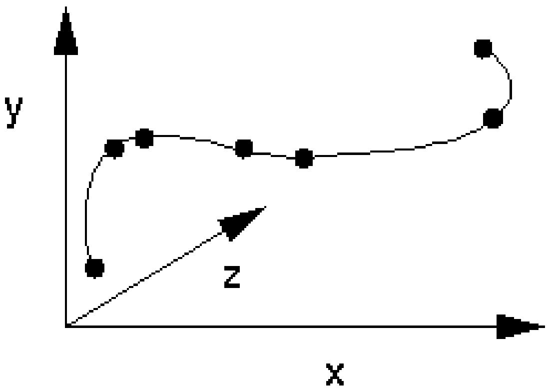
nDim = 2, nCoordVar = 2



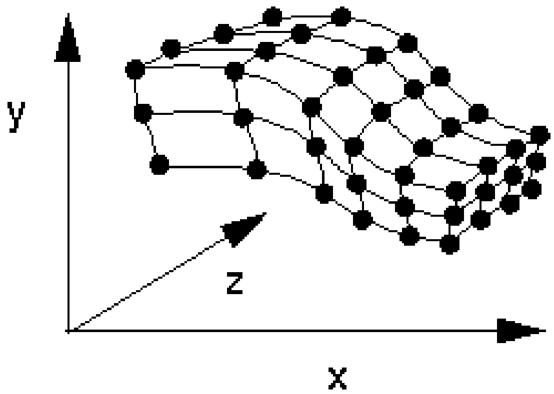
nDim = 1, nCoordVar = 2



nDim = 2, nCoordVar = 3



nDim = 1, nCoordVar = 3



nDim = 3, nCoordVar = 3

Figure 3–13 Curvilinear Lattices with various values for *nDim* and *nCoordVar*

This figure shows the wide variety of datasets that can be stored in a curvilinear lattice; from collections of points (*nDim* = 1) in 1, 2 or 3–D space (*nCoordVar* = 1, 2 or 3) through areas (*nDim* = 2) in 2 or 3–D space (*nCoordVar* = 2 or 3) to volumes (*nDim* = 3) in 3–D space (*nCoordVar* = 3). In addition it should be noted that, because of its greater generality, a curvilinear lattice can always store datasets that are stored in a perimeter or uniform lattice (and a perimeter lattice can always store datasets from a uniform lattice, for the same reason). However, this would be an inefficient use of storage space, since much of the coordinate information would be redundant under these circumstances, and it is always best to use the simplest type of lattice to store a given set of data.

3.3. Limiting Lattice Values

When you build a module, you specify the range of lattice types the module can accept on its input port or produce on its output port. You can define in general terms the lattice constraints that encompass a large range of values for a given element, or you can be very specific. The range you choose will depend on the kind of data you want the module to handle.

The Lattice Constraints window in **Figure 3–14** shows the settings for a more narrowly defined lattice, such as a colormap (see **"Lattice Examples"**). The port will accept a 1–D lattice with four data variables. The primitive data type must be a float, and the lattice coordinate type must be uniform. The number of coordinate dimensions is not limited.

See **"Setting Lattice Constraints"** for more information on using this window.

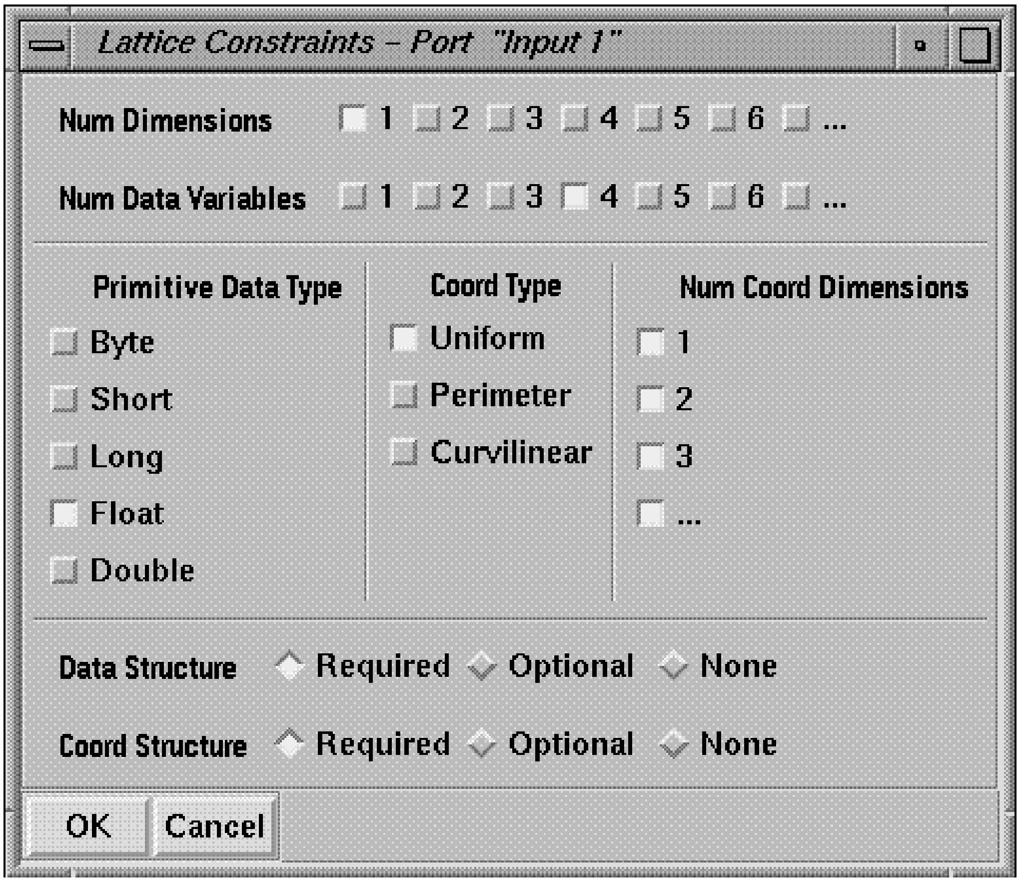


Figure 3−14 Defining Lattice Constraints

3.4. Lattice Examples

These examples show how to create some simple, commonly used lattices. They include code for a colormap and a 2−D image.

3.4.1. A Colormap Example

A colormap is a 1−D lattice with four variables per node (RGBA). Nodes are spaced uniformly. The data is usually in floating point format. The elements are:

nDim	usually 1, although colormaps can be a function of more than one variable

Footnote: If you use a 2−D colormap, you must create your own modules to process it. IRIS Explorer currently offers tools for 1−D colormaps only.

dims	256
nDataVar	4 (red, green, blue, alpha)
primType	float
coordType	uniform

See **"Code Examples"** for an example of a user function for a module that accepts a colormap.

3.4.2. A 2−D Image Example

An image is a 2−D lattice with one variable (greyscale), three variables (RGB), or four variables (RGBA) per node. The data is usually in byte format and the coordinate spacing between nodes is usually uniform. IRIS Explorer image−processing modules accept images of any size. The elements are:

nDim	2
dims	any
nDataVar	4 (red, green, blue, opacity)
primType	byte
coordType	usually uniform, although certain Landsat images and "fish−eye" photographic images may be distorted and therefore use curvilinear coordinates

See **"Code Examples"** for examples of user functions for modules that work with both 2−D and 3−D lattices.

3.4.3. Preparing Your Data for Lattices

To prepare your data for input into an IRIS Explorer lattice data type, follow these points:

1. All information in the data portion of a lattice must be of only one of the primitive types. You cannot have a lattice that mixes primitive types. For example, a data file containing a mixture of bytes and floats must either be read into two lattices (one containing the bytes, and the other containing the floats) or the data must be converted so that it is all of one type.
2. If you are coding in Fortran, remember to use zero−based indices instead of one−based indices.
3. If you have Fortran scalars, convert your primitive type into the equivalent IRIS Explorer primitive type (see **"Defining Primitive Values"**).
4. The coordinate data for uniform and perimeter lattices is stored differently from curvilinear lattices. Check in the appropriate part of **"Storing Coordinate Values"** to make sure you have your coordinates correctly arranged.

For information on converting foreign data arrays into IRIS Explorer lattices, see Chapter 7, "Using the DataScribe", in the *IRIS Explorer User's Guide.*

3.5. The Data Type Declaration

The *cxLattice* data type, though defined in the IRIS Explorer typing language, can be considered as a C structure. Fortran users need to set pointers to the data type structures when they use them. The type declaration resides in the header file */usr/explorer/include/cx/cxLattice.h*.

This is the *cxLattice* data type declaration:

```
#include <cx/Typedefs.h>

typedef enum {
    cx_coord_uniform,
    cx_coord_perimeter,
    cx_coord_curvilinear
} cxCoordType;

typedef struct cxCoord {
    long         nDim;
    long        *dims;
    cxCoordType  coordType;
    union {
       struct {
          float   *bBox;
        } cx_coord_uniform;
        struct {
           long    sumCoord;
           float  *perimCoord;
        } cx_coord_perimeter;
        struct {
           long    nCoordVar;
           float  *values;
        } cx_coord_curvilinear;
    } c;
} cxCoord;

typedef struct cxData {
    long         nDim;
    long        *dims;
    long         nDataVar;
    cxPrimType   primType;
    union {
        struct {
            unsigned char *values;
        } cx_prim_byte;
        struct {
            short         *values;
        } cx_prim_short;
        struct {
            long          *values;
        } cx_prim_long;
        struct {
```

```
            float          *values;
        } cx_prim_float;
        struct {
            double         *values;
        } cx_prim_double;
    } d;
} cxData;

typedef struct cxLattice {
    long          nDim;
    long         *dims;
    cxData       *data;
    cxCoord      *coord;
} cxLattice;
```

The Fortran type enumerations reside in the file */usr/explorer/include/cx/cxLattice.inc*.

The three coordinate mappings are specified as follows:

```
integer cx_coord_uniform
integer cx_coord_perimeter
integer cx_coord_curvilinear

parameter (cx_coord_uniform = 0 )
parameter (cx_coord_perimeter = 1 )
parameter (cx_coord_curvilinear = 2 )
```

Note: All Fortran data type access routines use zero−based indexing, as in the C language (except where otherwise stated).

3.6. The Lattice API Routines

You can use the API (Application Programming Interface) routines to manipulate data types in IRIS Explorer. The lattice subroutines are listed below and described in detail in the *IRIS Explorer Reference Pages* and the on−line *man* pages. They let you:

- create a lattice with both data and coordinates, either one, or neither
- calculate the size of the data set
- set lattice values
- copy a source lattice (or parts thereof) and allocate it to the destination
- extract specific values or descriptive information from the lattice

Some routines check the validity of the data on the inputs before the module is fired. **Table 3−2** lists the subroutines and briefly describes the purpose of each one.

Subroutine	Purpose
cxLatNew	Creates a lattice with data and coordinates
cxLatDataNew	Creates a lattice with data and no coordinates
cxLatCoordNew	Creates a lattice with coordinates and no data
cxLatRootNew	Creates a lattice with no data and no coordinates
cxDataNew	Creates new data structure
cxCoordNew	Creates new coordinate data structure
cxCoordDefaultNew	Creates default index coordinates
cxDataPrimSize	Returns the size of the primitive data type

cxDataPrimType	Returns the primitive data type
cxCoordType	Returns the lattice coordinate type
cxDimsProd	Returns the total number of data points in a lattice
cxDimsSum	Computes the sum of a dimensions vector (for perimeter lattices only)
cxLatPtrSet	Sets lattice data and coordinate pointers
cxLatPtrGet	Gets pointers to lattice data and coordinates
cxLatDup	Duplicates a lattice with data or coordinates
cxLatRootDup	Duplicates a lattice without data or coordinates
cxDataDup	Duplicates data values
cxCoordDup	Duplicates coordinates
cxLatDescGet	Gets descriptive information about a lattice
cxDataValsGet	Returns pointer to data values
cxDataValsSet	Sets pointer to data values
cxCoordValsGet	Returns pointer to coordinate value
cxCoordValsSet	Sets pointer to coordinate values
cxCoordNVarGet	Gets number of coordinate variables
cxCoordNVarSet	Sets number of coordinate values

Table 3−2 Lattice Subroutines

3.7. Code Examples

This section presents three examples of source code using the lattice data structure. Each one is written in C and in Fortran. The source code files and modules for all the examples reside in */usr/explorer/src/MWGcode*

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

.

3.7.1. 1−D Lattice

This example shows how to create a a simple colormap module. It inverts the colors of the input colormap. You may test it by connecting these modules: *GenerateColorMap* to this module and *PrintLat*; this module to another copy of *PrintLat*. Compare the output from the two *PrintLat* modules. The code resides in */usr/explorer/src/MWGcode/Lattice/C/ColorMap.c* and in */usr/explorer/src/MWGcode/Lattice/Fortran/ColorMap.f.*

C Version:

```
#include <stdio.h>

#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>

/*
 This user function takes a colormap input (1D lattice with 4 floats
 per sample) and inverts the red, green, and blue components of each
 lookup value.
*/
void cchange(
cxLattice *cin,  /* input colormap lattice */
```

```
cxLattice **cout) /* output colormap lattice */
{
 int i;  /* loop variable */
 int n;  /* number of colormap entries */
 float *a,*b;  /* pointers to colormap data */
 cxErrorCode err;

 /* create the new lattice */
 *cout = cxLatDataNew(
  1,  /* number of dimensions */
  cxLatticeDimensionsArrayGet(cin,&err),
  4,  /* number of data values */
  cx_prim_float); /* data type */

 /* use the same coordinate space as the input lattice */
 cxLatPtrSet(*cout,NULL,NULL,
  cxLatticeCoordStructureGet(cin,&err),NULL);

 /* get the number of samples */
 n = (int)*cxLatticeDimensionsArrayGet(cin,&err);

 /* get the data pointers */
 cxLatPtrGet(cin,NULL,(void **)&a,NULL,NULL);
 cxLatPtrGet(*cout,NULL,(void **)&b,NULL,NULL);

 /* invert the colors of the colormap entries */
 for ( i = 0; i < n; i++ )
 {
  b[0] = 1.0 - a[0]; /* red */
  b[1] = 1.0 - a[1]; /* green */
  b[2] = 1.0 - a[2]; /* blue */
  b[3] = a[3];  /* alpha */

  a += 4; b += 4;
 }
}
```

Fortran Version:

```
C      This user function takes a colormap input
C      (1D lattice with 4 real values per sample)
C      The red, green, and blue of each lookup value are inverted
C
       SUBROUTINE CHANGE(CIN,COUT)
C
       INCLUDE '/usr/explorer/include/cx/Typedefs.inc'
       INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
C      .. Scalar Arguments ..
       INTEGER            CIN, COUT
C      .. Local Scalars ..
       INTEGER            COORD, I, IER, NN, PA, PB, PN, P0
C      .. Local Arrays ..
       REAL               A(1), B(1)
       INTEGER            N(1)
```

```
C     .. External Functions ..
      EXTERNAL          CXLATDATANEW, CXLATDESCGET, CXLATPTRGET,
     *                  CXLATPTRSET
C     .. Pointers to Lattice Structures ..
      POINTER (PA,A)
      POINTER (PB,B)
      POINTER (PN,N)
C     .. Executable Statements ..
C
C     Get the number of samples
C
      P0 = 0
      IER = CXLATDESCGET(CIN,P0,PN,P0,P0,P0,P0,P0,P0)
      NN = N(1)
C
C     Create the new lattice
C
      COUT = CXLATDATANEW(1,N,4,CX_PRIM_FLOAT)
C
C     Use the same coordinate space as the input lattice
C
      P0 = 0
      IER = CXLATPTRGET(CIN,P0,P0,COORD,P0)
      P0 = 0
      IER = CXLATPTRSET(COUT,P0,%VAL(0),COORD,%VAL(0))
C
C     Get the data pointers
C     Note that we could have had the data pointers passed into
C     the user function as arguments, which may be easier
C
      P0 = 0
      IER = CXLATPTRGET(CIN,P0,PA,P0,P0)
      P0 = 0
      IER = CXLATPTRGET(COUT,P0,PB,P0,P0)
C
C     Invert the red, green, and blue lookup values
C
      DO 20 I = 1, NN
         B(4*I-3) = 1.0 - A(4*I-3)
         B(4*I-2) = 1.0 - A(4*I-2)
         B(4*I-1) = 1.0 - A(4*I-1)
         B(4*I) = A(4*I)
   20 CONTINUE
C
      RETURN
      END
```

3.7.2. 2−D Lattice

This example shows how to work with a 2−D image, and negates all elements of the data in the input image. Test it by connecting *ReadImg* to this module and *DisplayImg*; this module to *DisplayImg* (*DisplayImg* accepts more than one image, and you can display them in the same window; see the *DisplayImg(1EXP)* man page for more details). Read in the image file */usr/explorer/data/image/flowers2.rgb* and compare the two images. The code resides in */usr/explorer/src/MWGcode/Lattice/C/Image2D.c* and in */usr/explorer/src/MWGcode/Lattice/Fortran/Image2D.f*.

C Version:

```
#include <stdio.h>

#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>

/*
 This user function takes an input 2D image and negates all the
 elements of the data array.  It works for any primitive data type.
*/
void cimage(cxLattice *in,cxLattice **out)
{
 int i,j,k;  /* loop variables */
 void *a,*b;  /* data pointers */
 cxErrorCode err;

 /* create the new lattice */
 *out = cxLatDataNew(
  2,  /* number of dimensions */
  in->dims,  /* dimensions array */
  in->data->nDataVar, /* number of data variables */
  in->data->primType); /* data type */

 /* use the same coordinate space as the input lattice */
 cxLatPtrSet(*out,NULL,NULL,cxLatticeCoordStructureGet(in,&err),NULL);

 /* extract the data pointers */
 cxLatPtrGet(in,NULL,&a,NULL,NULL);
 cxLatPtrGet(*out,NULL,&b,NULL,NULL);

 /* loop over the data elements */
 for ( i = 0; i < in->dims[1]; i++ )
  for ( j = 0; j < in->dims[0]; j++ )
   for ( k = 0; k < in->data->nDataVar; k++ )
   {
    /* switch on the data type */
    /*
     This should be outside the loops for
     efficiency.  It is shown inside here
     for greater clarity.
    */
    switch ( in->data->primType )
    {
#define CASE(CXTYPE,TYPE) \
 case CXTYPE: \
  /* This is where the actual computation takes place */\
  *(TYPE *)b = -*(TYPE *)a; \
  a = (TYPE *)a + 1; b = (TYPE *)b + 1; \
  break;
     CASE(cx_prim_byte,unsigned char)
     CASE(cx_prim_short,short)
```

```
      CASE(cx_prim_long,long)
      CASE(cx_prim_float,float)
      CASE(cx_prim_double,double)
      default:
       break;
#undef CASE
    }
   }

}
```

Fortran Version:

```
C     This user function takes an image input
C     (2D lattice of any data type)
C     and takes the negative of every data element.
C
      SUBROUTINE IMAGE(IN,OUT)
C
      INCLUDE '/usr/explorer/include/cx/Typedefs.inc'
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
C
C     .. Scalar Arguments ..
      INTEGER          IN, OUT
C     .. Local Scalars ..
      INTEGER          COORD, I, IER, J, K, M, N, NDVAR, P, P0,
     *                 PTYPE
C     .. Local Arrays ..
      INTEGER *1       B0(1), B1(1)
      DOUBLE PRECISION D0(1), D1(1)
      REAL             F0(1), F1(1)
      INTEGER          DIMS(2), L0(1), L1(1)
      INTEGER *2       S0(1), S1(1)
C     .. External Functions ..
      EXTERNAL         CXLATDATANEW, CXLATDESCGET, CXLATPTRGET,
     *                 CXLATPTRSET
C     .. Pointers to Lattice Structures ..
      POINTER (PDIMS,DIMS)
      POINTER (PB0,B0),(PB1,B1)
      POINTER (PS0,S0),(PS1,S1)
      POINTER (PL0,L0),(PL1,L1)
      POINTER (PF0,F0),(PF1,F1)
      POINTER (PD0,D0),(PD1,D1)
C     .. Executable Statements ..
C
C     Extract information from the input lattice
C
      P0 = 0
      IER = CXLATDESCGET(IN,P0,PDIMS,P0,NDVAR,PTYPE,P0,P0,P0)
C
C     Compiler bug requires this hack
C
      N = DIMS(2)
      M = DIMS(1)
```

```
C
C     Create the new lattice
C
      OUT = CXLATDATANEW(2,DIMS,NDVAR,PTYPE)
C
C     Use the same coordinate space as the input lattice
C
      P0 = 0
      IER = CXLATPTRGET(IN,P0,P0,COORD,P0)
      P0 = 0
      IER = CXLATPTRSET(OUT,P0,%VAL(0),COORD,%VAL(0))
C
C     Extract data pointers based on primitive type
C
      IF (PTYPE.EQ.CX_PRIM_BYTE) THEN
         P0 = 0
         IER = CXLATPTRGET(IN,P0,PB0,P0,P0)
         P0 = 0
         IER = CXLATPTRGET(OUT,P0,PB1,P0,P0)
      ELSE IF (PTYPE.EQ.CX_PRIM_SHORT) THEN
         P0 = 0
         IER = CXLATPTRGET(IN,P0,PS0,P0,P0)
         P0 = 0
         IER = CXLATPTRGET(OUT,P0,PS1,P0,P0)
      ELSE IF (PTYPE.EQ.CX_PRIM_LONG) THEN
         P0 = 0
         IER = CXLATPTRGET(IN,P0,PL0,P0,P0)
         P0 = 0
         IER = CXLATPTRGET(OUT,P0,PL1,P0,P0)
      ELSE IF (PTYPE.EQ.CX_PRIM_FLOAT) THEN
         P0 = 0
         IER = CXLATPTRGET(IN,P0,PF0,P0,P0)
         P0 = 0
         IER = CXLATPTRGET(OUT,P0,PF1,P0,P0)
      ELSE IF (PTYPE.EQ.CX_PRIM_DOUBLE) THEN
         P0 = 0
         IER = CXLATPTRGET(IN,P0,PD0,P0,P0)
         P0 = 0
         IER = CXLATPTRGET(OUT,P0,PD1,P0,P0)
      END IF
C
C     Loop over all the data items
C
      P = 1
      DO 60 I = 1, N
         DO 40 J = 1, M
            DO 20 K = 1, NDVAR
C
C                Do the computation. The data type switching should be
C                outside the loop for efficiency.  It is shown inside
C                here for greater clarity.
C
               IF (PTYPE.EQ.CX_PRIM_BYTE) THEN
                  B1(P) = -B0(P)
```

```
               ELSE IF (PTYPE.EQ.CX_PRIM_SHORT) THEN
                  S1(P) = -S0(P)
               ELSE IF (PTYPE.EQ.CX_PRIM_LONG) THEN
                  L1(P) = -L0(P)
               ELSE IF (PTYPE.EQ.CX_PRIM_FLOAT) THEN
                  F1(P) = -F0(P)
               ELSE IF (PTYPE.EQ.CX_PRIM_DOUBLE) THEN
                  D1(P) = -D0(P)
               END IF
C
C              Increment the data item
C
               P = P + 1
   20       CONTINUE
   40    CONTINUE
   60 CONTINUE
C
      RETURN
      END
```

3.7.3. A 3–D Curvilinear Lattice

This example illustrates how to create a curvilinear lattice and rotate the coordinates. Test it by connecting *GenLat* to this module and *PrintLat*; this module to another copy of *PrintLat*. Compare the output from both PrintLat modules. The code resides in */usr/explorer/src/MWGcode/Lattice/C/Curvi3D.c* and in */usr/explorer/src/MWGcode/Lattice/Fortran/Curvi3D.f*.

C Version:

```
#include <stdio.h>
#include <math.h>

#include <cx/DataAccess.h>
#include <cx/DataExtract.h>
#include <cx/cxLattice.h>
#include <cx/cxLattice.api.h>

void cwind(float rot,cxLattice *in,cxLattice **out)
{
 int i,j,k;  /* loop variables */
 int index[3]; /* index vector for coordinate lookup */
 float *c;  /* pointer to output lattice coordinates */
 float coord[3]; /* input coordinate vector */
 float cosa,sina; /* utility variables */

 /* create the new lattice */
 *out = cxLatCoordNew(
 3,   /* number of dimensions */
 in->dims,  /* dimensions array */
 3,   /* coordinate dimensions */
 cx_coord_curvilinear); /* type of coordinates */

 /* use the same data from the input lattice */
 cxLatPtrSet(*out,in->data,NULL,NULL,NULL);
```

```
 /* get the coordinate data pointer */
 cxLatPtrGet(*out,NULL,NULL,NULL,(void **)&c);

 /* loop over the elements */
 for ( i = 0; i < in->dims[2]; i++ )
 {
  index[2] = i; /* set z index */

  for ( j = 0; j < in->dims[1]; j++ )
  {
   index[1] = j; /* set y index */

   for ( k = 0; k < in->dims[0]; k++ )
   {
    index[0] = k; /* set x index */

    /* get the original coordinates */
    cxLatCoordExtract(in,index,coord);

    /* wind the coordinates around the z-axis */
    cosa = cos(rot*coord[2]);
    sina = sin(rot*coord[2]);

    c[0] = coord[0]*cosa + coord[1]*sina;
    c[1] = coord[0]*sina - coord[1]*cosa;
    c[2] = coord[2];

    /* increment pointer */
    c += 3;
   }
  }
 }

}
```

Fortran Version:

```
      SUBROUTINE FWIND(ROT,IN,OUT)
C
      INCLUDE '/usr/explorer/include/cx/cxLattice.inc'
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
      INCLUDE '/usr/explorer/include/cx/Typedefs.inc'
C
C     .. Scalar Arguments ..
      REAL             ROT
      INTEGER          IN, OUT
C     .. Local Scalars ..
      REAL             COSA, SINA
      INTEGER          D, I, IER, J, K, M, NX, NY, NZ, P0
C     .. Local Arrays ..
      REAL             C(1), COORD(3)
      INTEGER          DIMS(3), INDEX(3)
C     .. External Subroutines ..
```

```
      EXTERNAL         CXLATCOORDEXTRACT
C     .. External Functions ..
      EXTERNAL         CXLATCOORDNEW, CXLATDESCGET, CXLATPTRGET,
     *                 CXLATPTRSET
C     .. Intrinsic Functions ..
      INTRINSIC        COS, SIN
C     .. Pointers to Lattice Structures ..
      POINTER (PDIMS,DIMS)
      POINTER (PC,C)
      POINTER (PA,A)
C     .. Executable Statements ..
C
C     Get the dimensions vector
C
      P0 = 0
      IER = CXLATDESCGET(IN,P0,PDIMS,P0,P0,P0,P0,P0,P0)
      NX = DIMS(1)
      NY = DIMS(2)
      NZ = DIMS(3)
C
C     Create the new lattice
C
      OUT = CXLATCOORDNEW(3,DIMS,3,CX_COORD_CURVILINEAR)
C
C     Use the same data as the input lattice
C
      P0 = 0
      IER = CXLATPTRGET(IN,D,PA,P0,PA)
      P0 = 0
      IER = CXLATPTRSET(OUT,D,%VAL(0),P0,%VAL(0))
C
C     Get the output data pointer
C
      P0 = 0
      IER = CXLATPTRGET(OUT,P0,P0,P0,PC)
C
C     Loop over the coordinates
C
      M = 1
C
C     Set z index (0 based for api routine)
C
      DO 60 I = 1, NZ
         INDEX(3) = I - 1
C
C           set y index
C
         DO 40 J = 1, NY
            INDEX(2) = J - 1
C
C           Set x index
C
            DO 20 K = 1, NX
               INDEX(1) = K - 1
```

```
C
C              Get the original coordinates
C
               CALL CXLATCOORDEXTRACT(IN,INDEX,COORD)
C
C              Wind the coordinates around the z-axis
C
               COSA = COS(ROT*COORD(3))
               SINA = SIN(ROT*COORD(3))
C
               C(M) = COORD(1)*COSA + COORD(2)*SINA
               C(M+1) = COORD(1)*SINA - COORD(2)*COSA
               C(M+2) = COORD(3)
C
C              Increment index
C
               M = M + 3
   20       CONTINUE
   40    CONTINUE
   60 CONTINUE
C
      RETURN
      END
```

Chapter 4

Using the Pyramid Data Type

This chapter discusses the *cxPyramid* data type and how it is used in the IRIS Explorer environment. It describes the pyramid structure and explains the concept of pyramid compression to save time and effort when constructing a pyramid. It also discusses how you use pyramid dictionaries to build the elements of a pyramid and lists the elements available in the IRIS Explorer standard pyramid dictionary.

The API (Application Programming Interface) routines are listed and examples of user function code for writing modules that build and manipulate pyramids are included.

4.1. Understanding Pyramids

Pyramids hold two kinds of data: irregular or unstructured grid data, such as finite element data, and molecular modeling data. In IRIS Explorer, the pyramid data type is used primarily for finite element modelling and creating irregular grids, and most IRIS Explorer pyramid modules handle only this kind of data, not molecular data.

The pyramid data structure, *cxPyramid*, defines the relationship between the different layers of data required to build a pyramidal structure. In finite element data, you can consider the element grid as a collection of vertices (points), edges (lines), faces (polygons), elements (three−dimensional cells), objects (collections of 3−D elements), assemblies (collections of objects), and so on. You can fit these together to make an object from faces, bounded by edges, which are in turn delimited by vertices.

Because *cxPyramid* combines several data types, it is an extremely flexible and powerful structure; witness its ability to handle both finite element data and molecular data. The flexibility of the *cxPyramid* data type also makes it somewhat difficult to use properly. It may take a little practice before the method of creating pyramids out of an hierarchical collection of points, lines, polygons, and elements makes sense.

In many cases where you are representing finite element data, it is simplest to use the *pyramid dictionary* representation. The reason is that the dictionary elements already contain the detailed hierarchy of points, lines, and faces. You simply create a list of vertices making up the overall finite element structure or irregular grid, and then describe the final structure as a collection of element types, each of which depends on certain identified vertices. Use of the dictionary also serves to cut down on the unnecessary repetition of information in the interests of computational and storage efficiencies.

You can write two kinds of pyramid modules; those that read pyramid data, such as *ComposePyr*, and those that process pyramid data, such as *IsosurfacePyr*. It is relatively easy to write a pyramid reader module without having to be an expert on pyramid structure and the first part of this chapter explains how to do so, using two code examples. If you want to write a pyramid processing module, you need to understand the finer details of pyramid composition, and these are discussed later in the chapter.

4.1.1. The Structure of a Pyramid

The *cxPyramid* data type is a root data type, which means it can be used on module ports to pass data into and out of modules. A pyramid consists of three main parts:

- the several layers of pyramidal data; for example, points, lines, and faces. These values are collected in *cxLattice.*

  The structure of *cxLattice* is described in **Chapter 3, "Using the Lattice Data Type"**.
- the relationships between these layers, described by *cxPyramidLayer*. You really need to know exactly how your lattices are constructed before you try to create the relationships that combine them.

  Pyramid layers and connections are described in detail in **"Pyramid Layer Connections"**.
- optional references to predefined pyramid elements, which are stored in a dictionary, *cxPyramidReference*. Pyramid dictionaries are described in detail in **"Structure of a Dictionary"**.

This is the data type definition for *cxPyramid*:

```
root typedef struct cxPyramid {
  cxLattice          baseLattice     "Base Lattice";
  long               count           "Num Levels";
  cxPyramidLayer     layer[count];
  cxPyramidReference ref;
} cxPyramid;
```

These are the variables for the pyramid:

baseLattice A 1−D curvilinear lattice that defines the vertices of the finite element pyramid. (It defines the bonds of a molecular pyramid.) The *baseLattice* must have coordinates in order to define the location of the vertices. Most finite element pyramid modules require data values as well.

count The count is an integer that indicates how many layers make up the pyramid, and thus its dimensionality. Layers are numbered from 0 to *count−1*, excluding the *baseLattice*. For example, a pyramid with *count*=4 would represent a collection of many 3−D elements (see **Figure 4−3**).

layer Each layer consists of a lattice and a *cxConnection* structure describing its relationship to the adjacent lattice one layer down. The explicit connections between lattices at adjacent levels determine the shape of each pyramid component, and hence the shape of the entire pyramid. The lattice values at each layer are optional, but the relationship structures are required at all layers in the pyramid. The definition and structure of the *cxPyramidLayer* and *cxConnection* data types are described in **"Pyramid Layer Connections"**.

ref Describes the pyramid dictionary, which is a set of predefined pyramidal data structures that you can incorporate by reference into your pyramid. See **"Structure of a Dictionary"** for a description of the *cxPyramidReference* data type.

A pyramid that incorporates a pyramid dictionary is said to be a *compressed pyramid* as opposed to the *uncompressed pyramids* lacking such a dictionary. A *pyramid dictionary* is simply a collection of uncompressed pyramids that describe in full detail their reference elements.

4.1.2. Pyramid Dictionaries

You can use pyramid dictionaries to:

- Save storage space. A finite element pyramid composed of many cells may take up a great deal of storage space, which is expensive in computing terms. This is especially true for a pyramid composed entirely of one kind of cell, such as hexahedrons (also called bricks). A pyramid dictionary offers a way for you to store and access represenions of pyramid cells without having to list every single face and edge in your pyramid data structure.
- Add your own dictionary entries. IRIS Explorer provides a standard reference dictionary of cell types which you can augment by adding any cells you wish. The dictionary is available through API calls.

The *cxPyramidReference* data structure, described in **"Structure of a Dictionary"**, lets you specify the composition of your pyramid in terms of already−defined pyramidal elements. For example, to create a tetrahedral finite element grid, you would specify the element type as tetrahedral, list the vertices necessary to create the grid, and enumerate the constituent tetrahedra based on their four corner vertices.

For a grid of hexahedral bricks, you would specify the type as hexahedral, list the vertices necessary to create the grid, and enumerate the constituent hexahedral bricks based on their eight corner vertices. This is illustrated in the *SimplePyrReader* module described below.

The best way to learn how to use the pyramid data type is by example. The next section provides two examples of user functions for pyramid modules.

4.1.3. A Pyramid Reader Module

This example, called *SimplePyrReader*, takes in nodal points, reads them and makes hexahedrons from them. It is a good template for creating any pyramid reader module that deals with only one data type. It uses compression and refers to the default pyramid dictionary. The C source code is in */usr/explorer/src/MWGcode/Pyramid/C/reader.c* and the Fortran source

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

is in */usr/explorer/src/MWGcode/Pyramid/Fortran/reader.f*. The directories also contain copies of an example input data file, *sample.data* (see below), and an example map *SamplePyrReader.map.* You can test the module by building it, loading the example map into the Map Editor, and viewing the points, vectors and polygons in the pyramid.

This is the *sample.data* file, which contains toy finite element data. The reading of this file (see the source code below) allows for blank lines, but no other text (such as comments, for example).

```
12 1 2

0 0 0 100
0 1 0 100
1 1 0 100
1 0 0 100
0 0 1 200
0 1 1 200
1 1 1 200
1 0 1 200
0 0 2 300
0 1 2 300
1 1 2 300
1 0 2 300

0 1 2 3 4 5 6 7
4 5 6 7 8 9 10 11
```

The first line gives the number of nodes (12), the number of data variables (1) and the number of hexahedrons or bricks (2). The next 12 lines list the coordinates (x,y,z) and the data value for each node. The data variable is pressure, changing in the Z direction. The bricks have eight nodes each, but share one face (four nodes) so there is a total of 12 nodes.

The last two lines define the connectivity list, or the relationship between the nodes that make up the bricks. The two bricks share the face delineated by nodes 4, 5, 6, and 7. The nodes are connected in the sequence indicated in the first list; the first brick is defined in the penultimate line, and the second brick is defined in the final line. The form of the hexahedron in the default pyramid dictionary is shown in **Figure 4–1**.

This is the user function for *SimplePyrReader.*

C Version:

```
/*
This module is a prototype reader of finite element data.
It reads in nodal points and then the connectivity for the cells.
This code assumes that only one cell type is used. The user
can alter the two define's for a different cell type.
*/

#include <cx/cxPyramid.h>
#include <cx/Pyramid.h>
#include <cx/cxPyramid.api.h>
#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define NODES 8
#define TYPE cx_pyramid_dict_hexahedron

int memclean(cxPyramid *pyr,
      cxErrorCode ec,
      cxConnection *conn)
{
/* this routine will clear up memory in case of errors */
  if(cxDataAllocErrorGet() || (ec != cx_err_none))
  {
    cxDataRefDec(pyr);
    cxDataRefDec(conn);
    return 1;
  } else {
    return 0;
  }
}

void  read_brick  (char *filename,
     cxLattice **base_lat,
     cxPyramid ** pyramid )
{
  FILE *file;
  float *coord, *data;
  cxConnection *conn;
  cxPyramidDictionary *dict;
  cxErrorCode ec = cx_err_none;
  long *connec, num_nodes, num_bricks;
  long nDataVar,i,j,k, *elem;

  if(filename == NULL) return;
  if(filename[0] == NULL) return;
  if((file = fopen(filename,"r")) == NULL)
  {
    printf("Cannot open file %s\n",filename);
    return;
  }
/*
  get the header information
   Number of Nodes,
   Number of Data Properties,
   Number of elements
*/
  fscanf(file,"%d %d %d",&num_nodes, &nDataVar, &num_bricks);
  *base_lat = cxLatNew(1,&num_nodes,nDataVar,cx_prim_float,
         3,cx_coord_curvilinear);
  if(cxDataAllocErrorGet()) return;
  cxLatPtrGet(*base_lat,NULL,(void **)&data,NULL,(void **)&coord);

/*
```

```
  read in the nodal data. This assumes there are 3 coords
  and nDataVar data values for each node.
*/
  for(i=0,k=0;i<num_nodes;i++)
  {
    fscanf(file," %f %f %f",&coord[3*i],&coord[3*i+1],&coord[3*i+2]);
    for(j=0;j<nDataVar;j++)
       fscanf(file,"%f",&data[k++]);
  }

/* build up the pyramid by ignoring the intermediate levels */
  *pyramid = cxPyrNew(0);
  cxPyrSet(*pyramid,*base_lat);
  cxPyrLayerSet(*pyramid,1,NULL,NULL);
  cxPyrLayerSet(*pyramid,2,NULL,NULL);

/* set up the connections list */

  conn = cxConnNew(num_bricks, num_bricks * NODES);
  if(memclean(*pyramid,ec,0))return;
  cxConnPtrGet(conn,NULL,NULL,&elem,&connec);
  *elem++ = 0;
  for(i=0,k=0;i<num_bricks;i++)
  {
    *elem++ = (i+1) * NODES;  /* each has NODES nodes per brick */
    for(j=0;j<NODES;j++)
    {
      fscanf(file,"%d",&connec[k++]); /* read in nodal list */
    }
  }
  cxPyrLayerSet(*pyramid,3,conn,NULL);
  if(memclean(*pyramid,ec,conn))return;

/* make the pyramid dictionary */

  dict = cxPyrDictDefault( 3 );
  cxPyramidDictionarySet( *pyramid, dict, &ec );
  if(memclean(*pyramid,ec,conn))return;
  cxPyramidCompressionTypeSet( *pyramid,
         cx_compress_unique, &ec );
  if(memclean(*pyramid,ec,conn))return;
  connec = (long *)cxPyramidCompressionIndexGet(*pyramid,&ec);
  if(memclean(*pyramid,ec,conn))return;
  *connec = TYPE;
}
```

Fortran Version:

```
      SUBROUTINE RDBRIK(FILNAM,BASLAT,PYRMID)
C
C     This module is a prototype reader of finite element data.
C     It reads in nodal points and then the connectivity for the cells.
C     This code assumes that only one cell type is used. The user
C     can alter the two define’s for a different cell type.
```

```
C
      INCLUDE '/usr/explorer/include/cx/Pyramid.inc'
      INCLUDE '/usr/explorer/include/cx/cxPyramid.api.inc'
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
      INCLUDE '/usr/explorer/include/cx/DataTypes.inc'
C
C     .. Parameters ..
      INTEGER           NODES
      PARAMETER         (NODES=8)
C     .. Scalar Arguments ..
      INTEGER           BASLAT, PYRMID
      CHARACTER*(*)     FILNAM
C     .. Local Scalars ..
      INTEGER           CONN, DICT, EC, I, IER, IFILE, J, K, NDVAR,
     *                  NBRIKS, NNODES
C     .. Local Arrays ..
      REAL              COORD(3,1), DATA(1)
      INTEGER           CONNC(1), ELEM(1)
C     .. External Functions ..
      INTEGER           MEMCLN
      EXTERNAL          CXCONNNEW, CXCONNPTRGET, CXDATAALLOCERRORGET,
     *                  CXLATNEW, CXLATPTRGET,
     *                  CXPYRAMIDCOMPRESSIONINDEXGET, CXPYRDICTDEFAULT,
     *                  CXPYRLAYERSET, CXPYRNEW, CXPYRSET, MEMCLN
C     .. External Subroutines ..
      EXTERNAL          CXPYRAMIDCOMPRESSIONTYPESET,
     *                  CXPYRAMIDDICTIONARYSET
C     .. Intrinsic Functions ..
      INTRINSIC         ICHAR
C     .. Pointers to Lattice Structures ..
      POINTER(PDATA,DATA), (PCOORD,COORD)
      POINTER(PELEM,ELEM), (PCONNC,CONNC)
C     .. Executable Statements ..
C
C     Get the file name from which to read the data
C
      IF (ICHAR(FILNAM(1:1)).EQ.0) RETURN
      IFILE = 7
      EC = CX_ERR_NONE
      OPEN (IFILE,FILE=FILNAM,STATUS='old',IOSTAT=IER)
      IF (IER.NE.0) THEN
         PRINT *, 'Cannot open file', FILNAM
         RETURN
      END IF
C
C     Get the header information (number of Nodes, number of data
C     properties, number of elements)
C
      READ (IFILE,FMT=*) NNODES, NDVAR, NBRIKS
      BASLAT = CXLATNEW(1,NNODES,NDVAR,CX_PRIM_FLOAT,3,
     *         CX_COORD_CURVILINEAR)
      IF (CXDATAALLOCERRORGET().NE.0) RETURN
      IER = CXLATPTRGET(BASLAT,I,PDATA,J,PCOORD)
C
```

```
C     Read in the nodal data. This assumes there are 3 coords
C     and nDataVar data values for each node.
C
      DO 20 I = 1, NNODES
         READ (IFILE,FMT=*) COORD(1,I), COORD(2,I), COORD(3,I),
     *      (DATA((I-1)*NDVAR+J),J=1,NDVAR)
   20 CONTINUE
C
C     Build up the pyramid by ignoring the intermediate levels
C
      PYRMID = CXPYRNEW(0)
      IER = CXPYRSET(PYRMID,BASLAT)
      IER = CXPYRLAYERSET(PYRMID,1,0,0)
      IER = CXPYRLAYERSET(PYRMID,2,0,0)
C
C     Set up the connections list
C
      CONN = CXCONNNEW(NBRIKS,NBRIKS*NODES)
      IF (MEMCLN(PYRMID,EC,0).NE.0) RETURN
      IER = CXCONNPTRGET(CONN,I,J,PELEM,PCONNC)
      ELEM(1) = 0
      DO 40 I = 1, NBRIKS
         ELEM(I+1) = I*NODES
         K = (I-1)*NODES
         READ (IFILE,FMT=*) (CONNC(J+K),J=1,NODES)
   40 CONTINUE
      IER = CXPYRLAYERSET(PYRMID,3,CONN,0)
      IF (MEMCLN(PYRMID,EC,CONN).NE.0) RETURN
C
C     Make the pyramid dictionary
C
      DICT = CXPYRDICTDEFAULT(3)
      CALL CXPYRAMIDDICTIONARYSET(PYRMID,DICT,EC)
      IF (MEMCLN(PYRMID,EC,CONN).NE.0) RETURN
      CALL CXPYRAMIDCOMPRESSIONTYPESET(PYRMID,CX_COMPRESS_UNIQUE,EC)
      IF (MEMCLN(PYRMID,EC,CONN).NE.0) RETURN
      PCONNC = CXPYRAMIDCOMPRESSIONINDEXGET(PYRMID,EC)
      IF (MEMCLN(PYRMID,EC,PCONNC).NE.0) RETURN
      CONNC(1) = CX_PYRAMID_DICT_HEXAHEDRON
      CLOSE (IFILE)
C
      RETURN
      END
      INTEGER FUNCTION MEMCLN(PYR,EC,CONN)
C
C     Clear up memory in case of errors
C
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
      INCLUDE '/usr/explorer/include/cx/Typedefs.inc'
C
C     .. Scalar Arguments ..
      INTEGER                  CONN, EC, PYR
C     .. External Subroutines ..
      EXTERNAL                 CXDATAREFDEC
```

```
C      .. External Functions ..
       EXTERNAL                  CXDATAALLOCERRORGET
C      .. Executable Statements ..
C
       MEMCLN = 0
       IF ((CXDATAALLOCERRORGET().NE.0) .OR. (EC.NE.CX_ERR_NONE)) THEN
          CALL CXDATAREFDEC(PYR)
          MEMCLN = 1
          CALL CXDATAREFDEC(CONN)
       END IF
C
       RETURN
       END
```

Reading Several Data Types

If you want to build a pyramid reader module that can handle several data types, use the source code for *ComposePyr* as a template. It references the complete pyramid dictionary. The source code is in */usr/explorer/src/ComposePyr/compose.c.*

4.1.4. Using the Dictionary Elements

The dictionary reference elements serve as building blocks from which the final pyramid is constructed. They include 2−D elements, such as triangles and quadrilaterals, and 3−D elements, such as hexahedrons (or bricks) and tetrahedrons. The IRIS Explorer default dictionary contains some of the most commonly used reference elements (see **Figure 4−1**).

The vertices are labelled in the order that IRIS Explorer requires to make each form, counter−clockwise on the bottom and then counter− clockwise on the top. You provide this information in the *connections* list in *cxConnection* (see **"Creating a Connection List"**).

Note: The vertex labelling of these elements is critical. The sequence determines how the vertices you provide in the connection list are interpreted. If you label the vertices differently, your finite element mesh will be made up of distorted or inverted elements.

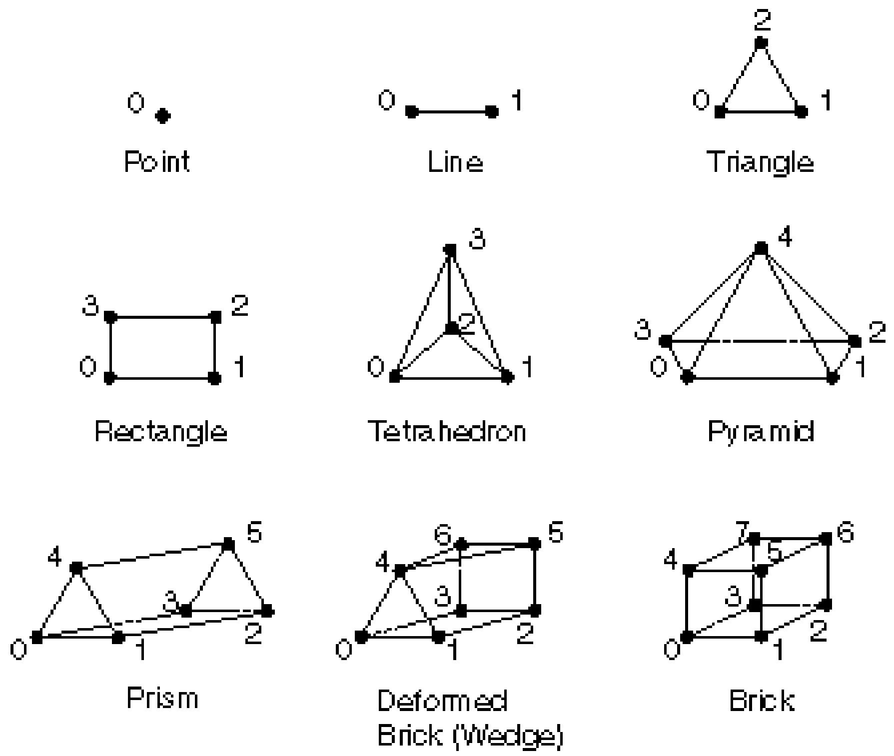


Figure 4–1 Pyramid Dictionary Reference Elements

Table 4–1 lists the elements illustrated above, with the dictionary name and number of nodes that make up each element.

Note: `cx_pyramid_dict_brick` and `cx_pyramid_dict_hexahedron` are synonyms; they can be used interchangably.

Dimension	Element	No. of Nodes	Name
0–D	Point	1	`cx_pyramid_dict_point`
1–D	Line	2	`cx_pyramid_dict_line`
2–D	Triangle	3	`cx_pyramid_dict_triangle`
	Rectangle	4	`cx_pyramid_dict_quadrilateral`
3–D	Tetrahedron	4	`cx_pyramid_dict_tetrahedron`
	Pyramid	5	`cx_pyramid_dict_pyramid`
	Prism	6	`cx_pyramid_dict_prism`
	Deformed brick (wedge)	7	`cx_pyramid_dict_wedge`
	Brick (hexahedron)	8	`cx_pyramid_dict_brick`, `cx_pyramid_dict_hexahedron`

Table 4–1 Standard Dictionary Reference Elements

4.2. Finite Element Pyramids

This pyramid type is useful for storing the results of finite element–based simulations. It consists of several levels: the base lattice and a number of layers, each of which contains a lattice with data and coordinates, if needed, and the connections between layers. The number of layers in a pyramid is infinite in theory, although in practice, pyramids tend to be self–limiting. A pyramid usually has at least three levels, including a base lattice.

Setting up the Lattice Structure

The structure of the finite element pyramid requires that the lattice at each level *must* be a 1–D curvilinear lattice. The 1–D restriction is so that the indexing information in the connections list makes sense. But there is a natural ordering from the nodes in a 3–D lattice to nodes in a 1–D lattice, so you can break out data from a 3–D lattice fairly easily into a 1–D lattice for inclusion in a pyramid. The natural ordering, or memory layout, of data in two or more dimensions gives the usual 1–D indexing into the array.

Figure 4–2 shows a 2–D lattice that, when translated into a 1–D curvilinear lattice (for inclusion in a pyramid layer, say) has 35 nodes, indexed as 0–34.

Note: In this figure, we have assumed the data is laid out in the C language (row–major) convention (see **Figure 3–4**).

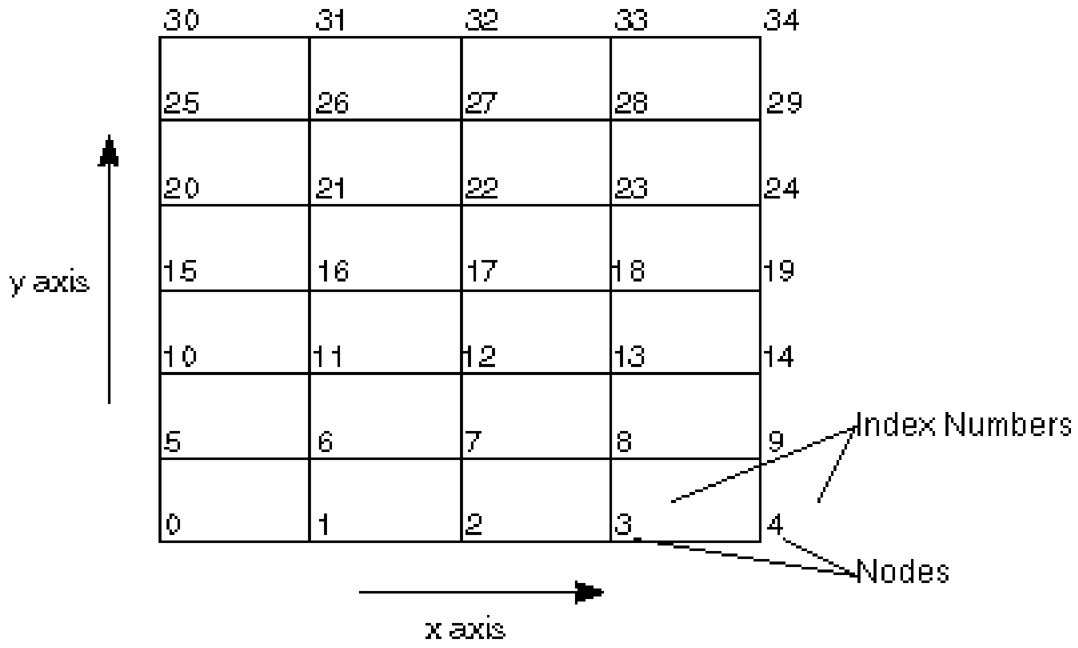


Figure 4–2 Memory Layout of a 2–D Lattice

Contents of Layers

Not all lattices in a pyramid need have data or coordinates. The base lattice, which contains all the nodal data and coordinate values, is frequently the only lattice that contains coordinate information at all. The 0–layer lattice can store edge–based data, and the 1–layer lattice can contain face–centered data, such as the mass flux, for example. The 2–layer lattice can store volumetric or centroid data, such as a mass fraction, centroid pressure, or gauss point information. In addition, the lattice at each layer can store the physical location of each piece of data in its coordinates part. For example, the mass flux through a face could be assigned to the coordinates of the midpoint of the face, or some other location on the face. Similarly, cell–based data such as pressure or composition could be explicitly associated with a position within the cell (at, say, its center, or some other point). See **"The Pyramid Layers"** below for more about the way in which the lattice is defined within the layer.

Designing a Finite Element Mesh

The *ClipPyr* and *CropPyr* modules work by removing elements at the highest level in the pyramid, if those elements contain any vertices outside the bounding plane(s). Therefore, you should construct finite element meshes with the top level being the most natural component. Often, this will mean that a finite element mesh has three layers in the pyramid (i.e. *count* = 3) and is composed of a large number of 3–D elements. This is the default behavior of the *LatToPyr* module, which forms a tetrahedral or hexahedral mesh from a lattice.

4.2.2. Creating a Tetrahedral Grid

The pyramid structure is built from lattices layered onto the base lattice. **Figure 4–3** shows the relationship between elements and layers in an uncompressed tetrahedral pyramid. It illustrates how a finite element grid can be built from the hierarchy of relationships between four vertices.

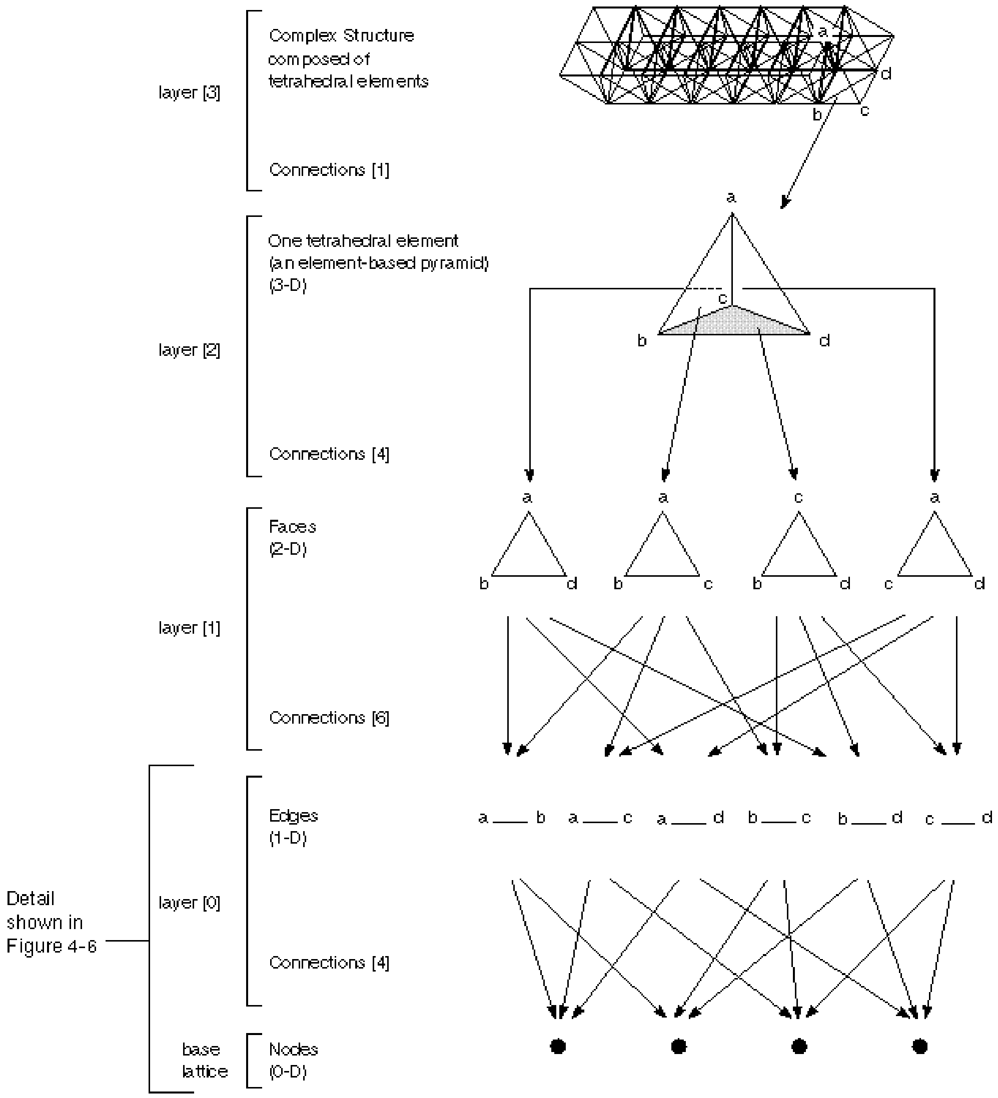


Figure 4–3 Layers in a Tetrahedral Grid

However, the simplest tetrahedral grid is just an array of tetrahedra, defined by specifying their vertices. The vertex locations and data are stored in the base lattice. You can omit edge and face information and go directly to the 3–D element layer of the pyramid. The *cxConnection* structure of this layer (`layer[2]` in C or `layer(3)` in Fortran) really does all the work of constructing the tetrahedral grid.

The example in **Figure 4–3** is made up of 32 tetrahedra, and so we can start to set values for some of the members of the *cxConnection* component of the layer:

```
numElements       32
elemArrayLen      33
elements[]         0   4   8   12   16 ... 128
```

Here, `numElements` is the number of *elements* (i.e., tetrahedra in the current example) in this layer, while `elements` is an array of length `elemArrayLen` which contains the indices into the `connections` array (see below) for each element in this layer.

The vertices are labelled in the order shown in **Figure 4–4**. Grid A shows the vertices at the bottom of the structure and grid B shows those at the top. Grid C shows the top partially overlaid on the bottom to illustrate how the tetrahedra are formed.

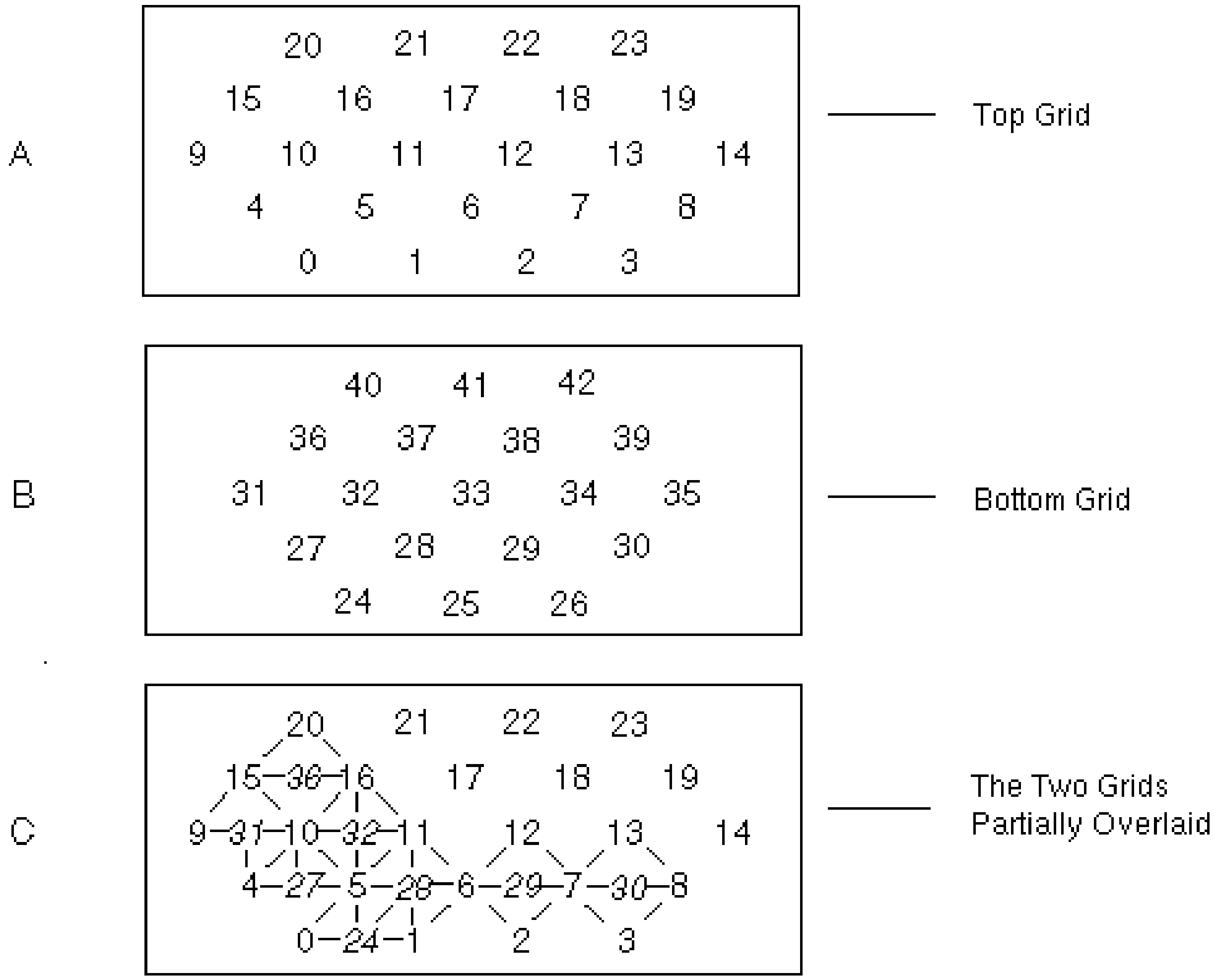


Figure 4–4 Top and Bottom of Compressed Tetrahedral Grid

The connections array is the concatenation of the sets of four vertices for the 32 tetrahedra:

```
numConnections  128
connections[]     0  1  5 24
                  1  2  6 25
                  2  3  7 26
                       ...
                 18 23 22 42
```

where `connections` is an array of length `numConnections` which contains the indices of the elements in the layer beneath which are to be connected together to construct the elements in this layer. The `elements` array is used to step along `connections` to extract the indices for the individual elements. Thus, element number `i` in this layer is built up by connecting together element numbers `connections[elements[i]]`, `connections[elements[i]+1]`, ... `connections[elements[i+1]-1]` from the layer beneath this one. Hence, element number `i` in this layer is made up of connections to `N` elements from the layer beneath, where `N = elements[i+1] - elements[i]`. See **"Pyramid Layer Connections"** for more about the members of *cxConnection*.

The construction of the tetrahedra in the example of **Figure 4–4** has been done directly in terms of their vertices (rather than their faces, edges and vertices), which, in the discussion above, are the elements in the layer "beneath" `layer[2]`, which defines the 3–D elements of the pyramid. We have been able to bypass the intervening layers (which would ordinarily contain face and edge connectivity information) because we are using a *compressed* pyramid – i.e. one which contains a pyramid dictionary. This is described in more detail in the following section.

4.2.3. Using a Pyramid Dictionary

When you use a pyramid dictionary, some of the cell structures in the pyramid are *compressed* and you may not have to enumerate the faces or edges making up the finite element grid. In general, it is unnecessary to specify any relationship information for the layers between the vertices and the layer at which dictionary elements are referenced. It is this careful omission of repetitive data that makes the pyramid dictionary structure simpler and more compact to use.

However, you must still define the lattice and connection list pairs in the `layer[]` array above the "compression dimension" (see the definition of *nDim* in **"Structure of a Dictionary"**, below). But below the compression dimension *nDim*, there are no lattice or *cxConnection* structures; these structures are superseded by the reference elements in the dictionary. The base lattice still holds the vertex data and coordinates necessary to give the compressed pyramid a location in physical space and to supply the data requirements of the pyramid modules.

In addition to the omission of some layers, there is an important change in the *cxConnection* structure at the compression layer. At `layer[nDim-1]` (in C) or `layer(nDim)` (in Fortran), the *cxConnection* structure must describe the composition of the compressed elements in terms of their vertices. You create such a connection list by listing the actual vertex indices of the *baseLattice* vertices you wish to go into the construction of each element, as in the example in **"Creating a Tetrahedral Grid"**.

The integer value that identifies the type of an element is really just an index into the pyramid dictionary. Since the dictionary is merely an array of uncompressed pyramids, it is natural to use the array index of the dictionary element as the identifying value. In the case of a grid of a single element type, however, the vector of identical indices is omitted in favor of a single universal element type. See **"Structure of a Dictionary"** below for how to specify these different cases.

Figure 4–5 shows how the tetrahedral grid looks in compressed form.

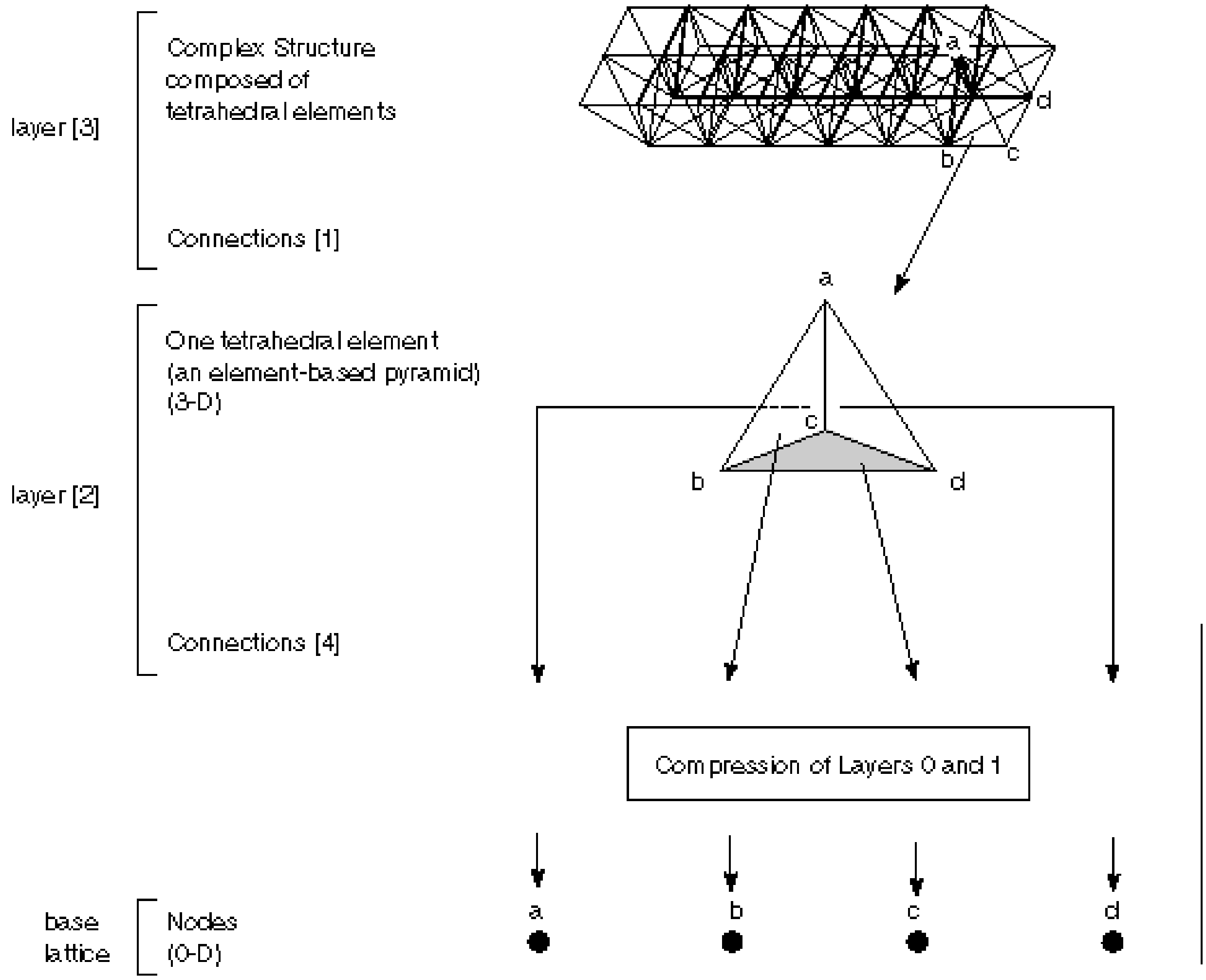


Figure 4–5 Layers in a Compressed Grid

Figure 4–3 shows the same grid in uncompressed form, with all the layers visible.

Advantages

Using the pyramid dictionary has two important advantages. Such a pyramid is easier to create, because you need only consider the vertices that make up each element. It also requires much less storage than its fully expanded version, because each reference element represents the internal face/edge/vertex hierarchy once for all instances of the type.

Disadvantages

Using the pyramid dictionary has one drawback, the significance of which depends on your application. Because the face and edge structure is contained entirely within the omitted layers of a 3–D compressed pyramid, such a pyramid has absolutely no representation of the sharing of faces or edges within the pyramid. An algorithm that requires connectivity information between cells of a finite element mesh will find such information is not present in a compressed pyramid.

To use a compressed pyramid in this context, you must create your own data structure showing connectivity between the compressed elements of the pyramid. See **Chapter 8, "Creating User−defined Data Types"** for more information.

If you do not want to use the dictionary, you can represent an empty dictionary by setting the *cxPyramidReference* member *compressType* to `cx_compress_none`. In this case, *cxPyramidDictionary* is never referenced.

4.3. Structure of a Dictionary

Each dictionary has a dimensionality, which is also the dimensionality of all the elements found in it, and a vector of reference elements described as uncompressed pyramids. The data type structure is described by *cxPyramidDictionary.* **Figure 4−1** shows the contents of the default 3−D pyramid dictionary. These nine reference elements are described as a vector of nine pyramids, each of which comprises points, lines, and faces for that element.

The brick or hexahedron element, for example, comprises six faces, each of which comprises four out of the twelve edges found in the brick. Each of the twelve edges refers to two endpoint vertices. In the pyramid dictionary, the vertices are not given data or coordinates because they are generic elements, without location in Cartesian space or data values.

This is the type definition for the dictionary data type, *cxPyramidDictionary*:

```
shared typedef struct {
  long          nDim                        "Num Dimensions";
  long          numEntries                  "Num Entries";
  closed struct cxPyramid table[numEntries] "Table";
} cxPyramidDictionary;
```

These are the variables:

nDim — Gives the dimensionality of the pyramid dictionary, indicating the dimension at which compression takes place. If the reference elements are 3−D, then *nDim* is 3, the reference element pyramids have a *count* of 3, and the compressed data pyramid has no *cxConnection* structure at layers 0 or 1.
For *nDim* values of 1, 2, or 3, you may be able to use the IRIS Explorer default dictionary (call the API routine *cxPyrDictDefault* to generate it). For *nDim* > 3, you must provide your own dictionary.

numEntries — The total number of entries in the vector *table[]* of pyramid structures.

table[] — The list of reference elements stored in the dictionary. Each element of *table[]* is a pointer to a fully expanded *cxPyramid* structure. The default 3−D dictionary table has the reference elements listed in **Table 4−1**.

This is the type definition for the dictionary reference structure, *cxPyramidReference*:

```
typedef struct {
  cxCompressType      compressType    "Compression Type";
  switch (compressType) {
    case cx_compress_unique:            /* Use one element. */
      long        index               "Compression Index";
    case cx_compress_multiple:          /* Use many elements. */
      long        numIndices          "Number Indices";
      long        indices[numIndices] "Compression Index";
  } r;
  cxPyramidDictionary     dictionary  "Dictionary";
} cxPyramidReference;
```

These are the variables:

compressType — Indicates the existence of the pyramid dictionary and the access pattern into it.
A `compressType` value of `cx_compress_none` indicates that no dictionary is used in this

fully expanded pyramid.
A `compressType` value of `cx_compress_unique` indicates that a dictionary is used, but that the current finite element mesh is composed of only a single reference element type.
A `compressType` value of `cx_compress_multiple` indicates that a dictionary is used and that the current finite element mesh is composed of one or more reference element types. In this case, each element at layer *nDim−1* (in C indexing notation) in the pyramid's `layer[]` array has its own reference element index.

index — The single index type of a `cx_compress_unique` pyramid.

numIndices — The number of different reference element types selected from the dictionary.

Indices — The vector of index types of a `cx_compress_multiple` pyramid. Each index is the array index into `table[]` of the reference pyramid to be used in giving structure to the element's vertices.

Note: There is no case statement in the `switch (compressType)` for `cx_compress_none`, because the case would have no members (since there is no compression) and this is prohibited by the IRIS Explorer data typing language. Hence, it doesn't appear in the data type definition. Similarly, it is absent from the data type *declaration* as well (see **"Data Type Declaration"**, below).

4.4. Pyramid Layer Connections

You use the connection list in each layer of a pyramid to define how elements in that layer and the one immediately below it are related. For example, the connection list in layer 0 of a tetrahedron represents the set of connections between the edge elements in layer 0 and the nodes, or point elements, in the base lattice. Each line element, or edge, is connected to two nodes, or vertices, and the concatenation of all connections is represented by a pair of arrays, *connections* and *elements*, in *cxConnection*.

The Pyramid Layers

The pyramid layers are described by the *cxPyramidLayer* structure. This is the type definition for a single pyramid layer:

```
typedef struct {
  closed cxConnection relation       "Layer %d Connection";
  closed cxLattice    nextLattice    "Layer %d Lattice";
} cxPyramidLayer;
```

These are the variables:

relation — Defines the way in which elements in this layer are built up from connections to elements in the layer beneath. The *cxConnection* structure is defined **below**.

nextLattice — The lattice containing data and coordinates for the elements in this layer, if required. It must be a 1−D curvilinear lattice, with *dims[0]*, the number of nodes, equal to the number of elements in this layer (i.e. *numElements*, see **below**). For example, in the pyramid shown in **Figure 4−3**, if there was to be data associated with the edges of the structure, it would be inserted into the data part of the *cxLattice* component of `layer[0]`; the lattice would have `dims[0]` = 6, the number of edges, and would contain data values for each edge. The edges would be specified in the same numerical order that is used when assigning the connections to the elements in the *cxConnection* part of the layer. If there was a location to be associated with each data value (for example, the midpoint of each edge), it would go into the coordinates part of the lattice.
For more on curvilinear lattices, see **"Curvilinear Lattices"**.

The Pyramid Connections

The connections among elements within a layer can be fairly complex and each one must be specifically defined in the *cxConnection* structure. This is the type definition for the set of connections in a pyramid layer:

```
shared typedef struct {
  long         numElements                 "Num Elements";
  long         elemArrayLen                "Elements Array Length";
  long         numConnections              "Connections Array Length";
  long         elements[elemArrayLen]      "Elements Array";
  long         connections[numConnections] "Connections Array";
} cxConnection;
```

These are the variables:

numElements	An integer that defines the number of elements, such as faces or lines, in this layer of the hierarchy.
elemArrayLen	The dimensioning variable for the elements[] array. It defines the length of the elements array, which will always be one more than the number of elements, `numElements` (see below).
numConnections	Defines the length of the connections array.
elements[]	The vector of connections between each element in this layer and elements in the next lower layer. The counts are listed cumulatively, so that `elements[i+1]` is `elements[i]` plus the number of connections from element number `i` (where `i` >= 0 in the C language). Thus, the final entry in `elements` is equal to the total number of connections from elements in this layer. Also, it can now be seen that `elements` must always contain one more entry than the number of elements, since the final entry is used to determine the number of connections for element number `numElements`.
connections[]	The vector of dependencies of current layer elements on elements in the layer below this one. The connections array lists the indices of elements in the layer beneath that are to be used in constructing the current layer. The entry at `elements[i]` in `connections[]` is the first subordinate item making up the `i`th element, with the others listed consecutively. The value at `elements[i]` is the index into the lattice at the next lower layer of the subordinate element (or into the base lattice if the current layer is numbered 0 in C).

Use of these variables is illustrated in **"Creating a Connection List"**, below.

4.4.2. Connection List Components

Figure 4–6 illustrates the components of a connection list for layer 0 of a tetrahedron. It contains six edge elements connected to four nodes in the base lattice by twelve dependencies. The connection data structure defines each edge–vertex relationship specifically (see **"The Pyramid Connections"**).

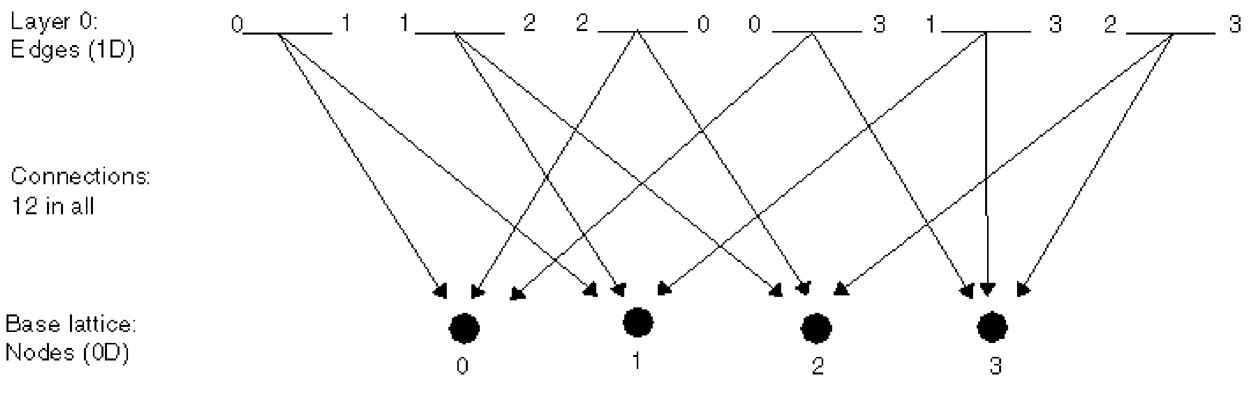


Figure 4–6 Connections in a Tetrahedron

You can see how the vertex information and the description of the connections between them at each level are used to construct 2−D and 3−D elements, resulting in the creation of a tetrahedron. It is, however, legal to create a finite element pyramid without base data, coordinates, or indeed a base lattice at all.

4.4.3. Creating a Connection List

This example shows how you fill out the connections list for each layer in a pyramid. The data are for the connections illustrated in **Figure 4−6**.

1. Use the API subroutine *cxConnNew* to create a new *cxConnection* structure, specifying the total number of elements in the current layer of the pyramid and the total number of connections between elements in this layer and nodes in the layer below. In this example, `numElements` is 6 and `numConnections` is 12. `elemArrayLen` equals one plus the number of elements (i.e., 7).
2. Get pointers to the `elements` and `connections` array from the new *cxConnection* structure using the API call *cxConnPtrGet*.
3. Define the connections between each element and the nodes, listed in increasing order from 0. Do this by listing the node indices in order (the order need not be sorted by node number). For example, in **Figure 4−6**:
   - Element 0 has two nodes and points to nodes 0 and 1.
   - Element 1 has two nodes and points to nodes 1 and 2.
   - Element 2 has two nodes and points to nodes 2 and 0.
   - Element 3 has two nodes and points to nodes 0 and 3.
   - Element 4 has two nodes and points to nodes 1 and 3.
   - Element 5 has two nodes and points to nodes 2 and 3.
4. Set the `connections` array to be the concatenation of the node indices associated with each element, as determined in step 3. In this case, the array is [0 1 1 2 2 0 0 3 1 3 2 3].
5. Set the *elements* array to index into the connections array so that `elements[i+1]` is the cumulative number of connections for elements number 0, 1, ... *i*. Element *i*, consequently, has `elements[i+1] - elements[i]` connections, starting at `connections[elements[i]]`, and ending at `connections[elements[i+1]-1]`. In the present example, this is straightforward to construct, since each element points to the same number (2) of nodes. So, `elements` is [0 2 4 6 8 10 12]. (More generally, if each element has `N` connections, then we have `elements[i] = i*N`, for i >= 0.)

4.4.4. Using the Compressed Pyramid API

It is somewhat involved to write a module that takes a pyramid as input in order to perform a computation, create a filtered output pyramid, or to generate geometry for visualization in the *Render* module. You must take into account the hierarchical nature of the pyramid data, with the added option of a dictionary to indicate element types.

Designing a Compressed Pyramid Module

Because a few concepts usually suffice to create a compressed pyramid manipulation module, the steps to create such a module are given here in outline form.

The first step is to discover the natural dimension at which the module works. For example, an isosurface module typically works with a 3−D input in order to create the isosurface through a single cell. If the input is viewed as a collection of 3−D cells, the output is a collection of 2−D surfaces. The behavior of the module is characterized by the dimensionality of input that it naturally processes. Generally, the input must have at least this dimensionality (you cannot isosurface a 2−D input and get many useful results), which translates into a requirement for at least this number of layers in its pyramid.

The second step is to develop the computational algorithm for the module assuming the presence of data at the desired level. If the module requires the full hierarchical information of vertices, edges, and faces, this is the place to take advantage of it; however, if the module can work by omitting some layers of data (for instance, by dealing with a face as a list of vertices) then you should assume that those layers are omitted and that the pyramid is compressed at the operational layer.

Next, assuming that the pyramid is compressed at the operational layer, or not at all, loop over all "active" elements. The active elements are those that are subordinate to some element at the top layer of the pyramid and are thus reachable from the top. Two API routines exist to list the active elements at a given layer: *cxPyrActive* and *cxPyrActiveList*. The former returns a Boolean byte vector (0 for false, otherwise true) indicating whether the element is active; the latter returns an array of the active indices. Use *cxPyrActive* unless you suspect that most elements in your input pyramid will be inactive. Using the active indicators, loop over all active elements and process them.

Dealing with Compressed Elements

If the pyramid is not compressed, but you wish to deal with compressed elements, you can create a default dictionary using *cxPyrDictDefault*, using the dimensionality of the operational layer as input. Then use *cxPyrDictLookup* to find the pyramid dictionary element that is appropriate for the fully expanded element you have. This routine returns the list of vertices that make up the element so that you can work directly with the vertex list (assuming that your algorithm is equipped to handle the type of element returned by the call to *cxPyrDictLookup*).

Using *cxPyrDictLookup* may be time−consuming. In effect, you are compressing your input at the operational layer; you may simply call *cxPyrCompress* if you wish to do this all at once, assuming that you have the storage necessary to hold both the expanded and compressed versions of the pyramid at one time.

Handling Compression at Non−operational Levels

At this point you have a module that will work with compressed or expanded inputs, assuming compression only at the operational layer. Now you must extend your module to handle inputs that are compressed at the wrong layer.

It is often sufficient to handle inputs that are compressed at too low a layer by simply expanding them, then dealing with the fully expanded input as above. The savings due to compression increase with the dimensionality of the compression (or conversely, decrease with decreasing dimensionality of compression), so a pyramid that has too low a compression level may not save much storage. If the cost of expanding an input is likely to prove too great, then you must extend your computational algorithm to handle compression at too low a level.

Handling pyramid compression at too high a level is relatively easy, but requires a little study to master the details. The concept is to represent each pyramid dictionary reference element at the too−high level as a collection of known dictionary reference elements at the operational layer, and then to process the input by processing the equivalent known elements.

A Practical Example

Assume that you are writing the 2−D part of *PyrToGeom*, which creates shaded polygons from an input pyramid. However, the input pyramid is compressed at the 3−D cell level. It is prohibitively expensive in time and memory to expand the pyramid and then recompress it to faces, so instead you preprocess the 3−D element dictionary in terms of its faces, then simply draw the faces for each 3−D element.

The first step is to preprocess the pyramid dictionary. The routine *cxPyrDictCompress* simplifies this task. This routine takes an input dictionary compressed at too high a level and returns a new, equivalent dictionary, where each reference element in the input dictionary is replaced by a `_compressed_` reference element in the output dictionary.

This violates the usual standard that a pyramid dictionary is made up of fully expanded pyramids, but the violation is temporary, as this output dictionary is really used as a translation table to process elements at the operational level. **Figure 4−7** shows the process.

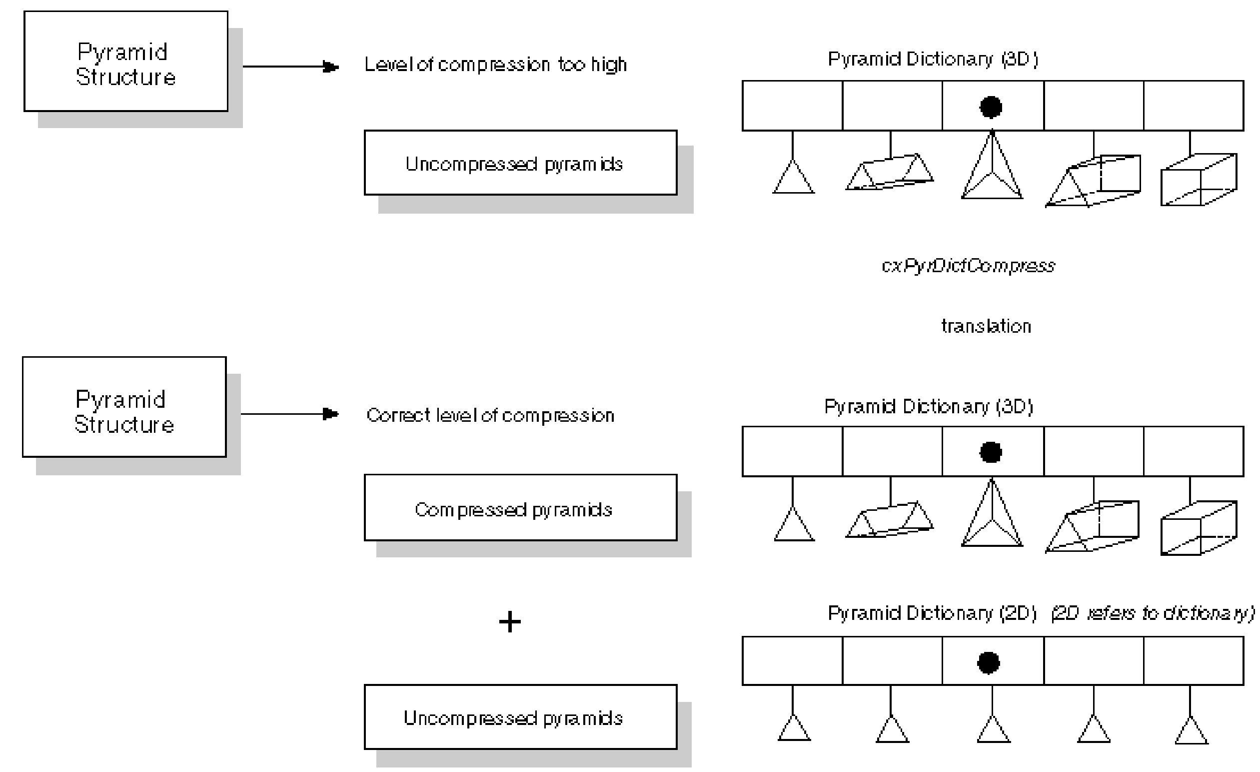


Figure 4–7 Preprocessing a Pyramid Dictionary

Each of the compressed reference elements now refers to the same dictionary, which has dimension equal to the operational level. This means that each reference element is now described in terms of elements that your algorithm can handle. Furthermore, all the reference elements in the input dictionary are described in terms of the same dictionary, which you can include in your output if necessary.

Relating this to the *PyrToGeom* example, you receive a grid of 3–D elements but want to work with faces. So you call *cxPyrDictCompress* to get a 3–D dictionary in which each 3–D reference element is instead a pyramid compressed at the 2–D level – that is, each 3–D reference element is now a compressed pyramid made up of faces.

Now it is easy to process the too highly compressed input. Simply operate on each active element at the compression level, looking up the element in the translation dictionary.

The element will comprise one or more elements at the correct level of compression, which can be processed directly by your algorithm. For instance, a 3–D prism cell input is made up of triangular and rectangular faces, which *PyrToGeom* can handle.

Lastly, keep track of the vertices that are used to represent each element. In the original input, the vertices are listed at the compression layer in the `connections[]` array of the *cxConnection* structure. The input dictionary uses vertices numbered 0, 1, ... *N*–1 to indicate the first *N* vertices of the compressed element. During recompression of the input dictionary, the higher level element is decomposed into lower level elements, and the relevant vertices from the set [0, 1, ..., *N*–1] are used.

It is important to pick up the indirect indexing from both the compressed dictionary and the input connection list, so that the correct vertices are used. For example, the prism has six vertices. If vertices 1, 2, 3, 6, 7 and 8 make up a sample prism, then the prism element in the input dictionary refers to vertices 0–5, meaning the first six vertices of the input element. Now, if you work with vertices 0, 1, 2, 3, 4 and 5, you will be interpreting the input data incorrectly. Instead, use the indirect addressing to pick up 0==>1, 1==>2, 2==>3, 3==>6, 4==>7, 5==>8.

The prism will then be made up of faces of two types, triangles (0,1,2) and rectangles (0,1,2,3). A sample `connection` array for a prism is [0 1 2 0 1 4 3 1 2 5 4 2 0 3 5 3 4 5]. When you work with the last triangle in the prism, make sure to index from its vertex number (0,1,2) into the connection array to get (3,4,5) as the vertices to look up in the original compressed input, yielding actual vertex numbers 6,7,8. This double translation may seem complex, but it naturally reflects the compression in two stages, first to too high a level, then down to the correct level.

Correct application of this technique makes it almost as easy to handle pyramid compression at any level as it is to handle it only at the desired level. Source code for the modules *ComposePyr*, *CropPyr*, and *LatToPyr* is provided in the directories below */usr/explorer/src*.

Most modules operate at a single level, but some, like *PyrToGeom*, can operate at many levels. In this case, you can create a table of operational level versus level of compression. You will usually find that the table breaks down into actions at the operational level, actions below that level (including, perhaps, simply expanding the input data), and recompressing the input dictionary for inputs above the operational level.

4.5. Chemistry Pyramids

Chemistry pyramids are used to construct objects according to information pertaining to molecular structures. The pyramid structure is more narrowly defined than that of the finite element pyramid and differs from it as follows:

- The 3–D layer, which defines the complete molecule, is not a volume but a ball–and–stick construction.
- The relationship between bonds (in the base lattice) and atoms (in the 0 layer) is not hierarchical, although the relationship of 0–D to 1–D layer is shown as such.
- The structure of the lattice at each level is closely defined.

The chemistry pyramid has four levels, which are:

layer[2] — The top layer, which defines a complete molecule.

layer[1] — Contains information about atomic residues. The lattice at this level is a 1–D uniform lattice (four vector, byte) that contains the four–character residue name.

layer[0] — Contains all atomic information in a 1–D curvilinear lattice of type float with *nCoordVar* = 3, and *nDataVar* = 4, or more. The coordinate part of the lattice contains the positions of the atoms in 3–D space. The data part of the lattice contains an ID number for the atom (referenced in the bonding information – see below) together with various other atomic parameters (see **Table 4–2**).

baseLattice — Defines the atomic bonds in a 1–D uniform lattice of type long with *nDataVar* = 3, or more. The length of the lattice (i.e., *dims[0]*, the number of nodes) is equal to the number of bonds in the molecule. Each node contains contains the bond order (such as single, double, or triple) and two atom IDs. The IDs are the same as those specified in the atomic lattice. Subsequent variables in the data vector may be used to store other information about the bond (such as its topology, for example – see **Table 4–2**).

Source code examples, in the form of the source code used to build the modules *MoleculeBuilder* and *ReadPDB*, are provided in the directories below */usr/explorer/src*.

Table 4–2 lists the minimum required set of data values for the base lattice and the layer 0 lattice in a chemistry pyramid.

Base Lattice Values	**Layer 0 Lattice Values**
Type: 1 = single, 2 = double, 3 = triple, 4 = H–bond, 5 = aromatic	ID number
First atomic number of the bond	Element number (from the Periodic Table of the Elements)
Second atomic number of the bond	Van der Waals radius
Topology: 1 = in ring, 2 = chain	Color index
	Covalent radius
	Mass
	Charge

Valence
Radical
Ionization
Hybridization

Table 4–2 User–defined Values in the Chemistry Pyramid

The connection list from atoms to bonds shows which bonds relate to which atoms, and the bond lattice's atomic IDs define which atoms are connected by which bonds. You can supply more information in additional vector components if you choose.

4.6. Data Type Declaration

The *cxPyramid* data type, though defined in the IRIS Explorer typing language, can be considered as a C structure. Fortran users need to set pointers to the data type structures when they use them. The type declaration resides in the header file */usr/explorer/include/cx/cxPyramid.h.*

This is the data type declaration for *cxPyramid*:

```
#include <cx/cxLattice.h>

typedef enum {
  cx_compress_none,
  cx_compress_unique,
  cx_compress_multiple
} cxCompressType;

typedef struct cxConnection {
  long               numElements;
  long               elemArrayLen;
  long               numConnections;
  long              *elements;
  long              *connections;
} cxConnection;

typedef struct cxPyramidLayer {
  cxConnection      *relation;
  cxLattice         *nextLattice;
} cxPyramidLayer;

typedef struct cxPyramidDictionary {
  long               nDim;
  long               numEntries;
  struct cxPyramid **table;
} cxPyramidDictionary;

typedef struct cxPyramidReference {
  cxCompressType    compressType;
  union {
    struct {
      long              numIndices;
      long             *indices;
    } cx_compress_multiple;
    struct {
```

```
      long              index;
    } cx_compress_unique;
  } r;
  cxPyramidDictionary *dictionary;
} cxPyramidReference;

typedef struct cxPyramid {
  cxLattice         *baseLattice;
  long               count;
  cxPyramidLayer    *layer;
  cxPyramidReference ref;
} cxPyramid;
```

The case of *cx_compress_none* does not appear in the union because it is empty. It is illegal to use an empty case in the IRIS Explorer data typing language.

The Fortran type enumerations reside in the file */usr/explorer/include/cx/cxPyramid.inc.*

The compression types are specified by this enumeration:

```
integer cx_compress_none
integer cx_compress_unique
integer cx_compress_multiple

parameter (cx_compress_none = 0)
parameter (cx_compress_unique = 1)
parameter (cx_compress_multiple = 2)
```

4.7. The Pyramid API Routines

You can use the API (Application Programming Interface) routines to manipulate data types in IRIS Explorer. The subroutines for creating and manipulating pyramids in IRIS Explorer, and for creating and referencing a pyramid dictionary, are listed below and described in detail in the *IRIS Explorer Reference Pages* and the on−line *man* pages.

The subroutines let you:

- create pyramids
- set the values in the pyramid and connection list data structures
- allocate and copy pyramid data types
- extract pyramid values
- create a pyramid dictionary
- reference a pyramid dictionary

Pyramid allocation is based on the number of levels of the pyramid; the user must allocate the lattice and connections for each level (use *cxLatNew* and *cxConnNew*). By convention, a pyramid with two lattices and one connection has one level (*count* = 1) because the base lattice is not counted as a level.

Note: All the subroutines use zero−based indexing except *cxPyrLayerGet* and *cxPyrLayerSet.* These two subroutines use 1−based indexing.

Table 4−3 lists the pyramid subroutines.

Subroutine	Purpose
cxPyrNew	Allocates a pyramid structure
cxConnNew	Allocates a connection list structure
cxLatNew	Allocates a lattice structure
cxPyrSet	Replaces the base lattice in a pyramid

cxPyrLayerGet	Returns the lattice and connection list from a layer of the pyramid
cxPyrLayerSet	Replaces the lattice and connection list from a layer of the pyramid
cxPyrDup	Makes a copy of a pyramid structure
cxConnDup	Copies a connection list structure
cxPyrGet	Returns a pointer to the base lattice and the number of layers from a pyramid
cxConnEleGet	Returns the connections of an element from a connection list
cxConnEleSet	Sets the connections of an element in a connection list
cxPyrClean	Removes unreferenced items from a pyramid
cxPyrMerge	Selectively merges duplicated elements in a layer
cxPyrLayerSkip	Returns a connection list relating two non−adjacent pyramid layers
cxConnPtrGet	Returns all contents of a connection list
cxConnPtrSet	Sets all contents of a connection list

Table 4−3 Pyramid Subroutines

Table 4−4 lists the pyramid dictionary subroutines.

Subroutine	Purpose
cxPyramidDictionary	Creates a new a pyramid dictionary
cxPyrDictDefault	Calls the default dictionary
cxDictUniqNew	Creates a new a dictionary containing only one kind of element
cxDictMultiNew	Creates a new a dictionary containing more than one kind of element
cxDictElemDefault	Returns the complete default IRIS Explorer dictionary of finite elements for the given level
cxPyrDictGet	Gets the pyramid dictionary
cxPyrDictSet	Replaces the dictionary in a pyramid
cxDictDescGet	Provides the information required to create a new dictionary structure
cxPyrCompress	Converts an expanded pyramid to a compressed pyramid
cxPyrDupExpand	Duplicates a pyramid and puts it in expanded form
cxPyrEleDupExpand	Converts an element of a compressed pyramid into an uncompressed element in its own pyramid
cxPyrConcat	Concatenates one pyramid onto another and expands or merges any dictionaries they may have
cxPyrEleNormalForm	Removes duplicated nodes and edges, and fixes the faces that contain removed edges
cxPyrEleBrickForm	Expands each element to 8−node form, adding duplicated nodes
cxDictIndexGet	Finds the "index" of a specified dictionary element
cxDictIndexVGet	Copies a vector of indices for the entire pyramid into "indices"
cxPyrSize	Returns the number of bytes used to store the pyramid

Table 4−4 Pyramid Dictionary Subroutines

4.8. Code Examples

Here are some examples of user functions for pyramid modules.

4.8.1. A Simple Example

This example uses compressed pyramids to generate a pyramid. The base lattice is passed into the module via its input port, and it reads the pyramid information from an input file. The C source code is in */usr/explorer/src/MWGcode/Pyramid/C/GeneratePyr.c* and the Fortran source

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

is in */usr/explorer/src/MWGcode/Pyramid/Fortran/GeneratePyr.f*. The directories also contain copies of an example input data file, *pyr.dat* (see below), and an example map *GeneratePyr.map*. You can test the module by building it, loading the example map into the Map Editor, and viewing the vectors in the pyramid, then by changing the map and data files (see below).

This is the *pyr.dat* file, which contains information about the pyramid to be generated. The reading of this file (see the source code below) allows for blank lines, but no other text (such as comments, for example).

```
6
1 3 5 7 9 11
4
0 1 2 5
8
2 4 6 8 10 12 14 16
```

The format of this file is the number of nodes (vertices) in the first element, followed by a list of the vertices in the first element, followed by the number of nodes in the second element, etc. The data values and coordinates of the vertices are taken from the base lattice; the numbering of the vertices is the same as that of the nodes in the lattice. There is a direct equivalence between this numbering and that for the connections. In this example, it can be seen that the pyramid is made up of a prism element, a tetrahedron element and a brick element.

The example *GeneratePyr.map* uses *GenLat* as the source of the base lattice. As an alternative, you could try reading this in from a file. For example, here is a file containing coordinates and data for a set of vertices:

```
17

0.0 0.0 0.0 1
1.0 0.0 0.0 2
0.0 1.0 0.0 2
0.0 0.0 1.0 2

2.0 2.0 2.0 2
2.0 2.5 2.0 2
2.5 2.5 2.0 2
2.5 2.0 2.0 2
2.0 2.0 2.8 4
2.0 2.5 2.8 4
2.5 2.5 2.8 4
2.5 2.0 2.8 4

3.0 3.0 4.0 2
3.0 4.0 4.0 2
4.0 4.0 4.0 2
4.0 3.0 4.0 2
3.5 3.5 3.0 1
```

You can read this file using the *ReadXYZData* module, which outputs a 1−D curvilinear lattice. Using this module, the first three values on each line (apart from the first) are read as the (x, y, z) coordinates of the node, and the fourth value is read as the data value at that node. Connect its output to the *BaseLat* input port on *GeneratePyr*

Footnote: The generic module name *GeneratePyr* here stands for *GeneratePyr_C* or *GeneratePyr_F*, the actual name of the modules built in C or Fortran.

. Then pass the following input file to *GeneratePyr*:

```
4
0 1 2 3
8
4 5 6 7 8 9 10 11
5
12 13 14 15 16
```

In this example, the pyramid is composed of a tetrahedral element, a hexahedral element and a pyramidal element (this division is reflected by the breaks in the lattice input file, above).

C Version:

```
#include <cx/cxLattice.api.h>
#include <cx/cxParameter.api.h>
#include <cx/cxPyramid.api.h>
#include <cx/Pyramid.h>
#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/UI.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define NUM_TYPES 9
#define NODE(i) ((i==0)?4:i)
long pyr_type[NUM_TYPES] = {
 cx_pyramid_dict_quadrilateral,
 cx_pyramid_dict_point,
 cx_pyramid_dict_line,
 cx_pyramid_dict_triangle,
 cx_pyramid_dict_tetrahedron,
 cx_pyramid_dict_pyramid,
 cx_pyramid_dict_prism,
 cx_pyramid_dict_wedge,
 cx_pyramid_dict_hexahedron,
};
int memclean(cxPyramid *pyr, cxErrorCode ec, cxConnection *conn)
{
  if(cxDataAllocErrorGet() || (ec != cx_err_none))
    {
      if(pyr != NULL)cxDataRefDec(pyr);
      if(conn != NULL)cxDataRefDec(conn);
      return 1;
    }
  else
    {
      return 0;
    }
}

void compose(cxLattice *baseLat, unsigned char *filename,
      cxPyramid **pyr, long quad)
{
```

```
 cxPyramidDictionary *dict;
 cxErrorCode ec = cx_err_none;
 cxConnection *conn;
 FILE *file;
 long *obj = NULL;
 long *node = NULL;
 long *iptr,i,num_nodes,num_objs,unique,levels;
 long show_quad,*elem,*connect;
 char buf[100];

 if(!filename || filename[0] == NULL)
   return;
 if((file = fopen(filename,"r")) == NULL)
   {
     printf(" Error reading file %s\n",filename);
     return;
   }

 /* the object list will not be any larger than the number of nodes */
 obj = (long *)calloc(baseLat->dims[0],sizeof(long));
 show_quad = FALSE;
 num_objs = num_nodes = 0;
 while(TRUE)
   {
     if(fscanf(file,"%d",&obj[num_objs]) < 0) break;
     if(node == NULL)
{
  node = (long *)malloc(obj[num_objs]*sizeof(long));
}
     else
{
  node = (long *) realloc
    ((void *)node, (num_nodes+obj[num_objs])*sizeof(long));
}
     for(i=0;i<obj[num_objs];i++)
{
  if(fscanf(file,"%d",&node[num_nodes+i]) < 1)
    {
      printf
 (" Error reading obj %d. Not enough data\n",num_objs);
      return;
    }
}
     num_nodes += obj[num_objs];

     /* determine whether it it a quad or tetra */
     if(obj[num_objs] == 4) {
obj[num_objs] = ((quad == 0) ? 0:4);
show_quad = TRUE;
     }
     num_objs++;
   }

 /* do I need the radio button? */
```

```
 if(show_quad)
   cxInWdgtShow("Quad");
 else
   cxInWdgtHide("Quad");

 /* am I a unique compressed pyramid ? */
 unique = TRUE;
 for(i=1;i<num_objs;i++)
   {
     if(obj[i] != obj[i-1])
{
  unique = FALSE;
  break;
}
   }

 /* build the pyramid */
 *pyr = cxPyrNew(0);
 cxPyrSet(*pyr,baseLat);
 levels = 3;
 conn = cxConnNew(num_objs,num_nodes);
 if(memclean(*pyr,ec,conn)) return;
 cxConnPtrGet(conn,NULL,NULL,&elem,&connect);

 /* fill elem & connect */
 elem[0] = 0;
 for(i=0;i<num_objs;i++)
 {
   elem[i+1] = elem[i] + NODE(obj[i]);
 }
 bcopy(node,connect,num_nodes*sizeof(long));

 /* verify all the nodes are legal */
 for(i=0;i<num_nodes;i++)
 {
   if(node[i] < 0 || node[i] >= baseLat->dims[0])
   {
     printf("Illegal connection %d at %d\n",node[i],i);
     *pyr = NULL;
     return;
   }
 }

 /* load all the intermediate levels of the pyramid */
 for(i=1;i<levels;i++)
 {
   cxPyrLayerSet(*pyr,i,NULL,NULL);
 }

 /* put the connectivity in the pyr */
 cxPyrLayerSet(*pyr,levels,conn,NULL);
 if(memclean(*pyr,ec,conn))
   return;
```

```
  /* add the compression info */
  dict = cxPyrDictDefault(levels);
  cxPyramidDictionarySet(*pyr,dict,&ec);
  if(memclean(*pyr,ec,NULL))
    return;
  if(unique)
  {
    cxPyramidCompressionTypeSet(*pyr,cx_compress_unique,&ec);
    if(memclean(*pyr,ec,conn))
      return;
    iptr =  cxPyramidCompressionIndexGet(*pyr,&ec);
    if(memclean(*pyr,ec,NULL))return;
    *iptr = pyr_type[obj[0]];
  }
  else
    {
      /* non-unique */
      cxPyramidCompressionTypeSet(*pyr,cx_compress_multiple, &ec);
      if(memclean(*pyr,ec,NULL)) return;
      cxPyramidNumberIndicesSet(*pyr,num_objs,&ec);
      iptr = (long *)cxPyramidCompressionIndexAlloc(*pyr);
      if(iptr != NULL)
 {
   cxPyramidCompressionIndexSet(*pyr,iptr,&ec);
   if(memclean(*pyr,ec,NULL))return;
   for(i=0;i<num_objs;i++)
     {
      *iptr++ = pyr_type[obj[i]];
     }
 }
    }
  free(obj);
  free(node);
}
```

Fortran Version:

```
      SUBROUTINE COMPOSE(BASLAT,FILNAM,PYR,QUAD)
C
      INCLUDE '/usr/explorer/include/cx/DataOps.inc'
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
      INCLUDE '/usr/explorer/include/cx/DataTypes.inc'
      INCLUDE '/usr/explorer/include/cx/Pyramid.inc'
      INCLUDE '/usr/explorer/include/cx/cxPyramid.api.inc'
C
C     .. Parameters ..
      INTEGER           LSIZE
      PARAMETER         (LSIZE=4)
C     .. Scalar Arguments ..
      INTEGER           BASLAT, PYR, QUAD
      CHARACTER*(*)     FILNAM
C     .. Local Scalars ..
      INTEGER           CONN, DICT, EC, I, I1, I2, I3, I4, I5, I6, I7,
     *                  IER, IOS, IUNIT, J, NUMNOD, NUMOBJ
```

```
      LOGICAL           SHOWQU, UNIQUE
C     .. Local Arrays ..
      INTEGER           CONCT(1), DIMS(1), ELEM(1), NODE(1), OBJ(1),
     *                  PYRTYP(9)
C     .. External Functions ..
      INTEGER           INODE
      EXTERNAL          CXCONNNEW, CXCONNPTRGET, CXDATAMALLOC,
     *                  CXLATDESCGET, CXPYRAMIDCOMPRESSIONINDEXALLOC,
     *                  CXPYRAMIDCOMPRESSIONINDEXGET, CXPYRDICTDEFAULT,
     *                  CXPYRLAYERSET, CXPYRNEW, CXPYRSET, INODE
C     .. External Subroutines ..
      EXTERNAL          CXINWDGTHIDE, CXINWDGTSHOW,
     *                  CXPYRAMIDCOMPRESSIONINDEXSET,
     *                  CXPYRAMIDCOMPRESSIONTYPESET,
     *                  CXPYRAMIDDICTIONARYSET,
     *                  CXPYRAMIDNUMBERINDICESSET, CXDATAFREE, REALLC
C     .. Data statements ..
      DATA              PYRTYP/CX_PYRAMID_DICT_QUADRILATERAL,
     *                  CX_PYRAMID_DICT_POINT, CX_PYRAMID_DICT_LINE,
     *                  CX_PYRAMID_DICT_TRIANGLE,
     *                  CX_PYRAMID_DICT_TETRAHEDRON,
     *                  CX_PYRAMID_DICT_PYRAMID, CX_PYRAMID_DICT_PRISM,
     *                  CX_PYRAMID_DICT_WEDGE,
     *                  CX_PYRAMID_DICT_HEXAHEDRON/
      DATA              IUNIT/11/
C     .. Pointers to Lattice Structures ..
      POINTER (POBJ,OBJ), (PNODE,NODE), (PDIMS,DIMS)
      POINTER (PELEM,ELEM), (PCONCT,CONCT), (PNOD,NOD)
C     .. Executable Statements ..
C
C     Open the input file for reading
C
      OPEN (IUNIT,FILE=FILNAM,STATUS='OLD',IOSTAT=IOS)
      IF (IOS.NE.0) THEN
         PRINT *, 'Error opening file ', FILNAM
         RETURN
      END IF
C
C     Get the input lattice description
C     Allocate space to hold the associated object list
C
      IER = CXLATDESCGET(BASLAT,I1,PDIMS,I2,I3,I4,I5,I6,I7)
      POBJ = CXDATAMALLOC(DIMS(1)*LSIZE)
C
C     Initialise some variables
C
      SHOWQU = .FALSE.
      NUMOBJ = 0
      NUMNOD = 0
C
C     Read the object and nodes until the end of the file is reached
C     Allocate space to hold the nodes
C
   20 READ (IUNIT,FMT=*,END=40) OBJ(NUMOBJ+1)
```

```
      IF (NUMNOD.EQ.0) THEN
         PNODE = CXDATAMALLOC(OBJ(NUMOBJ+1)*LSIZE)
      ELSE
         PNOD = PNODE
         CALL REALLC(PNODE,PNOD,(NUMNOD+OBJ(NUMOBJ+1))*LSIZE,
     *               NUMNOD*LSIZE)
      END IF
      READ (IUNIT,FMT=*,END=40) (NODE(J),J=NUMNOD+1,
     *  NUMNOD+OBJ(NUMOBJ+1))
      NUMNOD = NUMNOD + OBJ(NUMOBJ+1)
C
C     Determine if a quad has been entered
C
      IF (OBJ(NUMOBJ+1).EQ.4) THEN
         IF (QUAD.EQ.0) THEN
            OBJ(NUMOBJ+1) = 0
         ELSE
            OBJ(NUMOBJ+1) = 4
         END IF
         SHOWQU = .TRUE.
      END IF
      NUMOBJ = NUMOBJ + 1
      GO TO 20
   40 CONTINUE
      CLOSE (IUNIT)
C
C
C     Set the quad widget
C
      IF (SHOWQU) THEN
         CALL CXINWDGTSHOW('Quad')
      ELSE
         CALL CXINWDGTHIDE('Quad')
      END IF
      UNIQUE = .TRUE.
      DO 60 I = 2, NUMOBJ
         IF (OBJ(I).NE.OBJ(I-1)) UNIQUE = .FALSE.
   60 CONTINUE
C
C     Build the pyramid structure
C
      PYR = CXPYRNEW(0)
      IER = CXPYRSET(PYR,BASLAT)
      CONN = CXCONNNEW(NUMOBJ,NUMNOD)
      IER = CXCONNPTRGET(CONN,I1,I2,PELEM,PCONCT)
      ELEM(1) = 0
      DO 80 I = 1, NUMOBJ
         ELEM(I+1) = ELEM(I) + INODE(OBJ(I))
   80 CONTINUE
      DO 100 I = 1, NUMNOD
         CONCT(I) = NODE(I)
  100 CONTINUE
      DO 120 I = 1, NUMNOD
         IF (NODE(I).LT.0 .OR. NODE(I).GT.DIMS(1)) THEN
```

```
            PRINT *, 'Illegal connection ', NODE(I), ' at ', I
            PYR = 0
            RETURN
         END IF
  120 CONTINUE
      IER = CXPYRSET(PYR,BASLAT)
      IER = CXPYRLAYERSET(PYR,1,0,0)
      IER = CXPYRLAYERSET(PYR,2,0,0)
      IER = CXPYRLAYERSET(PYR,3,CONN,0)
      DICT = CXPYRDICTDEFAULT(3)
      CALL CXPYRAMIDDICTIONARYSET(PYR,DICT,EC)
      IF (UNIQUE) THEN
         CALL CXPYRAMIDCOMPRESSIONTYPESET(PYR,CX_COMPRESS_UNIQUE,EC)
         PCONCT = CXPYRAMIDCOMPRESSIONINDEXGET(PYR,EC)
         CONCT(1) = PYRTYP(OBJ(1))
      ELSE
         CALL CXPYRAMIDCOMPRESSIONTYPESET(PYR,CX_COMPRESS_MULTIPLE,EC)
         CALL CXPYRAMIDNUMBERINDICESSET(PYR,NUMOBJ,EC)
         PCONCT = CXPYRAMIDCOMPRESSIONINDEXALLOC(PYR)
         CALL CXPYRAMIDCOMPRESSIONINDEXSET(PYR,CONCT,EC)
         IF (PCONCT.NE.0) THEN
            DO 140 I = 1, NUMOBJ
               CONCT(I) = PYRTYP(OBJ(I)+1)
  140       CONTINUE
         END IF
      END IF
C
C     Free the allocated space for objects and nodes
C
      CALL CXDATAFREE(POBJ)
      CALL CXDATAFREE(PNODE)
C
      RETURN
      END
      INTEGER FUNCTION INODE(I)
C     .. Scalar Arguments ..
      INTEGER                I
C     .. Executable Statements ..
C
      INODE = I
      IF (I.EQ.0) INODE = 4
C
      RETURN
      END
      SUBROUTINE REALLC(PADDR,PIADDR,NEWSIZ,OLDSIZ)
C
C     Reallocate space for the nodes list as it grows
C
      INCLUDE '/usr/explorer/include/cx/DataOps.inc'
C
C     .. Scalar Arguments ..
      INTEGER            NEWSIZ, OLDSIZ
C     .. Local Scalars ..
      INTEGER           I
```

```
C     .. Local Arrays ..
      CHARACTER         ADDR(1), IADDR(1)
C     .. External Subroutines ..
      EXTERNAL          CXDATAFREE
C     .. External Functions ..
      EXTERNAL          CXDATAMALLOC
C     .. Pointers to Lattice Structures ..
      POINTER(PADDR,ADDR), (PIADDR,IADDR)
C     .. Executable Statements ..
C
      PADDR = CXDATAMALLOC(NEWSIZ)
      DO 20 I = 1, OLDSIZ
         ADDR(I) = IADDR(I)
   20 CONTINUE
      CALL CXDATAFREE(PIADDR)
C
      RETURN
      END
```

4.8.2. More Complex Examples

Source code for the modules *ComposePyr* (which is a more complicated variant on the *GeneratePyr* module above), *CropPyr*, and *LatToPyr* is provided in the directories below */usr/explorer/src*. To see how the modules are constructed, open the *modulename.mres* files for these modules in the Module Builder.

Chapter 5

Using the Geometry Data Type

This chapter describes the geometry data type and how it is used in the IRIS Explorer environment. It defines the geometry data type and explains how geometry is created in IRIS Explorer. It also describes the interface between IRIS Explorer and Open Inventor™, a 3−D graphics development tool, and explains the role of the Open Inventor scene graph in creating more complex geometry.

The API (Application Programming Interface) routines are listed, and examples of user function code for writing modules that produce or manipulate geometry are included.

5.1. Understanding Geometry

The *cxGeometry* data type is a root data type, which means it can be specified as a data type on a port and will pass data between modules. It enables IRIS Explorer to handle geometry objects by providing a container for Open Inventor scene graph specifications. Open Inventor is an object−oriented graphics library of objects and methods used to create interactive 3−D graphics. The IRIS Explorer *Render* module and other geometry modules draw on Open Inventor technology. The geometry objects that IRIS Explorer uses to visualize numerical data are created by connecting nodes to form an Open Inventor scene graph. The scene graph constitutes the IRIS Explorer geometry object.

IRIS Explorer allows you to manipulate Open Inventor nodes to form geometry objects through the geometry subroutines in the API. You can output most of the geometry you need in IRIS Explorer by using the Geometry Library, which creates Open Inventor objects, in this way. To create geometry in IRIS Explorer, you need not have Open Inventor installed on your system.

To build a module that reads geometry, that is, that accepts the *cxGeometry* data type on its input port, you need direct access to Open Inventor and its libraries. You also need Open Inventor if you want to obtain more complex rendering effects than IRIS Explorer can provide. To write a scene graph with direct reference to the Open Inventor library, you must have Open Inventor installed on your system.

You can use Open Inventor with C and C++, but not with Fortran. If you are programming in Fortran, you must use the IRIS Explorer API subroutines for all your geometry needs.

This chapter points you toward Open Inventor for creating complex graphical effects that are outside the scope of IRIS Explorer, but it does not tell you how to use Open Inventor. For more information, refer to *The Inventor Mentor: Programming Object−Oriented 3D Graphics with Open Inventor, Release 2* (Addison Wesley, 1994). For information on obtaining Open Inventor, contact one of the IRIS Explorer Centers.

5.2. The Geometry Data Type

The *cxGeometry* data type is a simple data structure that transcribes an Open Inventor scene graph into a linear data stream called the *delta protocol*, encoded in an Open Inventor binary file format. It is opaque to IRIS Explorer users. The scene graph can then be moved between modules that are capable of processing it. A scene graph is a hierarchical geometrical construction made of *nodes* that contain specific details about the scene graph, or geometry object. "Scene graph" is the Open Inventor term, and "geometry object" is the IRIS Explorer term for visualized data.

The delta protocol is so called because it is possible to convey information about changes in the composition of the original scene graph. These changes occur when you fire a module that renders, colors, stretches, reads, or writes geometry. For example, you may use a widget on an upstream module that processes geometry to alter the shape or orientation of an object in the *Render* module. Other IRIS Explorer modules that process geometry are *ReadGeom*, *WriteGeom*, *LatToGeom*, *PyrToGeom*, and *VolumeToGeom*.

The data type definition follows, although as specified by the ’closed’ keyword, the contents of this structure are not exposed within the Module Builder, as modules do not need direct access. The Geometry API routines manipulate this data structure internally.

```
closed shared root typedef struct {
    short    id;
    long     count; */ length of the data stream in bytes */
    char     data[count];  /* the delta protocol */
    long        numBuffers;
    cxMemory    buffers[numBuffers]; /* Shared memory data */
} cxGeometry;
```

Figure 5–1 shows the structure of the *cxGeometry* data type.

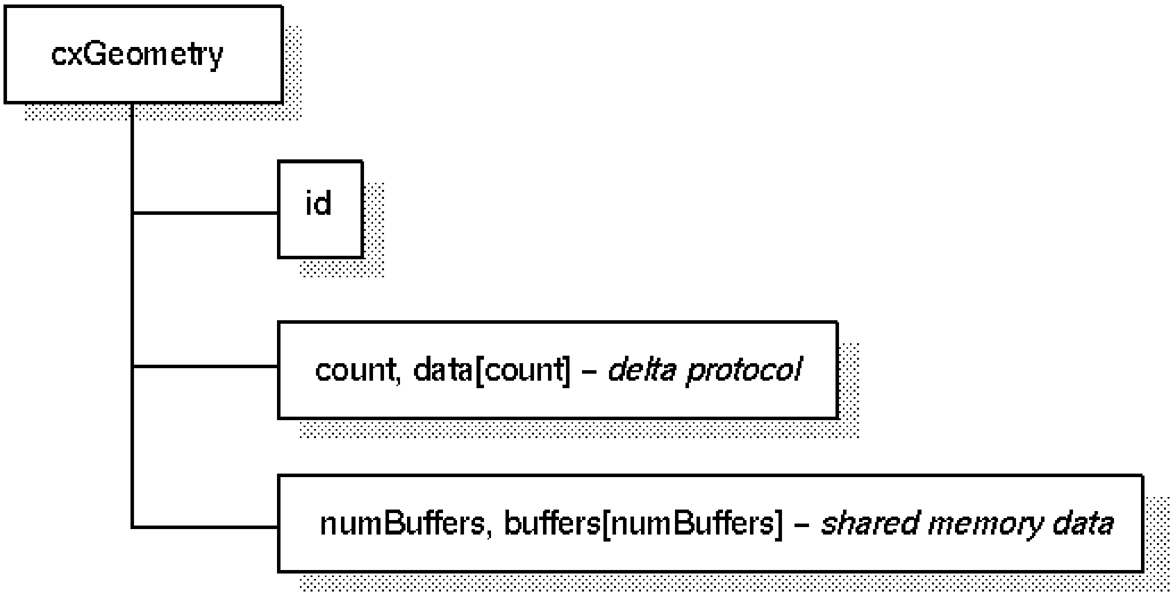


Figure 5–1 Schematic Structure of the Geometry Data Type

5.3. Creating Geometry Modules

A geometry module can be written that accepts non–geometry data but which creates geometry for output using the IRIS Explorer API routines. A simple example follows. These routines let you create geometry shapes and provide a few basic editing functions, such as changing the color of a geometry object, but they do not let you interpret an incoming delta protocol. To write a module that accepts and reads geometry on its input port, you must use Open Inventor. An introduction to Open Inventor scene graphs is given in **"Understanding Scene Graphs"** and is followed by an example geometry module that illustrates the use of Open Inventor. If you plan to use only the simple IRIS Explorer API to create geometry, you do not need to understand how Inventor represents scenes. However, a rudimentary knowledge of how Inventor works may help you understand how geometry is manipulated in IRIS Explorer.

5.3.1. An Example Geometry Module

To illustrate how to use the IRIS Explorer API to create geometry in a module, here is a simple module, that creates some line segments. It has only a geometry output port. The user function for the module resides in */usr/explorer/src/MWGcode/Geometry/C++/LineSegment.C*. The Fortran version is in */usr/explorer/src/MWGcode/Geometry/Fortran/LineSegment.f*.

These geometry API routines are introduced in the example:

- `cxGeoInit` performs initialisation for the Geometry API and also initialises Open Inventor. It must be called once in a module. This can be done using a static variable as shown or specifying an initialisation hook function, shown in the C example **Creating More Geometry Objects**.
- `cxGeoNew` creates a *cxGeometry* data structure. The items in the *cxGeometry* structure are manipulated internally within the Geometry API.
- `cxGeoBufferSelect` sets the current *cxGeometry* structure to be used by the Geometry API, that is its*data* array contains the *delta protocol* resulting from the cxGeo calls that follow.
- `cxGeoBufferPortSet` must be called after `cxGeoBufferSelect` and before any further cxGeo calls. It specifies in advance on which port the *cxGeometry* data will be placed. This allows the geometry API to determine a method of transcription of the geometry objects, dependent on the port's current connections. This is discussed further in **IRIS Explorer Node Classes**
- `cxGeoRoot`,`cxGeoDelete` calls are required to delete the previously created objects in the downstream module(s) as each firing of this module recreates all geometry.
- `cxGeoLinesDefine` creates a geometry object which is a set of lines, and is identified by *id*. This is now the current geometry object. Examples of other geometry objects that may be created are polygons, cones, cylinders, spheres, b–spline curves and patches. Attributes such as color, style, transparency and lighting model can be added to the current geometry object using API calls. To change attributes of previously created objects the current geometry object can be set using the call `cxGeoFocus`.
- `cxGeoBufferClose` must be called to indicate the end of the *delta protocol*

```
/* This example creates some line segments and */
/* outputs them as geometry   */
/* */
#include <cx/DataTypes.h>
#include <cx/DataAccess.h>
#include <cx/Geometry.h>
#include <cx/PortAccess.h>

static float p[] = {
        0.0, 0.0, 0.0,
        1.0, 0.0, 0.0,
        1.0, 1.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 1.0, 1.0,
        1.0, 1.0, 1.0,
        1.0, 0.0, 1.0,
        0.0, 0.0, 1.0
};

static long index[] = {
        0, 1, 2, 3, 0, 7, 6, 5, 4, 7, -1,
        1, 6, -1,
        2, 5, -1,
        3, 4, -1
};


extern "C" void line(cxGeometry **geom)
{
```

```
/* initialize geometry library once only */
 static int flag = 1;
 if ( flag )
 {
        flag = 0;
        cxGeoInit();
 }

  /* Create a new Geometry data type*/
     *geom = cxGeoNew();

  /* Make the buffer in this Geometry data be used to store the
     geometry created by following geometry creation calls i.e.
     the cxGeoLinesDefine call */
     cxGeoBufferSelect(*geom);

  /* Get the port for Geometry output */
     int port = cxOutputPortOpen("Output");

  /* Associate the Geometry data with this port */
     cxGeoBufferPortSet(port);

  /* set the current geometry object to be the root object and then
     delete all geometry created by previous module firings */
     cxGeoRoot();
     cxGeoDelete();

  /* Create a new geometry object */
     cxGeo id = cxGeoLinesDefine(8, p, 20, index);

  /* Close the buffer in the Geometry data, in preparation for
     it to be output on the port */
     cxGeoBufferClose(*geom);

}
```

5.4. Understanding Scene Graphs

A scene graph is the Inventor term for a hierarchical, 3−D description of scene data. It consists of an ordered collection of nodes, each of which defines a specific shape (or geometry), property, or grouping aspect of the scene. The hierarchy is created by adding sublevels of nodes to the top−level or root node in a scene graph, creating a directed acyclic graph. **Figure 5−2** shows a simple scene graph. The heavy line running from the root node down the right−hand branch to the shape node shows the hierarchical path or progression in the graph.

The node

Footnote: "Node" is used here in a different sense from the nodes in a lattice.

is the basic building block for creating scene graphs in Inventor. Each node holds a piece of information, such as a shape description, a geometric transformation, a camera position, or a light source. Each scene graph has a *root* node, which is the first node you create. You can apply actions to a scene graph, such as rendering, picking, and writing to a file. The IRIS Explorer *Render* module renders numerical data in the form of a scene graph.

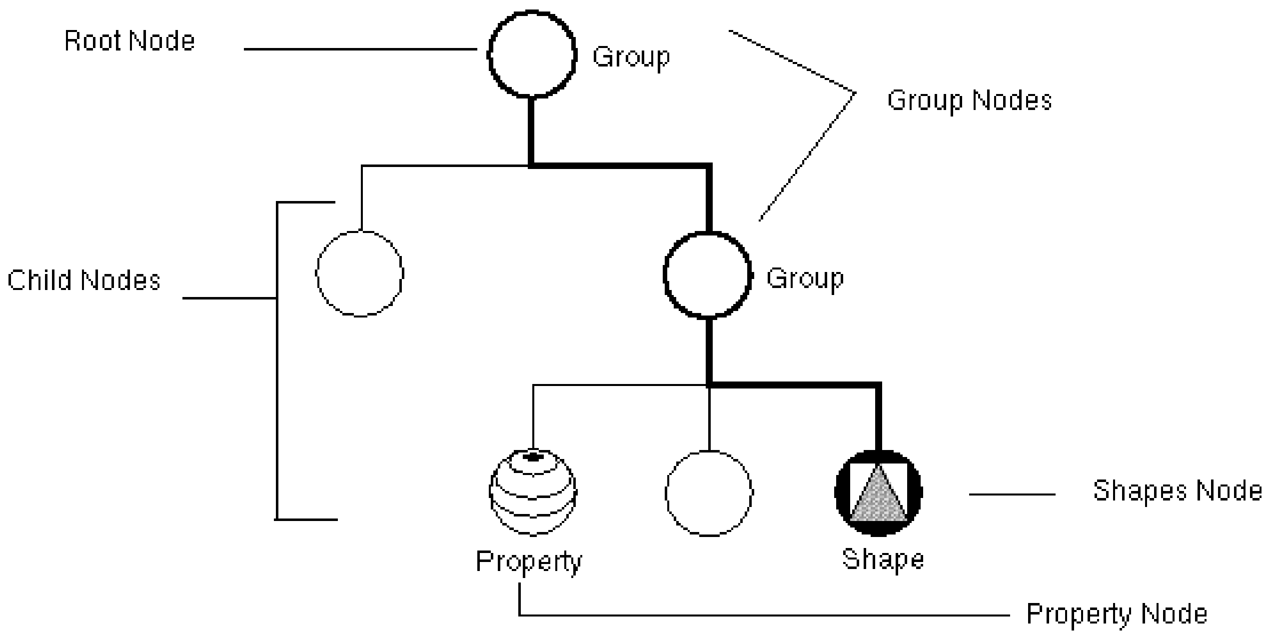


Figure 5−2 Inventor Scene Graph

Nodes fall into three categories: shape nodes, which represent 3−D geometric objects; property nodes, which represent qualitative values, such as appearance; and group nodes, which collect single nodes into graphs.

Each node has a set of fields that describe the parameters of the node. For example, a node defining a point light source has three fields, intensity, color, and location. Each of these fields can be given a value (location) or range of values (intensity). The order in which nodes are arranged is important because it determines how the values that each node contains are acted upon.

5.5. Using Open Inventor in Geometry Modules

5.5.1. Creating Geometry

Calls to the Geometry API create Inventor nodes and simultaneously transcribe the node data into the *delta protocol* stored in the *cxGeometry* data buffer. Inventor classes can be used directly to create a scene graph within an IRIS Explorer module. To move this scene graph to another module it must still be transcribed into the *cxGeometry* data buffer.

The routine `cxGeoInventorDefine` is used to perform the transcription of the root node and all nodes beneath it in the hierarchy. This call will also attach the given root node to the scene graph within the geometry API − when `cxGeoInit` is called a root node is always created. Standard Open Inventor binary transcription is performed using objects of classes *SoTranSender* and *SoOutput*.

A set of Open Inventor node classes have been implemented for use solely within IRIS Explorer. These classes, whose names are prefixed *cxSo*, for example *cxSoIndexedFaceSet*, have identical functionality to the corresponding standard Inventor node classes with *So* names. The difference is that they use new field classes derived from the standard Open Inventor multi−valued field classes. These new field classes, again prefixed *cxSo*, for example *cxSoMFLong*, store their data in the IRIS Explorer shared memory arena, when available. See **Understanding Reference Counting** for further discussion of shared memory.

These *cxSo* nodes are used when creating geometry within the Geometry API. To obtain the performance advantage of using shared memory when moving scene graphs between modules you should create nodes with these classes when using Open Inventor directly within IRIS Explorer modules. Simply replace the declaration of an *So* node class with the corresponding *cxSo* class.

To illustrate the use of Inventor, including the cxSo nodes, within an IRIS Explorer module, here is an example that creates a set of polygons. The classes cxSoCoordinate3, cxSoMaterial and cxSoNormal are used, therefore the appropriate header files in *cx/nodes* are included. The module has a geometry output port. The user function for the module resides in */usr/explorer/src/MWGcode/Geometry/C++/FaceSet.C*.

```
// This example creates a Face Set using the Open Inventor
// library and the Explorer extension classes whose field
// data is stored in Explorer shared memory

#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/Geometry.h>
#include <cx/PortAccess.h>

#include <cx/nodes/cxSoCoordinate3.h>
#include <cx/nodes/cxSoMaterial.h>
#include <cx/nodes/cxSoNormal.h>

#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoNormalBinding.h>
#include <Inventor/nodes/SoSeparator.h>

//  Eight polygons. The first four are triangles
//  The second four are quadrilaterals for the sides.
static float vertices[28][3] =
{
   { 0, 9, 0}, {-2,6, 2}, { 2,6, 2},            //front tri
   { 0, 9, 0}, {-2,6,-2}, {-2,6, 2},            //left  tri
   { 0, 9, 0}, { 2,6,-2}, {-2,6,-2},            //rear  tri
   { 0, 9, 0}, { 2,6, 2}, { 2,6,-2},            //right tri
   {-2, 6, 2}, {-4,0, 4}, { 4,0, 4}, { 2,6, 2},  //front quad
   {-2, 6,-2}, {-4,0,-4}, {-4,0, 4}, {-2,6, 2},  //left  quad
   { 2, 6,-2}, { 4,0,-4}, {-4,0,-4}, {-2,6,-2},  //rear  quad
   { 2, 6, 2}, { 4,0, 4}, { 4,0,-4}, { 2,6,-2}   //right quad
};

// Number of vertices in each polygon:
static long numvertices[8] = {3, 3, 3, 3, 4, 4, 4, 4};

// Normals for each polygon:
static float norms[8][3] =
{
   {0, .555,  .832}, {-.832, .555, 0}, //front, left tris
   {0, .555, -.832}, { .832, .555, 0}, //rear, right tris

   {0, .0739,  .9973}, {-.9972, .0739, 0},//front, left quads
   {0, .0739, -.9973}, { .9972, .0739, 0},//rear, right quads
};

extern "C" {
```

```
   void makeFaceSet(
        cxGeometry **geo)
{

// initialize geometry api once only
// this creates a root node
   static int first = 1;
   if ( first )
   {
    first = 0;
    cxGeoInit();
   }

// Create a new Geometry data type
   *geo = cxGeoNew();

// Make the buffer in this Geometry data be used to store the
// geometry created
   cxGeoBufferSelect(*geo);

// Get the port for Geometry output and associate
// the Geometry data with this port
   cxGeoBufferPortSet(cxOutputPortOpen("Face Set"));

// set the current geometry object to be the root node and
// delete all geometry created by previous module firings
// below the root node
   cxGeoRoot();
   cxGeoDelete();

// create the Inventor Scene using the Explorer nodes
// so that their field data is stored in shared memory

   SoSeparator *topNode= new SoSeparator();
   topNode->ref();

// Define the normals used:
   cxSoNormal *myNormals = new cxSoNormal;
   myNormals->vector.setValues(0, 8, norms);
   topNode->addChild(myNormals);
   SoNormalBinding *myNormalBinding = new SoNormalBinding;
   myNormalBinding->value = SoNormalBinding::PER_FACE;
   topNode->addChild(myNormalBinding);

// Define material for topNode
   cxSoMaterial *myMaterial = new cxSoMaterial;
   myMaterial->diffuseColor.setValue(.4, .4, .4);
   topNode->addChild(myMaterial);

// Define coordinates for vertices
   cxSoCoordinate3 *myCoords = new cxSoCoordinate3;
   myCoords->point.setValues(0, 28, vertices);
   topNode->addChild(myCoords);
```

```
// Define the FaceSet
   SoFaceSet *myFaceSet = new SoFaceSet;
   myFaceSet->numVertices.setValues(0, 8, numvertices);
   topNode->addChild(myFaceSet);

// Add this Scene Graph to the Geometry api's root node
// and add to the Geometry buffer
   cxGeoInventorDefine(topNode);

// Decrement reference count as it is now referenced by root node
   topNode->unref();

// Close the buffer in the Geometry data, in preparation for
// it to be output on the port
   cxGeoBufferClose(*geo);

}

}
```

5.5.2. IRIS Explorer Node Classes

The cxSo node classes that are available are listed below, together with the cxSo field class used for the attribute data of the node.

The transcription of the Inventor Scene Graph into the Geometry data buffer differs for cxSo nodes. It is not necessary to transcribe all the field data into the binary data stream, only the pointer to memory is added when shared memory is appropriate for the module to module connection. It is this that gives the performance advantage when moving scene graphs between modules. For this reason it is necessary to identify which port the *cxGeometry* data will be placed on before making further cxGeo calls, so that the correct transcription method can be used.

Table 5–1 lists the cxSo node classes and referenced cxSo field classes.

cxSo Node	Attribute Name	cxSo Field
cxSoColorIndex	index	cxSoMFLong
cxSoCoordinate3	point	cxSoMFVec3f
cxSoCoordinate4	point	cxSoMFVec4f
cxSoIndexedFaceSet		
cxSoIndexedLineSet		
cxSoIndexedShape	materialIndex	cxSoMFLong
	normalIndex	cxSoMFLong
	textureCoordIndex	cxSoMFLong
cxSoIndexedTriangleStripSet		
cxSoMaterial	ambientColor	cxSoMFColor
	diffuseColor	cxSoMFColor
	specularColor	cxSoMFColor
	emissiveColor	cxSoMFColor
	shininess	cxSoMFFloat
	transparency	cxSoMFFloat
cxSoMaterialIndex	ambientIndex	cxSoMFLong
	diffuseIndex	cxSoMFLong
	specularIndex	cxSoMFLong

	shininess	cxSoMFFloat
	transparency	cxSoMFFloat
cxSoNormal	vector	cxSoMFVec3f
cxSoPackedColor	rgba	cxSoMFULong

Table 5–1 IRIS Explorer Shared Memory Node Classes

5.5.3. Receiving Geometry

Open Inventor must be used in modules that receive geometry on input ports. An Inventor *receiver object* is required to decode the incoming data from a specific upstream module and create the resulting Scene Graph. The API function `cxGeoReceiverNew` creates this object, and `cxGeoReceive` performs the decoding from the given *cxGeometry* object. The incoming scene graph is attached to a parent node that has been previously created.

A different receiver object should be used for each geometry connection. This prevents data updates from one module being passed to nodes that contain data from another module. The parent node/receiver object pairs control the distribution of new data to the correct nodes.

The example **A Simple Renderer** illustrates a user function that receives geometry.

5.6. Data Type Declaration

The *cxGeometry* data type, though defined in the IRIS Explorer typing language, can be considered as a C structure. However the *cxGeometry* data type is usually accessed internally via the geometry API functions. It should not be necessary for module user code to access the data structure directly. The type declaration resides in the header file */usr/explorer/include/cx/cxGeometry.h.*

The *cxGeometry* data type structure is defined as follows:

```
typedef struct cxMemory {
    cxDataCtlr       ctlr;
    long              n;
    unsigned char    *mem;
} cxMemory;

typedef struct cxGeometry {
    cxDataCtlr       ctlr;
    short             id;
    long              count;
    unsigned char    *data;
    long              numBuffers;
    cxMemory        **buffers;
} cxGeometry;
```

5.7. The Geometry API Routines

The *cxGeometry* data type has its own API subroutine library that enables you to write simple IRIS Explorer modules that output scene graphs without manipulating Open Inventor code. You can create and manipulate the geometry objects by using the geometry subroutines listed below. To create more complex modules, you can use Open Inventor directly.

The geometry subroutines are described in the *IRIS Explorer Reference Pages* and the on–line *man* pages. **Table 5–2** lists the subroutines and briefly defines each one.

Subroutine	Purpose

cxGeoNew	Creates a new IRIS Explorer geometry data structure
cxGeoInit	Initializes the IRIS Explorer Geometry Library
cxGeoPointsDefine	Creates a set of points
cxGeoLinesDefine	Creates a set of unclosed polylines
cxGeoPolysDefine	Creates a set of polygons
cxGeoSplineDefine	Creates a NURB spline
cxGeoTrisDefine	Creates a set of triangle meshes
cxGeoPatchDefine	Creates a set of spheres
cxGeoSpheresDefine	Creates a set of spheres
cxGeoConesDefine	Creates a set of cones
cxGeoCylindersDefine	Creates a set of cylinders
cxGeoGridDefine	Creates a grid geometry object
cxGeoTextDefine	Creates a text object
cxGeoOctreeDefine	Creates a volume object rendered as an octree
cxGeoInventorDefine	Creates an entire Open Inventor scene graph
cxGeoNormalAdd	Adds normals to the current geometry object
cxGeoColorAdd	Adds colors to the current geometry object
cxGeoABGRAdd	Adds packed (32−bit) colors to the current geometry object
cxGeoComplexityAdd	Specifies the level of rendering detail
cxGeoLabelAdd	Adds a label to the current geometry object
cxGeoStyleAdd	Adds a drawing style to the current geometry object
cxGeoTransparencyAdd	Add transparency to the current geometry object
cxGeoLightModelAdd	Specifies the lighting model for the current geometry object
cxGeoXformPop	Makes the current object the parent of the current transform group
cxGeoXformPush	Groups objects under a parent hierarchy object
cxGeoIdentity	Resets the transform on the current object
cxGeoTranslate	Translates the current geometry object
cxGeoRotate	Rotates the current geometry object
cxGeoScale	Scales the current geometry object
cxGeoMatrixCat	Applies an arbitrary 4x4 transform to the object
cxGeoReceive	Receives IRIS Explorer input data for transcription
cxGeoFocus	Specifies the current object or transform for editing
cxGeoRoot	Makes the root object the current object
cxGeoDelete	Makes the root object the current object
cxGeoBufferClose	Prepares a geometry buffer to be sent
cxGeoBufferPortSet	Associates a port with the current geometry buffer
cxGeoBufferSelect	Specifies a geometry buffer for subsequent scene edits

Table 5−2 Geometry Subroutines

5.8. Code Examples

The examples that follow show the use of both IRIS Explorer subroutines and Open Inventor classes to create and manipulate geometry in modules. The code examples are located in the */usr/explorer/src/MWGcode/Geometry/** subdirectories

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

.

5.8.1. Modules using the Geometry API

Creating Line Segments

This example is the Fortran version of the previous C++ example that creates some line segments and outputs them as geometry. The code is in *MWGcode/Geometry/Fortran/LineSegment.f.*

```
C This module creates some line segments and outputs them as geometry

      subroutine linef ( h_geom )

      integer h_geom
C
C Please complete the user function by adding subprogram lines here.
C

      include '/usr/explorer/include/cx/DataAccess.inc'
      include '/usr/explorer/include/cx/Geometry.inc'
      include '/usr/explorer/include/cx/PortAccess.inc'
      integer   flag
      real      p(24)
      integer   index(20)
      integer port, id

      data flag/1/
      data p/0.0, 0.0, 0.0,
     +  1.0, 0.0, 0.0,
     +  1.0, 1.0, 0.0,
     +  0.0, 1.0, 0.0,
     +  0.0, 1.0, 1.0,
     +  1.0, 1.0, 1.0,
     +  1.0, 0.0, 1.0,
     +  0.0, 0.0, 1.0/
      data index/0, 1, 2, 3, 0, 7, 6, 5, 4, 7, -1,
     +           1, 6, -1,
     +           2, 5, -1,
     +           3, 4, -1/

C  initialize geometry library once only
      if ( flag .eq. 1 ) then
          flag = 0
          call cxGeoInit()
      endif
C
C  Create a new Geometry data type
      h_geom = cxGeoNew()
C
C  Make the buffer in this Geometry data be used to store the
C  geometry created by following geometry creation calls i.e.
C  the cxGeoLinesDefine call
      call cxGeoBufferSelect(h_geom)
C
C  Get the port for Geometry output
```

```
      port = cxOutputPortOpen("Output")
C
C  Associate the Geometry data with this port
      call cxGeoBufferPortSet(port)
C
C  set the current geometry object to be the root object and then
C  delete all geometry created by previous module firings
      call cxGeoRoot()
      call cxGeoDelete()
C
C  Create a new geometry object
      id = cxGeoLinesDefine(8,p,20,index)
C
C  Close the buffer in the Geometry data, in preparation for
C  it to be output on the port
      call cxGeoBufferClose(h_geom)
C
      RETURN
      END
```

Creating More Geometry Objects

The following example creates spheres, lines and polygons. The code is in */usr/explorer/src/MWGcode/Geometry/C/PolySegment.c* and */usr/explorer/src/MWGcode/Geometry/Fortran/PolySegment.f.*

C Version:

```
/* This example uses the API to create geometry and attach attributes
 * to the geometry
 * The module data wrapper is not used in this module,
   port access is therefore via api calls */

#include <cx/DataTypes.h>
#include <cx/DataAccess.h>
#include <cx/PortAccess.h>
#include <cx/UserFuncs.h>
#include <cx/Geometry.h>

#include <cx/Lookup.h>

float lins[] = {0.,0.,0., 0.,0.,1., 0.,1.,1., 0.,1.,0.,
                 1.,0.,0., 1.,0.,1., 1.,1.,1., 1.,1.,0. };

int conn[] = {0,1,2,3,0,4,5,6,7,4,-1,1,5,-1,2,6,-1,3,7,-1};
int conn2[] = {0,1,5,4,-1,2,3,7,6,-1};

void display()
{
 cxLattice *cmap;    /* Color map */
 cxGeometry *geo;
 cxLookup *clut;
 float col[4],sphs[10][3],rads[10],x;
 int i,j;
```

```
/* Get lattice for colormap from port and create lookup table */
cmap = (cxLattice*)cxInputDataGet(cxInputPortOpen("colormap"));
if(cmap == NULL){
        printf("geometry: no color map present\n");
        return;
}
clut = cxLookupCreate(cmap,1);

/* Create Geometry data type and set its buffer for the
   output of geometry */
geo = cxGeoNew();
if( cxDataAllocErrorGet() ) goto cleanupGeo;
cxGeoBufferSelect(geo);
/* set the port on which this geometry will be sent */
cxGeoBufferPortSet(cxOutputPortOpen("geom"));

/* Delete all geometry created by previous module firing */
cxGeoRoot();
if( cxDataAllocErrorGet() ) goto cleanupSelected;
cxGeoDelete();
if( cxDataAllocErrorGet() ) goto cleanupSelected;

/* Set radii and positions of spheres  and create the spheres*/
for(i=0;i<10;i++){
        for(j=0;j<3;j++){
               sphs[i][j] = i;
        }
        rads[i] = i / 10.0 + 0.01;
}

cxGeoSpheresDefine(5,sphs[0],rads);
if( cxDataAllocErrorGet() ) goto cleanupSelected;
x = 0.8;
cxLookupInterp(clut,&x,(void *)col);
cxGeoColorAdd(1,col,CX_GEO_PER_OBJECT);
if( cxDataAllocErrorGet() ) goto cleanupSelected;

/* Create the lines of the cube */
cxGeoLinesDefine(8,lins,20,conn);
if( cxDataAllocErrorGet() ) goto cleanupSelected;
x = 0.5;
cxLookupInterp(clut,&x,(void *)col);
cxGeoColorAdd(1,col,CX_GEO_PER_OBJECT);
if( cxDataAllocErrorGet() ) goto cleanupSelected;

/* Create the polygons of the cube */
cxGeoPolysDefine(8,lins,10,conn2);
if( cxDataAllocErrorGet() ) goto cleanupSelected;
x = 0.2;
cxLookupInterp(clut,&x,(void *)col);
cxGeoColorAdd(1,col,CX_GEO_PER_OBJECT);
if( cxDataAllocErrorGet() ) goto cleanupSelected;
```

```
 /* Close the buffer of the geometry data - ready for
    sending on the port */
 cxGeoBufferClose(geo);
 cxOutputDataSet(cxOutputPortOpen("geom"),geo);

 cxLookupDestroy(clut);

 return;

 /* Handle error conditions */
cleanupSelected:
 cxGeoBufferClose(geo);
cleanupGeo:
 cxDataRefDec(geo);
 cxLookupDestroy(clut);
 return;
}

/* Function specified as an 'Init Hook Function'
   in the Module Builder Build/Hook Functions option */
void geom_init()
{
 /* Initialise the geometry api first time the module fires */
 cxGeoInit();
}
```

Fortran Version:

```
c This example uses the API to create geometry and attach attributes
c to the geometry
c

        subroutine display(cmap,geo)
        include '/usr/explorer/include/cx/DataAccess.inc'
        include '/usr/explorer/include/cx/Lookup.inc'

        integer geo,cmap,clut
        integer CX_GEO_PER_OBJECT
        parameter(CX_GEO_PER_OBJECT = 1)
        logical first
        integer conn(20),conn2(10)
        real sphs(3,10),rads(10),lins(3,8),col(4)
        data lins/0.,0.,0., 0.,0.,1., 0.,1.,1., 0.,1.,0.,
     1           1.,0.,0., 1.,0.,1., 1.,1.,1., 1.,1.,0. /
        data conn/0,1,2,3,0,4,5,6,7,4,-1,1,5,-1,2,6,-1,3,7,-1/
        data conn2/0,1,5,4,-1,2,3,7,6,-1/
        data first/.true./
c
c Initialise the geometry api first time the module fires
        if(first)then
                call cxGeoInit
                first = .false.
        endif
c
```

```
c Create color lookup table from input lattice
        clut = cxLookupCreate(cmap,1)
c
c Create Geometry data type and set its buffer for the
c output of Geometry
        geo = cxGeoNew()
        if( cxDataAllocErrorGet() .NE. 0 ) goto 9999
        call cxGeoBufferSelect(geo)
c set the port on which this geometry will be sent
        call cxGeoBufferPortSet(cxOutputPortOpen("geom"))

c Delete all geometry created by previous module firing
        call cxGeoRoot
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
        call cxGeoDelete

c Set radii and positions of spheres  and create the spheres
c
        do i=1,10
         do j=1,3
                sphs(j,i) = i
         end do
         rads(i) = real(i)/10.0
        end do
c
        call cxGeoSpheresDefine(5,sphs,rads)
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
        x =0.8
        call cxLookupInterp(clut,x,col)
        call cxGeoColorAdd(1,col,CX_GEO_PER_OBJECT)
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
c
c Create the lines of the cube
c
        call cxGeoLinesDefine(8,lins,20,conn)
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
        x = 0.5
        call cxLookupInterp(clut,x,col)
        call cxGeoColorAdd(1,col,CX_GEO_PER_OBJECT)
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
c
c Create the polygons of the cube
c
        call cxGeoPolysDefine(8,lins,10,conn2)
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
        x = 0.2
        call cxLookupInterp(clut,x,col)
        call cxGeoColorAdd(1,col,CX_GEO_PER_OBJECT)
        if( cxDataAllocErrorGet() .NE. 0 ) goto 999
c
c Close the buffer of the geometry data - ready for
c sending on the port
c
        call cxGeoBufferClose(geo)
```

```
        call cxLookupDestroy(clut)
        return
c
c       memory allocation error cleanup
c
999     continue
        call cxGeoBufferClose(geo)
9999    continue
        call cxDataRefDec(geo)
        call cxLookupDestroy(clut)
        return
        end
```

Figure 5–3 shows what the module output looks like.

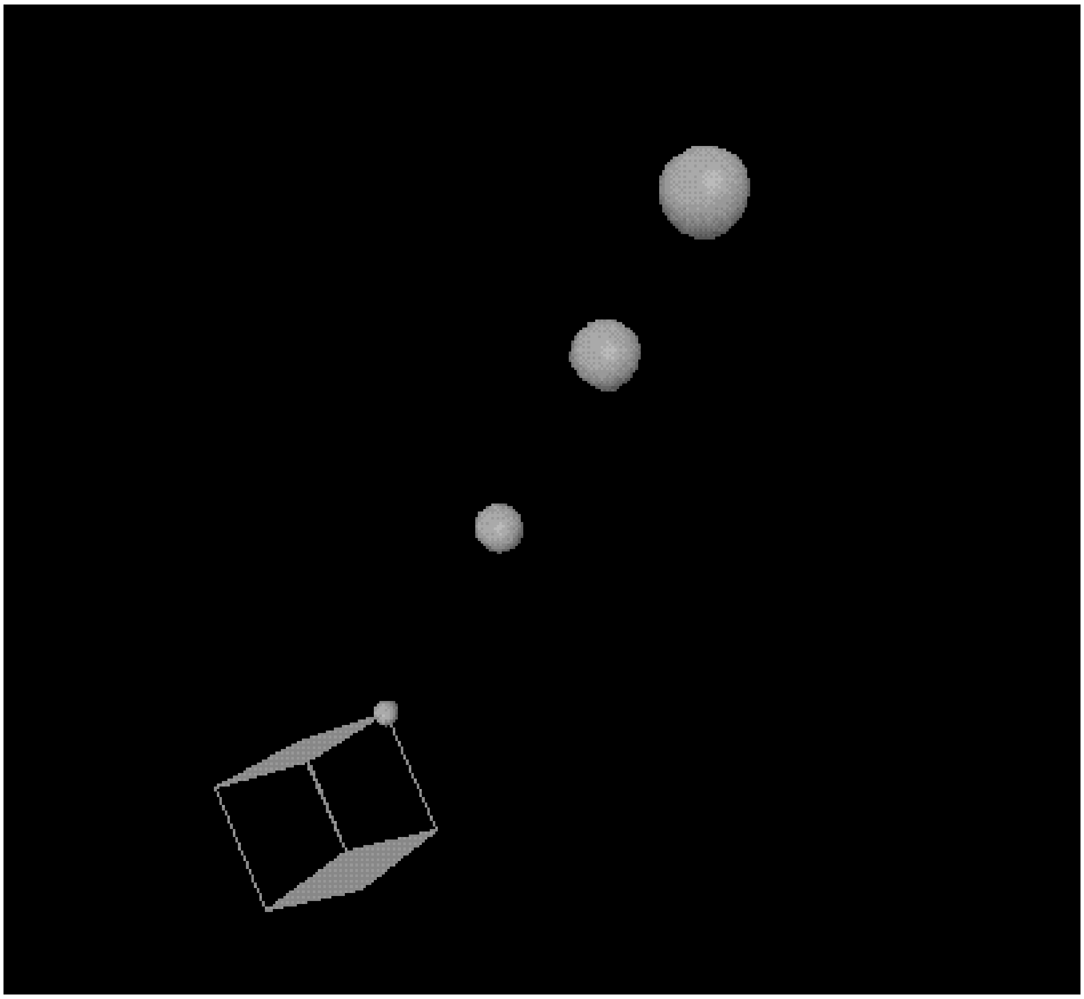


Figure 5–3 Output from Geometry Example

5.8.2. Creating Modules with Open Inventor Code

Here are two examples of a user function showing how you can use Open Inventor to create and display geometry objects. Neither example has a Fortran equivalent because Open Inventor has no Fortran API.

Texturing a Sphere

This is a simple example of how to use the Open Inventor library, written in C++. The function wraps an image onto a sphere. The code is in */usr/explorer/src/MWGcode/Geometry/C++/Texture.C*

```
// This example creates a sphere and wraps an image
// onto it, using the Open Inventor library. This
// Inventor scene graph is then added to the Explorer
// Geometry buffer.

#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/Geometry.h>
#include <cx/PortAccess.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoTexture2.h>
#include <Inventor/nodes/SoSeparator.h>
#include <cx/nodes/cxSoMaterial.h>


extern "C" {
  void  makeSphere  (
         float            radius,
         char            *texfile,
         cxGeometry *    * geo )
{
// initialize geometry api once only
// this creates a root node
  static int first = 1;
  if ( first )
  {
    first = 0;
    cxGeoInit();
  }

// Create a new Geometry data type
  *geo = cxGeoNew();

// Make the buffer in this Geometry data be used to store the
// geometry created
  cxGeoBufferSelect(*geo);

// Get the port for Geometry output and associate
// the Geometry data with this port
  cxGeoBufferPortSet(cxOutputPortOpen("Texture Geometry"));

// set the current geometry object to be the root node and
// delete all geometry created by previous module firings
// below the root node
  cxGeoRoot();
  cxGeoDelete();
```

```
// create the Inventor Scene
  SoGroup *topNode = new SoSeparator();
  topNode->ref();
  SoSphere *sph = new SoSphere();
  SoTexture2 *tex = new SoTexture2();
// Use the Explorer Material class so that its data is
// stored in shared memory
  cxSoMaterial *m = new cxSoMaterial();
  m->diffuseColor.setValue(1.0,1.0,1.0);
  tex->filename.setValue(texfile);
  sph->radius = radius;
  topNode->addChild(tex);
  topNode->addChild(m);
  topNode->addChild(sph);

// Add this Scene Graph to the Geometry api's root node
// and add to the Geometry buffer
  cxGeoInventorDefine(topNode);

// Decrement reference count as it is now referenced by root node
  topNode->unref();

// Close the buffer in the Geometry data, in preparation for
// it to be output on the port
  cxGeoBufferClose(*geo);
}

}
```

A Simple Renderer

This is a reasonably small module, written in C++, that implements a geometry renderer using the Open Inventor library. The code is in */usr/explorer/src/MWGcode/Geometry/C++/SimpleRender.C*

For simplicity, several limitations have been accepted:

- The rendering window does not downsize very gracefully. The real renderer turns off decoration when the window goes below a certain size.
- Only one geometry input is supported. This module can have fatal namespace collisions with fanned–in inputs, though in practice it will usually work. The proper way to handle multiple geometry streams is to use a different receiver object for each one.
- The viewer in this module is very simple and does not support any editing or picking facilities.

```
/*
 * This example creates a simple render module using
 * Inventor code and is an example of a module that
 * receives Geometry data.
 *
 */

#include <X11/Intrinsic.h>

#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoSeparator.h>
```

```
#include <Inventor/nodes/SoShapeHints.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/Geometry.h>
#include <cx/PortAccess.h>
#include <cx/UserFuncs.h>
#include <cx/XtArea.h>

Widget top;             /* top level widget for drawing area */
SoGroup *group;         /* root for incoming geometry */
cxGeoReceiver receiver; /* receiver object for incoming geometry */
SoXtViewer *viewer;     /* Inventor viewer */

extern "C" void renderlite(
    cxGeometry *geo,    /* incoming geometry */
    int geoFlag,        /* nonzero if geometry is new */
    long window,        /* window id of drawing area */
    int windowFlag)     /* nonzero if window is new */
{
    static int first = 1;   /* initialization flag */
    if ( first && window )
    {
        /* reset initialization flag */
        first = 0;

        /* initialize geometry api once only  -
           this initialises cxSo classes so that they
           are recognised in incoming geometry */
        cxGeoInit();

        /* create top level widget for drawing area */
        top = cxXtAreaInitialize();

        /* initialize Inventor for use with Xt*/
        SoXt::init(top);

        /* create the environment */
        SoGroup *root = new SoSeparator();
        root->ref();

        /* shape hints */
        SoShapeHints *n1 = new SoShapeHints();
        n1->vertexOrdering = SoShapeHints::COUNTERCLOCKWISE;
        root->addChild(n1);

        /* create the node for the incoming geometry */
        group = new SoGroup();
        root->addChild(group);

        /* create the receiver object */
        receiver = cxGeoReceiverNew(group);
```

```
        /* create the viewer */
        viewer = new SoXtExaminerViewer();
        viewer->setSceneGraph(root);
        viewer->setSize(SbVec2s(100,100));
        viewer->show();

/* attach the viewer widget to the Explorer Form widget */
        Widget vw = viewer->getWidget();
        XtVaSetValues(vw,
                XmNtopAttachment,XmATTACH_FORM,
                XmNbottomAttachment,XmATTACH_FORM,
                XmNrightAttachment,XmATTACH_FORM,
                XmNleftAttachment,XmATTACH_FORM,
                NULL);

        /* attach widget to window */
        cxXtAreaAttach(top,window);
   }

   if ( windowFlag )    /* window has changed state or size */
        cxXtAreaResize(top,window);

   if ( geoFlag )               /* new geometry has arrived */
   {
        cxGeoReceive(geo,receiver);
        viewer->viewAll();
   }
}
```

Chapter 6

Using the Pick Data Type

This chapter describes the *cxPick* data type and how to use it in the IRIS Explorer environment. It defines the data type and describes how you can use it to write modules that read or write pick data. This is a specialized data type and requires a fairly sophisticated module writer to understand it. In general, you should be able to use the standard IRIS Explorer modules that accept and manipulate the *cxPick* data type for most of your needs. They include *Render*, *QueryLat*, *DisplayImg*, and *MoleculeBuilder*.

The API (Application Programming Interface) routines are listed and there are examples of user function code for writing modules that manipulate pick data.

6.1. Understanding the Picking Function

The data type described here, *cxPick*, can be used in two kinds of modules: those that read *cxPick* data objects on their input ports, and those that write *cxPick* data objects through their output ports.

The purpose of the *cxPick* data type is to enable the user to pick, or select, a particular location in a module display window and obtain information about it. *cxPick* is a specialized data type, used only in 2D and 3D modules.

You can obtain information about an image in the window of a module that displays images (2D), or a geometry object in the window of a rendering module (3D).You do this by using the mouse to select items out of a picture in a module display window. For example, you may want to query part of a picture of an isosurface. You could find the coordinates of a spot on the surface or the temperature at that point. The window in which the isosurface is displayed can be in the *Render* module, in *DisplayImg*, or in your own rendering module, all of which have Pick output ports.

You can send the information you have picked in the *Render* or *DisplayImg* window to a module with a Pick input port, such as *PrintPick*.

You can also create an object where one did not previously exist, or position an object in a particular location in a rendering module display window. As you click on the position in the window, the information goes to the upstream module, which creates an object, for example a line or an atom, and sends it to the selected position in the Render window.

There are two points to note about the *cxPick* data type:

- It is considerably easier to write modules that read *cxPick* data sent downstream by modules such as *DisplayImg* or *Render* than it is to write modules that generate pick data.
- If you write a pick module which outputs pick data, you need to write your own user interface using the X Window System, OpenGL, or a toolkit on top of these. This is because none of the IRIS Explorer widgets supply pick information.

6.2. Using the Pick Data Type

You can use *cxPick* in any module that you want to output information about geometry objects, and in any module that you want to receive this information. In the IRIS Explorer module suite, for example, the *QueryLat*, *PrintPick* and *MoleculeBuilder* modules all have *cxPick* on their input ports and can therefore read pick data. The *Render* module can write pick data to its Pick output port and hence feed information back to the *QueryLat* and *MoleculeBuilder* modules.

Module users can activate Pick mode in a Render window by selecting "User Pick Mode" from the Viewing Menu and clicking on the "arrow" icon to turn off viewing mode (to go back to viewing mode click on the "hand" icon). To use Pick mode in a *DisplayImg* window, they simply press the middle mouse button.

The *cxPick* data type provides a vast amount of data, as you can see from its data type structure in **Figure 6–1**. It takes some thought to decide exactly what you want *cxPick* to do for you. There are two ways in which module writers commonly use *cxPick*; to build modules that read pick data, and to build modules that write pick data. Further, your module could read pick data from either an image (2D) or from geometry (3D). Or it could write pick data obtained from an image or a geometry object.

Modules that Read Pick Data

Modules that read pick data need only inspect the components of the *cxPick* data structure in which the module has an interest. These modules, such as *QueryLat* and *PrintPick*, receive the data from a module that writes pick data.

For example, *QueryLat* reads information about the position of the mouse cursor relative to a lattice in the Render window. It takes the information, generates a text label that contains the information, and sends the label back to the Render window for the user to see. The code is given in **"Writing Pick Data"** later in this chapter.

Modules that Write Pick Data

Modules that write pick data include the *Render* module and *DisplayImg* in IRIS Explorer. These modules tend to be more difficult to write because of the effort it takes to manage the user interface.

6.2.2. The cxPick Structure

The variables in the *cxPick* data structure let you specify all the information you want to put into or get from the module. You may not need all the variables; for example, if you write a module that displays images, you will not need the variables that pertain to the picking of geometric objects. When you create the *cxPick* type, you should null out all the unfilled variables.

Terminology

To make the function of each variable easier to understand, it helps to clarify some terms. A *pick* is what you do when you select a spot in the display window of a module by clicking the mouse on it. A *hit* is when you touch on part of an object or image, instead of empty space. A *pick object* is the object or image you touched on. A *pick event* is the act of picking the object, and it occurs when you press a mouse button or key, or drag the mouse across the window.

Once you have made a hit, you can gather a great deal of information about the object you hit. This information is laid out in the subtype, *cxPickHit.*

This is the data type definition:

```
root typedef struct
{
    int           eventType               "Event Type";
    long          eventData               "Event Data";
    long          eventModifiers          "Modifier Keys";
    int           position[2]             "Position Array";
    int           windowSize[2]           "Window Size Array";
    float         origin[3]               "Ray Origin";
    float         direction[3]            "Ray Direction";
    float         projectionMatrix[4, 4]  "Projection Matrix";
    int           numHits                 "Num Object Hits";
    cxPickHit     hit[numHits]            "Hits Array";
} cxPick;
```

The variables are:

event type	Indicates what the user action was, i.e. whether the user dragged the mouse, pressed or released a mouse button, or pressed a key. The action is specified by the X Window event number for the particular action.
eventData	Indicates which mouse button or key was pressed. The X Window System definitions are used for

	buttons and keys.
eventModifiers	Indicates whether a modifier key, such as **`<Shift>`** or **`<Alt>`**, was pressed. Specified by the X Window event set of flags.
position	Gives the exact 2D position in the window where the action took place. It is *x,y* from the top left corner, in integers.
windowSize	Gives the size of the module window (in pixels) in which the pick occurred.
origin	Indicates the position of the eye (3D only). It can be defined as a ray running from eye point to the pick point, and it has both position and direction.
direction	Specifies the direction of the ray from eye to object (3D only)
projectionMatrix	Specifies the 3D projection; contains a perspective transformation, if present. Of the standard IRIS Explorer modules, only *Render* generates this data.
numHits	Gives the number of object that were clicked on, or "hit" by the user's action.
hit	Tells you what kind of object you hit, ordered from nearest to furthest from the viewer. The information is given by *cxObjectDefine*, which specifies whether the hit was on a point, face, or volume.

If the user does not hit any object in the module window, or if the window is empty, the variable *numhits* is zero and *hits* will contain no data.

If the user actually hits something, then a *cxPickhit* data structure will be filled in with information about the object or objects.

Figure 6–1 depicts the pick data type structure.

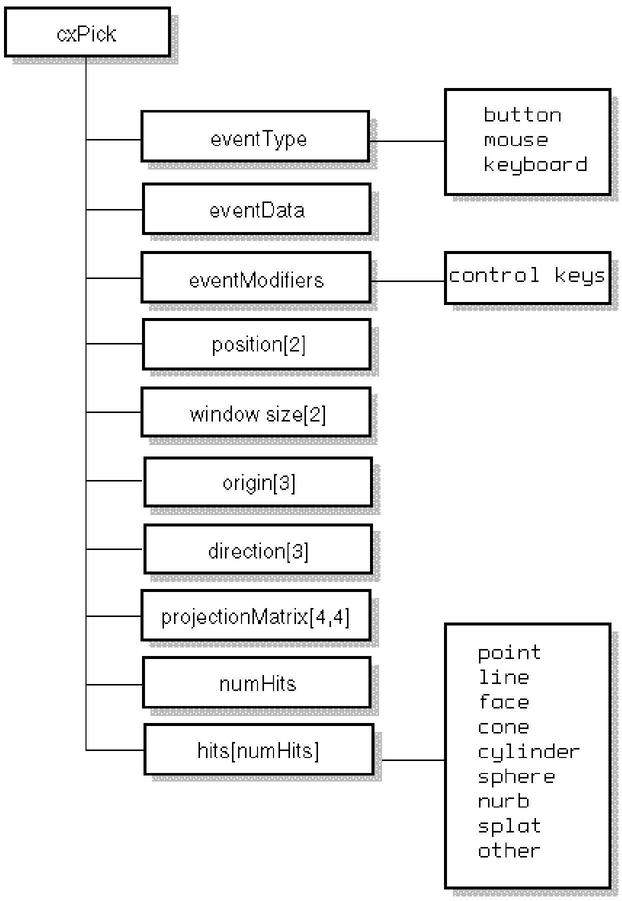


Figure 6–1 Schematic Structure of the Pick Data Type

6.2.3. Getting Data on the Hit Object

When the user presses a button in Pick mode and hits an object or image, the *cxPickHit* data structure can provide a lot of information about that object or image.

The data type definition is:

```
typedef struct
{
    long            id;

    long            validInfo;
    float           point[3];
    float           normal[3];
    float           color[3];
    string          label;
    float           localTransform[4, 4];
    float           globalTransform[4, 4];

    cxPickObject    objectType;
    switch (objectType) {
    case cx_pick_point:
        long            index;
    case cx_pick_line:
        long            index[2];
        float           point[2, 3];
        float           normal[2, 3];
    case cx_pick_face:
        long            nvertices;
        long            index[nvertices];
        float           point[nvertices, 3];
        float           normal[nvertices, 3];
    case cx_pick_cone:
        long            part;
    case cx_pick_cylinder:
        long            part;
    } o;
} cxPickHit;
```

These are the variables:

id	Gives an ID for the module that generated the picked image or object. This field allows the module that generated the picked geometry to identify it. For example, if a module generates geometry and then gets back pick information, it can compare the pick ID with its own ID to see if they match. If not, it can ignore the information.
validInfo	A bit vector that indicates whether the pick information in the following six fields is valid. The fields are all bit fields in *validInfo*.
point	Defines a point which gives the placement of an object. *DisplayImg* returns coordinates (x, y) in image space and *Render* in 3D world space (x, y, z). The coordinates are floating point values.
normal	Gives the surface normal (if available) through the API routine *cxGeoNormalAdd*.
color	Gives the color, if it has one. The color of an object in *Render* is governed by an upstream module, which may not necessarily send any color value down. In this case, *Render* gives the

object a default color, but the hue bears no relation to the data values at all. You can use the API routines *cxGeoColorAdd* or *cxGeoABGRAdd* to set a color.

You can also use the Open Inventor nodes SoMaterial, SoBaseColor, and SoPackedColor, or the IRIS Explorer Inventor nodes cxSoBaseColor and cxSoPackedColor, if you are working with geometry.

label — Names the object by its label name in the geometry scene graph. *DisplayImg* does not use the label field. You can use *cxGeoLabelAdd* or add the label in Open Inventor.

localTransform — Gives the transform matrix if a geometry object has been translated or rotated in the *Render* module that generated the pick event. Upstream modules don't know about local transforms. You can use the API routines *cxGeoRotate*, *cxGeoTranslate*, and *cxGeoScale* to perform the transform.

globalTransform — Gives the transform matrix if the object has been translated or rotated anywhere in the module chain. This includes transforms from upstream modules. It is a camera transform.

Note: Not all fields are applicable to all objects. The bit masks for each field in *validInfo* are in the *cxPickValid* enumeration in */usr/explorer/include/cx/Pick.h*

objectType — Tells you what kind of object sustained the hit. For some objects, including points, lines, faces, cones, and cylinders, you can collect information about indices, normals, vertices, and parts.

Figure 6–2 shows the structure of the *cxPickHit* data type.

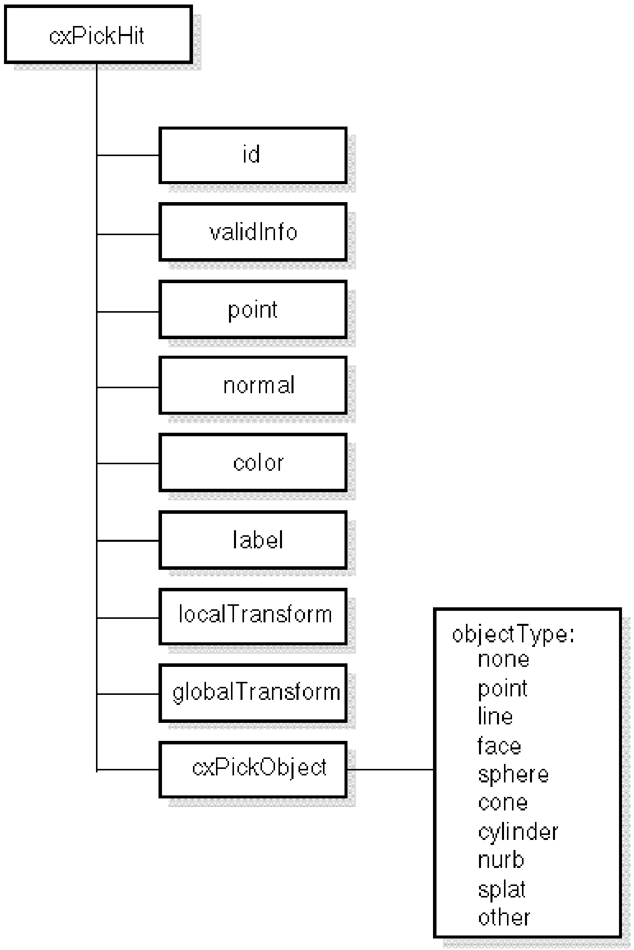


Figure 6–2 Schematic Structure of the PickHit Data Type

Labeling Items in Scene Graphs

The *QueryLat* module is an example of how to label items in a scene graph.*QueryLat* responds to pick mouse button events by determining the lattice value at the first pick intersection and creating a label, which is text geometry, with that value. The value passes into *Render* through its annotation port and is displayed at the correct location in the Render window.

Figure 6–3 shows the "pick" map, which contains *QueryLat*. The map is distributed with IRIS Explorer and can be launched from the Module Librarian.

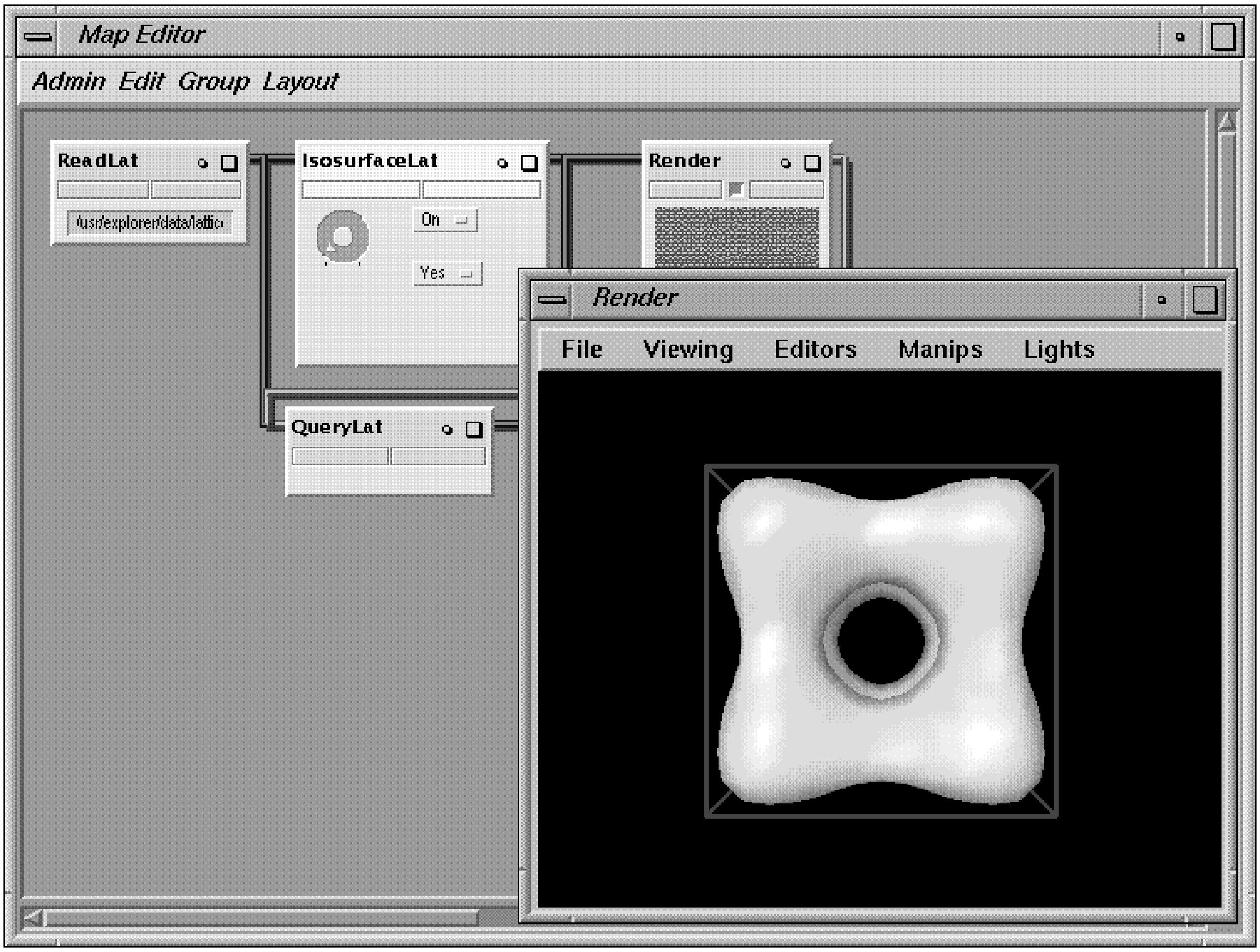


Figure 6–3 The Pick Map

QueryLat takes as input a lattice and the pick data from the *Render* module. The input port of *QueryLat* has connections from *ReadLat* and the *Render* pick output. Its output port is wired into the *Render* "Annotations" input port. When you hold down the mouse button (in User Pick mode) on a point in the Render window, the value of the lattice at that point is displayed. If you pick on an empty part of the window, the value is null and no annotation is plotted.

Gathering Shape Information.

You can collect more information about certain types of hit objects from the *Render* module:

points	you can determine the index of the point that was hit
lines	you can collect the indices of end−points and the normals at the end−points (if they exist)
polygons	you can determine the number of sides, and the indices, locations, and normals of the vertices (3D only)
cones and cylinders	*Render* outputs a part number, taken from Inventor, which indicates whether you picked the side, base, or top of the object. For cylinders, the part numbers are: side = 1 top = 2 bottom = 4 For cones, the part numbers are: side = 1 bottom = 2

6.3. The Data Type Declaration

The *cxPick* data type, though defined in the IRIS Explorer typing language, can be considered as a C structure. Fortran users need to set pointers to the data type structures when they use them. The type declaration resides in the header file */usr/explorer/include/cx/cxPick.h*.

```
typedef enum {
    cx_pick_none,
    cx_pick_point,
    cx_pick_line,
    cx_pick_face,
    cx_pick_sphere,
    cx_pick_cone,
    cx_pick_cylinder,
    cx_pick_nurb,
    cx_pick_splat,
    cx_pick_other
} cxPickObject;

typedef struct cxPickHit {
    long              id;
    long              validInfo;
    float            *point;
    float            *normal;
    float            *color;
    char             *label;
    float            *localTransform;
    float            *globalTransform;
    cxPickObject      objectType;
    union {
        struct {
            long               index;
        } cx_pick_point;
        struct {
            long             *index;
            float            *point;
            float            *normal;
        } cx_pick_line;
        struct {
```

```
            long              nvertices;
            long             *index;
            float            *point;
            float            *normal;
            } cx_pick_face;
        struct {
            long              part;
        } cx_pick_cone;
        struct {
            long              part;
        } cx_pick_cylinder;
    } o;
} cxPickHit;

typedef struct cxPick {
    cxDataCtlr       ctlr;
    int               eventType;
    long              eventData;
    long              eventModifiers;
    int              *position;
    int              *windowSize;
    float            *origin;
    float            *direction;
    float            *projectionMatrix;
    int               numHits;
    cxPickHit        *hit;
} cxPick;
```

The Fortran type enumerations reside in */usr/explorer/include/cx/cxPick.inc*.

```
INTEGER cx_pick_none
INTEGER cx_pick_point
INTEGER cx_pick_line
INTEGER cx_pick_face
INTEGER cx_pick_sphere
INTEGER cx_pick_cone
INTEGER cx_pick_cylinder
INTEGER cx_pick_nurb
INTEGER cx_pick_splat
INTEGER cx_pick_other
PARAMETER (cx_pick_none                   = 0)
PARAMETER (cx_pick_point                  = 1)
PARAMETER (cx_pick_line                   = 2)
PARAMETER (cx_pick_face                   = 3)
PARAMETER (cx_pick_sphere                 = 4)
PARAMETER (cx_pick_cone                   = 5)
PARAMETER (cx_pick_cylinder               = 6)
PARAMETER (cx_pick_nurb                   = 7)
PARAMETER (cx_pick_splat                  = 8)
PARAMETER (cx_pick_other                  = 9)
```

6.4. The Pick API Routines

You can use the API (Application Programming Interface) routines to manipulate the pick data type in IRIS Explorer.

They are described in detail in the *IRIS Explorer Reference Pages* and on−line in the *man* pages. They let you:

- extract part of the data collected by the data structure
- extract details about a specific object that has been hit
- create a new instance of the *cxPick* data structure

Table 6−1 lists the subroutines and briefly describes the purpose of each one.

Subroutine	Purpose
cxPickGet	Gets information from a *cxPick* data structure
cx PickHitGet	Gets information from a *cxPickHit* data structure
cxPickHitConeGet	Gets detailed information about a hit on a cone
cxPickHitCylinderGet	Gets detailed information about a hit on a cylinder
cxPickHitFaceGet	Gets detailed information about a hit on a polygon
cxPickHitLineGet	Gets detailed information about a hit on a line
cxPickHitPointGet	Gets detailed information about a hit on a point
cxPickNew	Creates a new *cxPick* data structure

Table 6−1 Pick Subroutines

6.5. Code Examples

This section includes examples of user functions for modules that read and write pick data

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

.

6.5.1. Reading Pick Data

The following function lists the number of hits, looks at the list, and builds a 1D lattice from the data in the list. The C code and resources file may be found in directory */usr/explorer/src/MWGcode/Picking/C*. There is currently no equivalent Fortran module.

```
#include <cx/DataAccess.h>
#include <cx/cxLattice.h>
#include <cx/cxPick.h>
#include <cx/cxPick.api.h>

void readpick(cxPick *pick,cxLattice **lattice)
{
  cxErrorCode ec;
  float *points, p1[3];
  int i, j;
  long *data, dims[1], index;

  /* Do not generate a lattice if the pick did not hit anything */
  dims[0] = cxPickNumObjectHitsGet(pick, &ec);
  if (dims[0] == 0)
    return;

  /* Create a 1D curvilinear output lattice
   * of type long with 3 coordinates and dims[0] nodes
   */
  *lattice = cxLatNew(1, dims, 1, cx_prim_long, 3,
        cx_coord_curvilinear);
```

```
        cx_coord_curvilinear);

  /* Get the pointer to the coordinate and data values */
  cxLatPtrGet(*lattice,NULL,(void **)&data, NULL, (void **)&points);

  /* Loop over the hits */
  for (i = 0; i < dims[0]; i++)
    {
      ec = cxPickHitGet(pick, i, &index, NULL, p1, NULL, NULL, NULL,
   NULL, NULL, NULL);

      /* Get the lattice coordinates */
      for(j = 0; j < 3; j++)
 points[i*3+j] = p1[j];

      /* Set the lattice data value to be the pick identifier */
      data[i] = cxPickHitIdGet(pick, i, &ec);
    }
}
```

6.5.2. Writing Pick Data

Here is a user function based on the user function for the *QueryLat* module. Seeing how the *cxPick* data type is used here may help you in writing your own pick module. The C code and resources file may be found in directory */usr/explorer/src/MWGcode/Picking/C*. There is currently no equivalent Fortran module.

```
#include <stdio.h>

#include <cx/PortAccess.h>
#include <cx/DataTypes.h>
#include <cx/DataAccess.h>
#include <cx/UserFuncs.h>
#include <cx/Geometry.h>
#include <cx/Lookup.h>

#include <cx/Pick.h>

long pick()
{
  cxGeometry *geo;
  cxLattice *lattice;
  cxPick *p;
  int lport;

  static cxLookup *lut = NULL;
  static cxPrimType ltype = cx_prim_byte;
  static int flag = 1;

  if ( flag )
    {
      flag = 0;
      cxGeoInit();
    }

  /* allocate new geometry object */
```

```
  /* allocate new geometry object */
  geo = cxGeoNew();
  cxGeoBufferSelect(geo);

  /* delete everything */
  cxGeoRoot();
  cxGeoDelete();

  /* if this is a new lattice, create the lookup table */
  lport = cxInputPortOpen("Lattice");
  if ( cxInputDataChanged(lport) )
    {
      if (lut)
 cxLookupDestroy(lut);
      lattice = (cxLattice *) cxInputDataGet(lport);
      lut = cxLookupCreate(lattice, cx_lookup_linear);
      ltype = lattice->data->primType;
    }

  /* get the pick information */
  p = (cxPick *) cxInputDataGet(cxInputPortOpen("Pick"));

  if (p->eventType == CX_PICK_BTNDOWN ||
      p->eventType == CX_PICK_MOTION)
    {
      if ( p->numHits > 0 )
 {
   /* do a lookup */
   switch (ltype)
     {
#define CASE(ctype,type,format) \
     case ctype: { \
     type value; \
     if ( cxLookupInterp(lut,p->hit[0].point,&value) ) \
     cxGeoTextDefine("domain error", \
       NULL); \
     else { \
     char str[128]; \
     sprintf(str,format,value); \
     cxGeoTextDefine(str, \
       NULL); \
          } \
   } \
       break;
       CASE(cx_prim_byte,char,"%d")
  CASE(cx_prim_short,short,"%d")
    CASE(cx_prim_long,long,"%d")
      CASE(cx_prim_float,float,"%f")
        CASE(cx_prim_double,double,"%f")
#undef CASE
     }
#ifdef ASDF
   static float point[3] = { 0.0, 0.0, 0.0};
   static float radius = 1.0;
   cxGeoSpheresDefine(1,(float *)point,&radius);
```

```
   cxGeoSpheresDefine(1,(float *)point,&radius);
#endif
   cxGeoTranslate(p->hit[0].point[0],
    p->hit[0].point[1],
    p->hit[0].point[2]);
 }
    }

  /* close the buffer and output */
  cxGeoBufferClose(geo);
  cxOutputDataSet(cxOutputPortOpen("Annotation"),(void *)geo);

  return 0;
}
```

Chapter 7

Using the Parameter Data Type

This chapter describes the *cxParameter* data type and how it is used in the IRIS Explorer environment. It defines the data type and lists the API (Application Programming Interface) routines available for manipulating it. Code examples are also included.

7.1. Understanding Parameters

The *cxParameter* data type is a simple data type that holds a single scalar value. Its purpose is to pass scalar values between modules in one of three forms, or primitive types. You can use the *cxParameter* API routines to extract the data values to be passed.

For example, a module that computes the minimum and maximum of a lattice may produce two *cxParameter* outputs, one containing the lattice minimum value and the other containing the lattice maximum value. The API routines also do automatic type coercion to one of the permissible types (if possible).

This is the type definition:

```
root typedef struct {
    cxParamType   type              "Type";
    switch (type) {
          case cx_param_long:
                  long    val       "Value";
          case cx_param_double:
                  double  val       "Value";
          case cx_param_string:
                  string str        "Value";
    } p;
} cxParameter;
```

These are the variables:

type	The primitive data type: one of long integer, double−precision floating point, or character string (null−terminated).
val	The value of the primitive data type.

Figure 7−1 depicts the structure of the *cxParameter* data type.

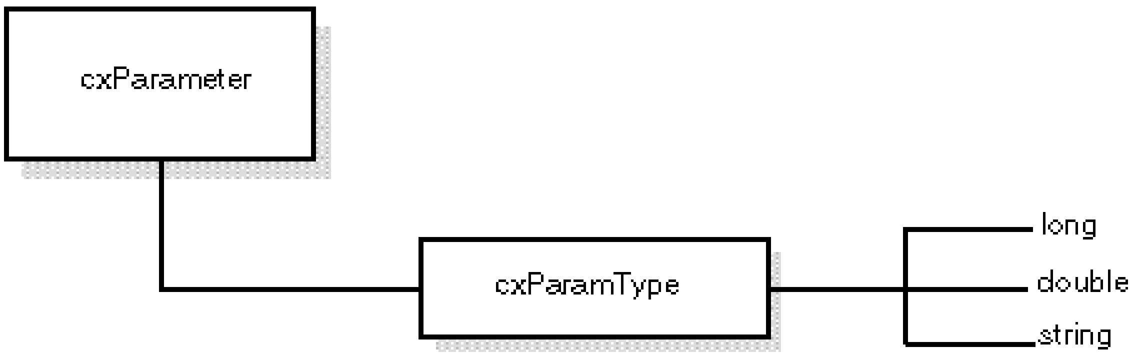


Figure 7–1 Schematic Structure of the Parameter Type

Control Panel Widgets

It is easy to confuse the *cxParameter* data type, which is a root data type and operates just like *cxLattice* or *cxPyramid*, with the parameter values that are passed by widgets on the module control panels. The widget happens to use the parameter data type to pass new values, but modules can also pass parameters without using widgets. The widget is merely a way of making the system more flexible in passing changes to parameter values.

7.2. The Data Type Declaration

The *cxParameter* data type, though defined in the IRIS Explorer typing language, can be considered as a C structure. Fortran users need to pass a pointer to the data structure if they use it. The type declaration is contained in the header file */usr/explorer/include/cx/cxParameter.h.*

This is the *cxParameter* data type declaration:

```
typedef enum {
    cx_param_long,
    cx_param_double,
    cx_param_string
} cxParamType;

typedef struct cxParameter {
    cxDataCtlr       ctlr;
    cxParamType       type;
    union {
        struct {
            long             val;
        } cx_param_long;
        struct {
            double           val;
        } cx_param_double;
        struct {
            char            *str;
        } cx_param_string;
    } p;
} cxParameter;
```

7.2.1. Fortran Type Enumeration

These are the Fortran type enumerations for the parameter type. They are listed in the file */usr/explorer/include/cx/cxParameter.inc*

```
INTEGER cx_param_long
INTEGER cx_param_double
INTEGER cx_param_string

PARAMETER (cx_param_long                  = 0)
PARAMETER (cx_param_double                = 1)
PARAMETER (cx_param_string                = 2)
```

7.3. Using the Parameter API

This section lists the API (Application Programming Interface) routines for *cxParameter*, which are described in detail in the *IRIS Explorer Reference Pages* and on−line in the *man* pages. They allow you to:

- create new *cxParameter* data structures
- create different types of parameters
- set scalar and string parameters
- duplicate parameters
- extract parameter values

Table 7−1 lists the subroutines and briefly describes the purpose of each one.

Subroutine	Purpose
cxParamNew	Creates a new *cxParameter* data structure
cxParamDoubleNew	Creates a parameter of type double
cxParamLongNew	Creates a parameter of type long
cxParamStrNew	Creates a character string parameter
cxParamTypeSet	Sets the parameter type
cxParamLongSet	Sets the parameter to an integer value
cxParamDblSet	Sets the parameter to a double value
cxParamStrSet	Sets the parameter to a string value
cxParamDup	Creates a duplicate of the parameter
cxParamDblGet	Returns the parameter value as a double
cxParamPathnameGet	Expands a parameter string value into a UNIX pathname
cxParamLongGet	Returns a parameter value as `long`
cxParamStrGet	Returns a parameter value as a string
cxParamTypeGet	Returns a parameter type

Table 7−1 Parameter Subroutines

7.4. Code Example

Here is an example of how to use the *cxParameter* data type. This fragment shows how to define each of the three primitive types.

```
cxParameter *p;
  p = cxParamNew ( );

   cxParamDblSet  (p, 3.1415926);
   cxParamLongSet (p, 17);
   cxParamStrSet  (p, "Hello, world.");
```

Chapter 8

Creating User−defined Data Types

This chapter describes how to create your own data types for handling data that does not fit one of the existing data types. The concept of the "user−defined" data type is discussed and the syntax for the IRIS Explorer data typing language (ETL) is laid out.

The chapter provides examples of the typing language, as well as the specification for a new data type and code for modules that can use it.

8.1. Overview

All the data that IRIS Explorer manipulates must fit into a recognized IRIS Explorer data type. IRIS Explorer has several built−in data types that satisfy the needs of nearly all users and fit most scientific data; however, sometimes the existing IRIS Explorer data types do not adequately describe the data you want to visualize, hence the user−defined data type (UDT), designed specifically for the task at hand. For example, one might define a new data type for quantum dynamic calculations on molecules, along with a suite of modules to do the calculations and create geometric representations suitable for use by the other modules in IRIS Explorer.

IRIS Explorer provides a data typing language (ETL) for defining data types that can be passed among modules. Such data types are called root types. Using ETL, you can create a new root data type for use in custom−built IRIS Explorer modules, in the Map Editor, and in the Module Builder. The IRIS Explorer data types *cxLattice*, *cxParameter*, *cxPyramid*, *cxGeometry*, and *cxPick* were created using ETL and are examples of how to use ETL.

For a new data type to work in IRIS Explorer, it must have a C structure definition that can be used to build modules, as well as a description of the type that IRIS Explorer and the Module Builder can load at run−time. It is also useful to have a library of accessor functions that allow C, C++, and Fortran users access to the type with a minimum of programming. IRIS Explorer allows you to create all these things.

When you create a new data type, you have to define all aspects of the structure that IRIS Explorer may need to know at any time. You also need to write new modules to accept, process, and output the data conveyed in the new data type, so the design of your data type should be carefully thought out.

Most users will find that the standard IRIS Explorer data types are sufficient for their needs. Furthermore, a large number of modules are already built to handle the standard data types. For those with data from specialized domains, the effort to define a new data type is relatively small; however, the true cost of using a new type effectively is considerably greater because of the larger investment in module writing.The creation of new types, therefore, is best suited to those users already intending to write a suite of new modules.

8.2. Creating a Data Type

IRIS Explorer data types consist of a root data type and, optionally, one or more subsidiary data types. The root data type is the named structure that appears as a data type on module input and output ports and can be passed between modules.

The subsidiary data types expand the functionality of the root data type but cannot stand alone on a port. For example, *cxConnection* and *cxPyramidDictionary* are subsidiary data types used in defining the root data type, *cxPyramid.* All these type definitions are collected together in a type declaration file.

8.2.1. Type Declaration File

Data type descriptions reside in type declaration files, which end with the suffix *.t* (for "type"). You create and name the type declaration file when you define the new data type.

This is the basic structure of a *modulename.t* file:

1. Any necessary include statements for data types defined elsewhere:

```
#include <---->
#include <---->
...
```

2. The root type definition:

```
root typedef struct {
..
} RootTypeName
```

3. Subsidiary type definitions:

```
shared typedef struct {
...
} Name1
closed typedef struct {
...
} Name2
typedef struct {
...
} Name3
```

The keywords used in each type definition are defined in **"Syntax"** later in this Chapter.

As can be seen from this outline of the *modulename.t* file contents, the root type is a `struct`, similar to a C structure. The building blocks of the type include other `structs`, enumerations, unions, and arrays of any of these.

8.2.2. Naming Files and Data Types

All data types have names. The naming conventions are similar to those in C, that is, the first character must be a letter, but the rest of the name may contain digits and the underscore (_). Data type names may be of any length and case is significant. The type filename and data type name must agree. For example, if the data type is to be called *myCylinder*, the file must be named *myCylinder.t*. If the file is named *bxMyCylinder.t*, the data type must be called *bxMyCylinder*.

A single type declaration file may contain several type declarations if the root data type contains subsidiary data types. The order of data type declarations in the file is not important; they are sorted during the translation into C.

8.3. Using the ETL

The IRIS Explorer data typing language resembles the structure definition syntax of C in many respects. It uses the constructions `typedef` and `struct`, and a construction similar to union, called `switch`.This section takes an in−depth look at the typing language, including the definition of its syntax and some examples.

The presentation of the syntax follows the format used in *The C Programming Language*, Second Edition, by B.W. Kernighan and D.M. Ritchie (Prentice−Hall, 1988). The syntax and some discussion of the restrictions placed on forming compound types follow.

Conventions

These conventions apply in this chapter:

- Syntactic object categories are indicated by *italics*, for example, *integer_value*.
- Literal words and characters such as IRIS Explorer keywords are indicated by a font as follows: `closed`.

- Alternate categories are listed sequentially on separate lines.
- Sets of narrow alternates are listed on one line, preceded by the phrase `one of`.
- An optional symbol has the suffix $_{opt}$.
- The double period `..` denotes one of the range of values in the ASCII collating sequence; for example, `a..z` indicates one of the range of characters `a,b,c,..,z`.
- The triple period ... denotes a repeated syntactic object, for example, *case* ....

8.3.2. Syntax

There is a small set of special keywords used in defining an IRIS Explorer data type.

`root`	Indicates a data type accessible by module input and output ports, and therefore transportable from module to module. For example, the IRIS Explorer data types *cxLattice*, *cxGeometry*, *cxPick*, *cxPyramid* and *cxParameter* are all root data types. `root` modifies a `typedef struct` definition. Root structures are reference−counted.
`shared`	Modifies an instance of a subsidiary structure used in creating another, containing structure. It indicates that the structure should have a reference count associated with it. For example, the *cxData*, and *cxCoord* structures, contained within *cxLattice*, are reference−counted.
`closed`	Used to hide the contents of a structure when the type is displayed in the Module Builder connections window. It can be used either as a modifier to a `typedef` or a member of a `struct`. It is used in recursive data types, for example the recursive loop in *cxPyramid.* It can also be used for moving the data in a structure intact through a module, for example, *cxGeometry.*
`typedef`	Defines a new type name.
`struct`	Indicates a structure definition. A structure is a collection of named members of different kinds.
`switch`	Indicates a discriminated union. A discriminated union is a union structure for which another variable (the discriminator) determines which variant of the union is active.
`case`	Indicates one variant of a discriminated union.
`enum`	Creates a new set of enumerated constants.

There are special language symbols for a few of these keywords.

Other keywords are used in−place in subsequent sections of the language definition.

	Keyword
port_modifier:	`root`
reference_modifier:	`one of root shared`
modifier:	*port_modifier, reference_modifier*
closed:	`closed`

8.3.3. Rules for Type Definitions

Declarations in the IRIS Explorer typing language (ETL) are more constrained than in C. These are the basic rules for data type definitions:

- Comments may be placed anywhere in a type definition file, enclosed by `/*` and `*/` (see **"Inserting Comments"** later in this Chapter).
- DISCRIMINATOR and CONSTANT in a *switch_type* statement must be of the same enumerated type (see **"Composition Rules"** later in this Chapter).
- BOUND must be a positive integer scalar or an array of positive integers (see **"Array Dimensioning"** later in this Chapter).
- IRIS Explorer module ports may have any type that is a *reference_type* (see **"Defining a New Type"** later in this chapter).
- Shared memory types may be defined in any order.

8.3.4. Naming Conventions

There is a a standard naming convention for data types defined by IRIS Explorer. A standard scheme makes it easy to recognize the different forms of names used in a type and to identify their purpose.

Here are the standard IRIS Explorer conventions:

Type names

In a concatenation of words, each word after the first is capitalized. No underscores are used, and the leading word or prefix is not capitalized. For example, here are type names for a data and a pyramid dictionary type:

```
typedef struct {  } cxData;
typedef struct {  } cxPyramidDictionary;
```

Member names

A concatenation of words capitalizes each word after the first. No underscores are used, and the leading word or prefix is not capitalized. For example, here are type names for an *nDim*, a *dims*, and a *primType* variable:

```
long         nDim;
long         dims;
cxPrimType   primType;
```

Enumeration values

Words are all lowercase and separated by underscores. The enumerated values typically contain the enumeration type name as a prefix.
For example, here are the enumerated values for enumeration types *cxCompressType* and *cxPrimType*:

```
typedef enum {
        cx_compress_none,
        cx_compress_unique,
        cx_compress_multiple
} cxCompressType;

typedef enum {
        cx_prim_byte,
        cx_prim_short,
        cx_prim_long,
        cx_prim_float,
        cx_prim_double,
        cx_prim_string
} cxPrimType;
```

8.3.5. Scalar Types

These are the basic scalar types that IRIS Explorer will accept.

Scalar Type	C	Fortran
char	char type	
short	short type	
int	int type	integer type
long	long int type	integer type
float	float type	real type
double	double type	double precision type
string	null–terminated string	character type (converted to null–termination)

Table 8–1 Scalar Types in IRIS Explorer

The grammar fragments are:

	Scalar Type
base_integer	`char, short, int,` or `long`
floating	`one of float double`
string	`string`

Signing Integer Types

There are also sign modifiers for the *base_integer* types, namely:

- signed
- unsigned

Signing Other Scalar Types

According to the C language specification, scalars of type `short`, `int`, and `long` whose type is not specified are considered signed. Also, in IRIS Explorer `char` scalars whose type is not specified are considered unsigned, independent of the local compiler interpretation of `char`. For example, on Silicon Graphics IRIS computers, the C compiler treats `char` as unsigned, while the C++ compiler treats `char` as signed.

IRIS Explorer typing forces `char` to unsigned `char` so that both languages treat an IRIS Explorer `char` variable identically.

sign	`one of signed unsigned`
integer	$sign_{opt}$ *base_integer*
simple_type	*integer, floating, string*

8.3.6. Enumerated Constant Types

You can define, as a new type, an enumeration of consecutive named constants starting at zero.

enum_type `enum` { *name* } `enum` { *name*, *name* ... }

8.3.7. Composition Rules

An IRIS Explorer type includes structures composed of the basic scalar types in three forms:

- array
- switch (union)
- structure

The composition rules may be applied both to scalars and to other composed types. Thus, you may construct an array of integers, and also an array of enumerated constants, structures, unions, or arrays. The language syntax for these elements is shown below.

Struct Structures

`struct` is an IRIS Explorer structure, almost identical in syntax to the C structure into which it is transformed. It looks like this:

struct_type `struct {` *member ...* `}` (member is defined below)

You must define switch discriminator and array subscript variables within the same `struct` definition as the *switch* and *array* variables to which they refer.

Switch Structures

`switch` is an IRIS Explorer union, declared as a named structure, and indicating a discriminator member that can indicate which variant of the union to consider at run−time.

The structure looks like this:

switch_type	`switch (` DISCRIMINATOR `) {` *case* ... `}`
case	`case`CONSTANT *:* *member* ...

DISCRIMINATOR is a *name* of type *enum_type*; *CONSTANT* is a *name* that is a possible value of type *enum_type*. For example, *cx_prim_type*.

8.3.8. Defining a New Type

In C, it is possible to declare a structure without defining that structure as a new type. In IRIS Explorer, you must name all new types in order to declare members of that type. The `typedef` statement defines a new type from a simple type, a `struct`, or an enumeration. It can also be used to create a new type from a previously created type, in effect creating a pseudonym for an existing type. These operations are defined below:

abstract_type	*simple_type* *struct_type* *enum_type*
named_type	STANDARD_TYPE
reference_type	REFERENCE_TYPE
defined_type	*simple_type* *switch_type* *named_type* *reference_type*
standard_definition	
	closed$_{opt}$`typedef` *abstract−type type_name*;
reference_count_definition	
	closed$_{opt}$*modifier* `typedef`*struct_type type_name*;

STANDARD_TYPE is a *name* of type *standard_definition;*REFERENCE_TYPE is a *name* of type *reference_count_definition*

8.3.9. Including Other Type and Header Files

Both C header files and other IRIS Explorer type files (with a *.t* suffix) may be included into a type file using the standard C preprocessor syntax, which uses a # character in the first column. If a file *TypeA.t* needs definitions from file *TypeB.t*, it must include *TypeB.t* in a `#include` statement.

Inclusion of an IRIS Explorer type file has the effect of incorporating the included file into the current typing file. This allows you to reuse previously defined types; for example, the *cxPyramid* type uses the *cxLattice* type. Inclusion of a *modulename.t* file also means the corresponding *modulename.h* file will be included in the resulting C header file.

Inclusion of a C header file has no effect on the actual type processing, but only serves to put the same include statement into the resulting C header file.

FILENAME is a *name* referring to a file in the file system, located in *$EXPLORERHOME/include/cx* or in the current directory.

include	`#include<FILENAME>` `#include"FILENAME"`

8.3.10. Structures

There are two forms of structures in the IRIS Explorer typing language, in−line and reference−counted structures. Reference−counted structures are denoted by the `root` and `shared`keywords in the typing language. Both keywords imply reference−counting, but `root` has other meanings as well, described below.

In−line Structures

In−line structure members reside contiguously with the preceding and following members in the enclosing structure.

Thus, the following example creates a `twoInt` structure named *two* with two members, *two.a* and *two.i.b*:

```
 typedef struct {
       int b;
 }oneInt;

typedef struct {
      int a;
      oneInt i;
 } twoInt;

twoInt two;
```

The actual storage layout of two is contiguous, with member a followed by `i.b`. Member names in this example are `two.a` and `two.i.b`.

The grammar fragment is:

standard_definition

$\quad$ *closed*$_{\text{opt}}$`typedef` *abstract−type type_name*;

Reference−counted Structures

Reference−counted structure members do not reside contiguously with the preceding and following members in the enclosing structure. Rather, the entire structure occupies its own section of memory and is referenced by a pointer. The following example creates a `twoFloat` structure named *two* with two members, *two.a* and *two.i−>b*:

```
shared typedef struct {
       float b;
} oneFloat;

typedef struct {
        float a;
        oneFloat i;
} twoFloat;

twoFloat two;
```

The actual storage layout of two is the value of a followed by the address of an instance of a oneFloat. Member names in this example are `two.a` and `two.i->b`.

The grammar fragment is:

reference_count_definition

$\quad$ *closed*$_{\text{opt}}$ *modifier*`typedef` *struct_type type_name*;

For more on reference counting, see **"Understanding Reference Counting"** in Chapter 9.

Differences Between Structures

There are two differences to be noted in accessing the in−line and the reference−counted structures.

The first difference is that a `shared`member will always be a pointer to a structure, rather than an in−line structure. An array of a `shared` type is really an array of pointers to shared structures.

The following example shows a structure member *two* with two members, *two.n* and *two.i*, where the several double precision numbers in *two* are *two.i[j]−>b*, for values of *j* between 0 and n−1:

```
    shared typedef struct {
           double b;
    } oneDouble;

    typedef struct {
            int   n;
    oneDouble i[n];
  }  doubleArray;

   doubleArray two;
```

The second main difference, and the reason why there is a distinction between in−line and reference−counted of structures, is that the reference−counted structure has some additional information that allows the structure to be shared between two or more data sets. Because the reference−counted structure is referred to by a pointer, several data sets can hold the same pointer: a single update of the shared data affects all data sets. It is impossible for two or more structures to share a single in−line structure.

Note: The user should not change values within the reference−counting structure, nor rely on the form or content of that structure, as it is subject to change in subsequent releases of IRIS Explorer.

Examples of Reference−counted Structures

The *cxData* and *cxCoord* types are implemented within a *cxLattice* as reference−counted structures. This means that several lattices could share data or coordinates, for example to allow the same data values to be mapped simultaneously onto several coordinate mappings without duplicating the (potentially large) set of data.

In addition to the reference−counted meaning of the `root` keyword, `root` also implies that the type is visible as the type of an IRIS Explorer module's port. For example, *cxLattice* is a root type and is a legal port type, but *cxData* (contained in *cxLattice*) is a `shared` type, but not a `root` type, and is thus not a legal port type. Hence `root` implies `shared`, but `shared` does not imply `root`.

8.3.11. Structure Member Declaration

At most, one variable may be declared in each *member* statement. This is distinct from the C style, where multiple variables of the same type may be declared together. The grammar fragment is:

simple_type VAR;

For example, this is legal in IRIS Explorer:

```
int a;        /* Legal */
int b;
```

but this is not:

```
int a, b;     /* Illegal */
```

The most general form of a member declaration is:

member	$closed_{opt}$*defined_type VARarray_specifier*$_{opt}$*label*$_{opt}$;
	$closed_{opt}$*switch_type VAR array_specifier*$_{opt}$*label*$_{opt}$;

However, the example

simple_type VAR;

is actually much simpler than the general form. It has no `closed` keyword, uses an integer *simple_type*, is not an array, and has no label.

Other simple−typed members include:

```
int a;
unsigned long b[n] "Value of B";
closed   int c;
float    d[ a, a, 3 ];
double   e    "Real Number E";
```

Table 8−2 lists the parts of a simple−typed member.

simple_type	Name	Bound	Label
unsigned long	b	[n]	"Value of B"

Table 8−2 A Simple−typed Member

You could also use common IRIS Explorer types in this example, where you use other *defined_type* types:

```
cxLattice a;
cxLattice b[10];
closed cxLattice c "Not visible in Module Builder";
```

8.3.12. Array Dimensioning

IRIS Explorer arrays can be dimensioned by a combination of integer constants, scalar integer variables, and arrays of integers. It is important to understand how the array declaration in the IRIS Explorer typing language translates into an array of bytes in memory and how to access that memory in C, C++, or Fortran.

All IRIS Explorer arrays are represented by 1−D arrays of memory. The indexing into this long array depends on the order of the array bounds in the array declaration. Array bounds are given in the normal Fortran order, so that the first bound represents the fastest varying index. For example, the following 3−D array *ThreeD* has shape *m* by *n* by *p*, with the *m*−based index varying fastest:

```
int m;
int n;
int p;
int ThreeD[ m, n, p ];
```

The equivalent Fortran declaration is:

```
integer ThreeD(m, n, p)
```

The equivalent C or C++ declaration is:

```
long ThreeD[p, n, m];
```

Incorrect: this is illegal in C, but gives the sense.

```
long ThreeD[p][n][m];
```

Incorrect:the generated array has only one dimension, not three as suggested here.

```
long ThreeD[p*n*m];
```

Correct: this is the size. You need to know m varies fastest.

Arrays can have integer constants or scalars as bounds. The constant or the value of the variable is used at run−time to determine the array length. Arrays can also have other arrays as dimensioning variables, as in the data array of*cxLattice* ;

```
long      nDim;
long      dims[nDim];
double    values[nDataVar, dims];
```

In this case, the array contents are treated as a list of scalar values, so that the previous example is equivalent to the following one for `nDim = 3`:

```
long      dims[3];
double    values[ nDataVar, dims[0], dims[1], dims[2] ];
```

Thus the length of an array can be calculated as the product of all of its dimensioning bounds, where a dimensioning array (for example, *dims*) has a product equal to the product of its integer contents.

array_bound   *positive_integer_value*
BOUND

BOUND is a *name* of type *integer* or an *array* of type *integer*

array_specifier   [*array_bound*]
[*array_bound*, *array_bound* ...]

member   *closed*$_{opt}$ *defined_type VAR array_specifier*$_{opt}$ *label*$_{opt}$;
closed$_{opt}$ *switch_type VAR array_specifier*$_{opt}$ *label*$_{opt}$;

8.3.13. Assigning Labels

Most members in a type should be assigned labels. Labels are used in the creation of automatically generated API routines for programmatic access to the data types. All members that are to be accessed through this API must have labels. Labels are also used in the Module Builder Connections window for data type member wiring.

Members that can get labels include scalars, arrays, and reference−counted structures. Members that do not get labels include in−line structures (see below), switch structures, and type definitions.

The author of the type file may assign labels to structure members. The label is a quoted text string following the variable name and optional array bounds. **Table 8−3** shows how the scalars are labeled in a lattice.

Scalar Type	Name[Bounds]	Label
long	nDim	"Num Dimensions";
long	dims[nDim]	"Dimensions Array";

Table 8−3 Labels on Lattice Scalars

```
member
  closed(opt) defined_type VAR array_specifier(opt) label(opt);
  closed(opt) switch_type VAR array_specifier(opt) label(opt);
```

Constructing Labels

Most members of an IRIS Explorer structure are given a text label for use elsewhere in IRIS Explorer, for example, in creating automatically generated API routines. If the user does not supply a label on a member which requires a label, the IRIS Explorer type compiler assigns a default label computed from the member name.

IRIS Explorer makes some effort to ensure that no two assigned labels are identical, but does not try to avoid collisions between automatically generated labels and user−supplied ones.

blank
self defining

tab
self defining

underscore
– (Note: Do not overlook it.)

alpha   one of
`A..Z  a..z`

numeric
`0..9`

non_zero

`1..9`

character
: *text*
: *whitespace*

text
: *alpha*
: *numeric*
: *underscore*

whitespace one of
: *blank* *tab*

label
: `"`*character ...* `"`

integer_value
: *numeric*
: *numeric ...*

positive_integer_value
: *non_zero*
: *non_zero numeric*
: *non_zero numeric ...*

name
: *alpha*
: *alpha text*
: *alpha text ...*

member
: *closed*$_{opt}$*defined_type VAR array_specifier*$_{opt}$*label*$_{opt}$;
: *closed*$_{opt}$*switch_type VAR array_specifier*$_{opt}$*label*$_{opt}$;

VAR is an item of *name*.

8.3.14. Inserting Comments

IRIS Explorer type files use the standard C comment syntax, which is a block of text surrounded by the delimiters `/*` and `*/`. Comments may be inserted anywhere in the file.

comment `/*` *character ...* `*/`

8.4. Differences between ETL and C

The IRIS Explorer typing system lacks certain features that C structure definitions offer. There are three obvious differences, discussed below.

8.4.1. No Separating Comma

The comma "," does not work in ETL as it does in C. You must define each variable on a separate line. For example, this structure in C:

```
typedef struct {
       float x,y; /* Invalid in ETL */
} mine;
```

looks like this in ETL:

```
typedef struct {
```

```
        float x; /* Correct in ETL */
        float y;
} mine;
```

8.4.2. No Pointer Variables

The ETL does not allow you to explicitly declare pointer variables in IRIS Explorer types. Thus you may not declare an object by reference. This is because IRIS Explorer transcription routines need to know the size of the object pointed to, and a pointer to an object could be a pointer to one or many consecutive objects. However, in the generated C structure, two IRIS Explorer type members are represented as pointers to allow for multiple references. These are the array and the reference−counted structures (`shared` and `root` structures). The array has a variable dimension determined at run−time, so it cannot be stored in−line in the containing structure. The reference−counted structures are described above.

8.4.3. No Anonymous Structure Arrays

ETL must be able to calculate the size of every non−in−line structure it manipulates so that it can handle the transcription of data properly. To do this, it needs a name for the structure.

This means you must use a `typedef` statement to name all such structures, and the name must be unique. If the structure is anonymous (or if the name is duplicated), there is no name handle for ETL to use when calculating its size.

This example illustrates the premise that all types must be explicitly and uniquely named. You may not declare an IRIS Explorer type of the following form, for IRIS Explorer would have no way of naming the structure or of representing its size and contents internally:

```
struct {
     int a;
     int b;
} s;        /* WRONG: Can't have an anonymous structure. */
```

The correct way to do this is to define a new type:

```
typedef struct {
   int a;
   int b;
} twoInt;   /* RIGHT: Structure has a type. */

twoInt s;
```

This example illustrates the premise that an embedded `struct` must be a named type unless it is in−line

Wrong: Undefined (no name), and therefore anonymous structure. No name available for computing the size of "foo".

```
typedef struct {
   int n;
   struct {
      float x;
      float y;
   } foo[n];
} mine;
```

The footype is first defined in its own typedef

```
typedef struct {
   float x;
   float y;
```

```
} footype;
```

Size of "foo" computable from n* sizeof(footype).

```
typedef struct {
    int n;
    footype foo[n];
} mine;
```

8.5. Scoping

There are several issues relating to scoping in IRIS Explorer's typing system. *Scoping* refers to the structure context in which a variable is known to exist (by IRIS Explorer's typing system). The scoping requirements limit the extent to which IRIS Explorer must search for a particular structure member.

8.5.1. Ordering Array Members

Array bounds must exist at the same level of lexical scope where they are used. Furthermore, the array bound member must precede the array in which it is used in the structure. The following examples illustrate this concept.

In this example, the variables *m* and *n* are used without having been defined.

```
typedef struct {
     int    len;
     float  vec[len];/* Correct scope for len  */
     float  box[m,n];/* Incorrect scope for m and n. */
} Example1;
```

Here, the first use of *len* occurs before it has been defined.

```
typedef struct {
      float   vec[len];     /* Len not yet set -- incorrect.*/
      int     len
      float    box[len,len];/* Len set -- correct. */
} Example2;
```

The member order in this fragment of *cxLattice* is correct:

```
root typedef struct {
     long   nDim            "Num Dimensions";
     long   dims[nDim]      "Dimensions Array";
     ...
     cxLattice;
```

But this order is wrong, because *nDim* is used before it has been defined:

```
root typedef struct {
    long   dims[nDim]   "Dimensions Array"
    long  nDim          "Num Dimensions";
    ...
} cxLattice;
```

Note: Shared memory types, such as *cxData*, *cxCoord*, and *cxLattice*, may be defined in any order.

Using Switch Discriminators

In general, switch discriminators must exist in scope where they are used. However, it is acceptable for an array bound or switch discriminator to exist in an outer scope, for example in the containing structure, provided that the inner structure is an in−line structure and not a reference−counted one.

For example, in the *cxData* structure, *primType* exists outside the switch scope and is used correctly inside it:

```
shared typedef struct {
   long           nDim;
   long           dims[nDim];
   long           nDataVar       "Num Data Variables";
   cxPrimType     primType       "Primitive Data Type";
   switch         (primType) {
      case cx_prim_byte:
           char     values[nDataVar, dims]  "Data Array";
      case cx_prim_short:
            short    values[nDataVar, dims] "Data Array";
      case cx_prim_long:
           long     values[nDataVar, dims]  "Data Array";
      case cx_prim_float:
           float    values[nDataVar, dims]  "Data Array";
      case cx_prim_double:
           double   values[nDataVar, dims]  "Data Array";
     } d;
} cxData;
```

8.5.2. The Resulting C Structure

To be useful to the IRIS Explorer programmer, the IRIS Explorer typing system provides a data structure that can be accessed through a conventional programming language. IRIS Explorer types are translated into C structures, which can be accessed directly from C or through a functional application programmer interface (API) from C, C++, or Fortran.

Here are some points to note about the structure:

- There is a direct, natural conversion from most of IRIS Explorer's typing language to C structures.
- The scalars are converted as described in **"Scalar Types"** earlier in this Chapter.
- IRIS Explorer arrays are handled as one−dimensional C arrays, but may be considered as multi−dimensional arrays by using a Fortran indexing scheme.
- Enumerated constants are translated to C enumerated constants with an origin at zero.
- The IRIS Explorer `struct` composition is identical to the C `struct` composition, however, `shared` and `root` structures add a reference−counting substructure as a prefix to the user−defined structure.

Labels for the structure members are not reflected in the resulting C structure, but are used to create the API for accessing the data structure and the menus in the Module Builder.

The `switch` statement is translated into a `union` of variant structures in the resulting C code. Each variant structure is named by the enumerated constant value that selects the variant. For example, this IRIS Explorer structure translates into the subsequent C structure:

IRIS Explorer `switch` structure:

```
switch (myType) {
case cx_my_long:
    long    a;
case cx_my_float:
     float         b;
 case cx_my_double:
     double  d ;
} s;
```

C equivalent (`union` structure):

```
union {
        struct {
               long  a;
                      } cx_my_long;
        struct {
               float        b;
                   } cx_my_float;
        struct {
                double d;
                   } cx_my_double;
   } s;
```

An important detail to notice here is that the structure members are accessed in C as *member*s:

```
s.cx_my_long.a
s.cx_my_float.b
s.cx_my_double.d
```

The `closed` keyword has no effect on the generated C structure.

8.6. Summary of ETL Syntax

This section summarizes the ETL syntax discussed in the previous sections.

- Syntactic object categories are indicated by *italics*, for example, *integer_value*.
- Literal words and characters such as IRIS Explorer keywords are indicated as in the following example: `closed`.
- Alternate categories are listed sequentially on separate lines.
- Sets of narrow alternates are listed on one line, preceded by the phrase one of.
- An optional symbol has the suffix $_{opt}$.
- The double period `..` denotes one of the range of values in the ASCII collating sequence; for example, `a..z` indicates one of the range of characters `a,b,c,...,z`.
- The triple period ... denotes a repeated syntactic object, for example, *case* ... .
- *blank* – self defining
- tab – self defining
- underscore – `_` (Note: Do not overlook it.)
- *alpha* – one of `A..Z a..z`
- *numeric* – `0..9`
- *non_zero* – `1..9`
- *text* – *alpha, numeric, underscore*
- *whitespace* – one of *blank tab*
- *character* – *text, whitespace*
- *label* – *character ...* `"`
- *comment* – `/*` *character ...* `*/`
- *include* – `#include` <FILENAME>`, #include "FILENAME"`
- *integer_value* – *numeric, numeric ...*
- *positive_integer_value* – *non_zero, non_zero numeric, non_zero numeric ...*
- *name* – *alpha, alpha text, alpha text ...*
- *base_integer* – one of `char short long`
- *sign* – one of `signed unsigned`
- *integer* – *sign*$_{opt}$*base_integer*
- *floating* – one of `float double`
- *string* – `string`

- *simple_type* – *integer, floating, string*
- *struct_type* – `struct { ` *member ...* `}` (*member* is defined below)
- *enum_type* – `enum {` *name* `}, enum {` *name*, *name ...* `}`
- *abstract_type* – *simple_type, struct_type, enum_type*
- *case* – `case` *CONSTANT : member ...* (*member* is defined below)
- *switch_type* – `switch (` DISCRIMINATOR `) {` *case ...* `}`
- *named_type* – STANDARD_TYPE
- *reference_type* – REFERENCE_TYPE
- *defined_type* – *simple_type, switch_type, named_type, reference_type*
- *array_bound* – *positive_integer_value* , *BOUND*
- *array_specific*– `[` *array_bound* `], [` *array_bound*, *array_bound ...* `]`
- *closed* – `closed`
- *member* – *closed*$_{opt}$ *defined_typeVAR array_specifier*$_{opt}$ *label*$_{opt}$;
  closed$_{opt}$ *switch_type VAR array_specifier*$_{opt}$*label*$_{opt}$;
- *port_modifier* – `root`
- *reference_modifier* – one of `root shared`
- *modifier* – *port_modifier, reference_modifier*
- *standard_definition* – *closed*$_{opt}$`typedef` *abstract–type type_name*;
- *reference_count_definition* – *closed*$_{opt}$ *modifier*`typedef` *struct_type type_name*;

VAR is a *name;* BOUND is a *name* of type *integer;* DISCRIMINATOR is a *name* of type *enum_type*; CONSTANT is a *value* of type *enum_type;*STANDARD_TYPE is a *name* of type *standard_definition;*`REFERENCE_TYPE` is a *name* of type *reference_count_definition.*

8.7. Building a Type Declaration

You now need to build the type declaration and install the files that IRIS Explorer creates in a directory of your choosing. Once this is done, you can run the Module Builder. When you open the Connections window, the new data type will show up on the port menus.

8.7.1. Building the Type and Related Files

Let us suppose that you have created the data structure for the *myCylinder* data type by using the grammar and principles described in **"Using the ETL"** earlier in this Chapter. You have saved it in a file called *myCylinder.t.*

To build the data type and install it in IRIS Explorer, we follow these steps.

1. Make a subdirectory in your current directory for constructing your data type and its related files. Use the directory name *~/explorer/cyl.*

   Go to that directory:

   ```
   mkdir ~/explorer/cyl
   cd ~/explorer/cyl
   ```

2. Put the *myCylinder.t* file in the subdirectory.
3. Make a file called *TYPES* in the same subdirectory and put the name of the data type in it. For example,

   ```
   cat > TYPES
   myCylinder
   ^D
   ```

   This *TYPES* file now contains the name `myCylinder`.
4. Create a *Makefile* for your *~/explorer/cyl* directory by running the type Makefile generator. Type the command:

   ```
   cxmkmf
   ```

IRIS Explorer then creates an *Imakefile* and a *Makefile*.

5. Build the type and related files by typing:

   ```
   make
   ```

   or

   ```
   make all
   ```

6. Install the type into the *$EXPLORERUSERHOME* subdirectory so that it will be visible to all of IRIS Explorer by running the command:

   ```
   make install
   ```

This command installs the made files in *$EXPLORERUSERHOME*. These files are:

```
include/cx/myCylinder.t
include/cx/myCylinder.h
include/cx/myCylinder.inc
include/cx/myCylinder.api.h
include/cx/myCylinder.api.inc
lib/libmyCylinder.a
types/myCylinder.type
man/man3/myCylinder.3
```

8.7.2. What the Files Do

When you run **`make install`**, IRIS Explorer translates the language in the type declaration file into C and creates the *myCylinder.type* file. It also creates the header file for the data type, *myCylinder.h* and the include file *myCylinder.inc*. You include this file in the user function file (for example, in *ReadCyl.c*) when you build a module that makes use of the Cylinder data type (for example, to read cylinder data).

The files have these uses:

include/cx/myCylinder.t
: The original type description file. Placed into a known location so that it can be included in other type files. For example, *cxLattice.t* is used in *cxPyramid.t.*

include/cx/myCylinder.h
: The generated C structure definition for the type. This header file is included by C programs that manipulate the type.

include/cx/myCylinder.inc
: The generated Fortran definition of all enumerated constants used in building the type. This header file is included by Fortran programs that manipulate the type.
Since the Fortran programmer must access the type through a set of subroutines and functions, the enumerated constants are necessary as function parameters. No structure information is included in the Fortran `include` file, since IRIS Explorer structures are not accessed directly from Fortran.

include/cx/myCylinder.api.h
: The C header file containing function prototypes for the application programmer interface to the type's accessor functions. See the library file *lib/libmyCylinder.a* below for a discussion of the accessor functions.

lib/libmyCylinder.a
: In the process of building a type, several accessor functions are created to allocate the type, to set and get members of the type, to inquire about the enumerated type and length of members, and to read and write data of this type in the standard IRIS Explorer transcribed form. Both C and Fortran versions of the library are created and stored in the archive file *lib/libmyCylinder.a*.

In addition, the library archive contains an object file *myCylinder.meta.o*, which contains a C structure describing the type. This "meta–type" description is required by the type accessor library and fully describes the type: it is a compiled object representation of the *myCylinder.t* file.

types/myCylinder.type

The *myCylinder.type* file is a binary version of the *myCylinder.meta.o* structure that can be loaded by IRIS Explorer at run–time. It is used by the Map Editor and Module Builder to discover the description of types.

Figure 8–1 shows the sequence of events in the build process.

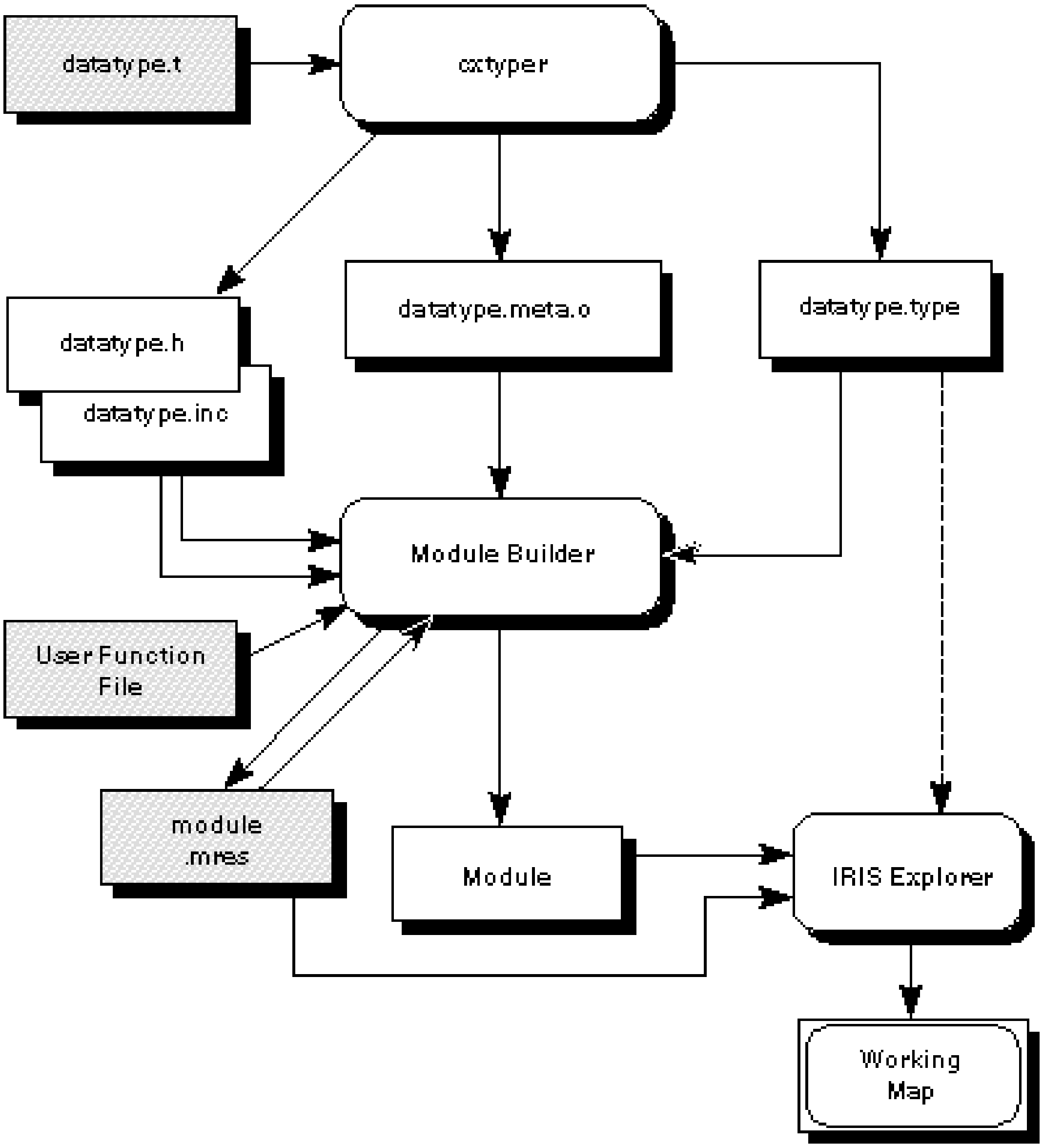


Figure 8–1 IRIS Explorer Types Information Flow

8.8. Visibility of Data Types in IRIS Explorer

Data types are present in many guises within IRIS Explorer. You can use the Module Builder to create a series of modules that use the new data type, then wire the modules into a map in the Map Editor and look at the results. You will be able to see the various manifestations of the data type declarations.

8.8.1. In the GUI

The data types are visible to users at several places in the GUI (graphical user interface). In the Map Editor, the user sees the data type on each module port when a map is wired. The Map Editor uses the data types and auxiliary constraint information to determine which ports are valid for a given wiring, hence the IO pad highlighting to illuminate potential wiring destinations.

In the Module Builder, the module writer sees a menu of all available port data types when creating input and output ports. Input and output ports can be of any root data type. The Module Builder must configure these menus at run time from the wiring connections made in the Connections panel. Description strings appear as menu items in the Connections window.

8.8.2. Between Modules

Module communication across socket links, for example between modules on different hosts or on a single host with non−shared memory, requires that both the sender and receiver know how to transmit the data structure. IRIS Explorer data structures may have variably sized portions, pointers to shared pieces, and may even be recursive. Each module is compiled with the necessary information about its constituent data types, so that it can correctly transmit, read, and write those types. Furthermore, some modules, such as *For* and *While*, must be able to receive and send data structures that were not known when they were built.

8.9. Application Programming Interface

Each data type is a structure made up of several members. The structure members can be scalars of certain types, or compositions of scalars in the form of arrays, unions, and structures. Associated with each member is a set of accessor functions, the application programming interface (API), that allows the user to set and get its value. The API is automatically generated from the label given to each member, either by the user or internally by the typing system. **Table 8−4** gives an example of a member:

Type	Variable Name	Label
short	count	"No. Of Items"

Table 8−4 Member of a Data Type

IRIS Explorer suppresses spaces and other non−alphanumeric characters such as parentheses, and runs the alphanumeric characters together to form the base name of the API routine. It then adds the type prefix (which for IRIS Explorer is *cx*). For the above example, the API routines would have the base name *shortNoOfItems**. A verb (either *Get* or *Set*) that indicates the function of the API routine is then appended; for example, to manipulate *count*, you can call *shortNoOfItemsGet* and *shortNoOfItemsSet*.

All members get the functions **Get* and **Set*, and some members have additional routines. These are listed in the next section, and you can refer to the generated manual page for the exact interface.

8.9.1. API Member Types and Functions

Here is a list of the types of members and their functions. The type is `myType`, the label is `Label`, and a trailing name indicates the form of access.

Functions for All Members

All members of a data type get these routines:

- *myTypeLabelGet()*
- *myTypeLabelSet()*

Functions for Reference−counted Structure

The reference−counted structures get these routines:

myTypeAlloc() Allocates the structure

myTypeRead() Reads an IRIS Explorer transcribed (ASCII or binary) type file

myTypeWrite() Writes an IRIS Explorer transcribed (ASCII or binary) type file

myTypeDup() Duplicates all contents of the data structure

Note: *cxDataRefDec()* is used to delete all reference−counted structures.

Functions for Arrays

The arrays get these routines:

myTypeLabelLen()
Computes the length of the array in words

myTypeLabelAlloc()
Allocates the array

Dimensioning arrays

These include arrays such as lattice−>data−>dims.

myTypeLabelProd()
Computes the product of the array's entries, to be used in computing the array length of the array for which this is a dimension (or bound).

Functions for Members in a Switch

In some cases, several members within a `switch` construct may be given the same label. The members must lie within a discriminated union (a switch) and no two members can lie in the same case of the union. In this case, the several members take on a single identity, with the active member depending on which case of the union is active.

These labelled unions have additional accessor functions:

myTypeLabelType()
The primitive data type of the active member

myTypeLabelLen()
The length of the active member in words (1 for scalar)

C and Fortran versions of the routines are generated and loaded into the library archive *lib/libmyType.a.*

Memory Handling

All IRIS Explorer user−defined type structures are placed in shared memory on machines with shared memory. When you use the *<TYPE><LABEL>Set()* routines, they free up the memory at the location they are about to overwrite if the member is an array. If the member being overwritten is a reference−counted item, the reference count is decremented.

8.9.2. Fortran Wrappers

Fortran wrappers are generated for almost all user−defined type API routines. They follow the usual rules for the IRIS Explorer Fortran API:

- Scalars are passed by reference in Fortran, rather than by value as in C.

- All user–defined types are passed as integer handles. Arrays are passed by reference to the first entry in the standard Fortran style. In some cases arrays will be returned by functions and thus require use of the `Pointer` construct in Fortran (note that this is not the Fortran 90 *POINTER* type).

Suppressing char Routines**

Incorrect wrappers are generated for routines returning a `char**` return value. For example, the prototype in the *myType.api.h* file for the routine named *myTypeItemGet()* looks like this:

```
char **myTypeItemGet( myType* src, cxErrorCode *ec );
```

You can suppress generation of these routines by following these steps.

1. *cd* to the directory in which the user is building *myType.api.o.*
2. Create a file *Imakefile.default* by typing this command at the shell prompt:

   ```
   echo "CXC2FPREPROC = c2fpreproc.sed" > Imakefile.default
   ```

3. Type this command:

   ```
   cp $(EXPLORERHOME)/lib/c2fpreproc.sed c2fpreproc.sed
   ```

4. Edit the file *c2fpreproc.sed* by adding this line at the end:

   ```
   s/.*myTypeItemGet.*$//g
   ```

Fortran wrapper generation will be suppressed for that routine. If users really want such a routine, they will have to create it themselves, by hand.

Fortran Bugs

At this time the Fortran interface is broken for routines with a *cxstring*** argument. Such a routine would be generated for a member that is an array of strings, for example:

```
int         nFiles              "Number Files";
string      filename[ nFiles ]  "File Names";
```

In this case, the routines *myTypeFileNamesGet()* and *myTypeFileNamesSet()* would not function correctly.

8.9.3. Location of API ManPages

The manual pages for the API, also automatically generated, will be installed in *$EXPLORERUSERHOME/man/man3.* For information on setting the value of the environment variable *EXPLORERUSERHOME* refer to **"Defining EXPLORERUSERHOME"** in Chapter 2.

This directory is probably not searched by default by the *man* command, but you can add this directory to your search path by defining the *MANPATH* variable as follows:

```
setenv MANPATH (${MANPATH}:$EXPLORERUSERHOME/man)
```

8.10. Crossing Machine Boundaries

This applies when you go from one type of machine to another, for example, from an IRIS to a Cray, or when you go from shared memory to non–shared memory on a single IRIS. The Module Control Wrapper (MCW) translates the information from the data type into a contiguous stream of data and sends it from one machine to the other. For the process to work, the other machine must know how to reconstitute this data stream into a recognizable data type. This means that the other system must know about the new data type.

You must recreate the type declaration file for the new data type on the other machine because the *modulename.type* file is specific to the machine it is created on. For example, an IRIX *modulename.type* file will provide incorrect information to the part of IRIS Explorer running on a Sun, IBM or HP.

8.11. Example of a User−defined Type

Here is an example of a user−defined type which is basically an IRIS Explorer lattice with the addition of three new scalars (*Minimum*, *Maximum*, and *Empty*). For each data variable, there is a description label string

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

.

This is the data type definition. It is the ETL description of the user−defined type "tstLattice" and resides in */usr/explorer/src/MWGcode/UserTypes/types/tstLattice.t*.

```
#include        <cx/DataCtlr.h>
#include        <cx/Typedefs.t>
#include        <cx/cxLattice.t>

shared root typedef struct {
    long                nDim        "Num Dimensions";
    long                dims[nDim]  "Dimensions Array";
    tstData(nDim, dims) data        "Data Structure";
    cxCoord(nDim, dims) coord       "Coord Structure";
} tstLattice;

shared typedef struct {     /* IRIS Explorer Lattice's Data array */
    long            nDim;
    long            dims[nDim];
    float           minimum     "Minimum";
    float           maximum     "Maximum";
    float           empty       "Empty";
    long            nDataVar        "Num Data Variables";
    string          labels[nDataVar]    "Data Labels";
    cxPrimType      primType        "Primitive Data Type";
    switch          (primType) {
       case cx_prim_byte:
          char      values[nDataVar, dims] "Data Array";
       case cx_prim_short:
          short         values[nDataVar, dims] "Data Array";
       case cx_prim_long:
          long          values[nDataVar, dims] "Data Array";
       case cx_prim_float:
          float         values[nDataVar, dims] "Data Array";
       case cx_prim_double:
          double        values[nDataVar, dims] "Data Array";
    } d;
} tstData(nDim, dims);
```

8.11.1. Test Module 1

This code uses the user−defined data type, *tstLattice*. It takes in an IRIS Explorer lattice and generates a tstLattice. Before you can make this module you must have installed the user−defined data type *tstLattice*, following the instructions given in **"Building the Type and Related Files"**.

The easiest way to demonstrate this is to connect *GenLat* to the input. You can extract some information from the new data type by building the second test module, and connecting it to the first one. The code resides in */usr/explorer/src/MWGcode/UserTypes/C/generate.c* and */usr/explorer/src/MWGcode/UserTypes/Fortran/generate.f.* Resources files may be found in the same directories.

C Version:

```
/* This test module inputs a cxLattice and outputs a tstLattice.
 * The differences are that the tstLattice has additional fields in the
 * tstData (equivalent to cxData) structure for minimum, maximum,
 * empty_cell_value, and labels for each data variable.
 * This can be seen as a super-lattice.
 */

#include <cx/DataAccess.h>
#include <cx/DataTypes.h>
#include <cx/DataOps.h>
#include <cx/tstLattice.api.h>

#define CLEAN(A,B,C) if(memclean((void *)A,B,C)) return

int memclean(void *ptr, cxErrorCode ier, tstLattice *lat)
{
  if((ptr == NULL) || (ier != cx_err_none))
    {
      cxDataRefDec(lat);
      return 1;
    }
  else
    return 0;
}

void generate(cxLattice *inLat, tstLattice **outLat)
{
  char **labels;
  cxCoord *coord;
  cxCoordType coordType;
  cxData *data;
  cxErrorCode err;
  cxPrimType primType;
  int i, num_bytes;
  long nDim,*dims,hasData,nDataVar,hasCoord,nCoordVar;
  tstData *tdata;
  void *inDataArray, *outDataArray;

  /* Get the lattice description for the incoming lattice */
  cxLatDescGet(inLat,&nDim,&dims,&hasData,&nDataVar,&primType,
        &hasCoord,&nCoordVar,&coordType);

  /* Make the complete tstLattice structure and all its components
   * using the incoming lattice characteristics
   * Return if any of the allocations fail
   */
  *outLat = tstLatticeAlloc(nDim,dims);
```

```
  if(*outLat == NULL)
    return;
  tdata = tstDataAlloc(nDim,dims,nDataVar,primType);
  CLEAN(tdata,0,*outLat);
  tstLatticeDataStructureSet(*outLat,tdata,&err);
  CLEAN(tdata,err,*outLat);
  outDataArray = tstDataDataArrayAlloc(tdata);
  CLEAN(outDataArray,0,*outLat);
  tstDataDataArraySet(tdata,&outDataArray,&err);
  CLEAN(tdata,err,*outLat);
  cxLatPtrGet(inLat,&data,&inDataArray,&coord,NULL);
  tstLatticeCoordStructureSet(*outLat,coord,&err);
  CLEAN(tdata,err,*outLat);

  /* Additional characteristics for the new tstLattice
   * limit the data range that is accepted by the lattice
   */
  tstDataMinimumSet(tdata,0.0,&err);
  CLEAN(tdata,err,*outLat);
  tstDataMaximumSet(tdata,1.0,&err);
  CLEAN(tdata,err,*outLat);
  tstDataEmptySet(tdata,-999.,&err);
  CLEAN(tdata,err,*outLat);

  /* Get the pointer to the labels structure and
   * define the labels (they are the main difference between the
   * new tstLattice type and cxLattice)
   */
  labels = tstDataDataLabelsGet(tdata, &err);
  CLEAN(tdata,err,*outLat);
  for(i=0;i<nDataVar;i++)
  {
    labels[i] = cxDataMalloc(sizeof("label 99")+1);
    CLEAN(labels[i],err,*outLat);
    sprintf(labels[i],"label %d",i);
  }

  /* Copy the rest of the incoming lattice to the outgoing tstLattice */
  num_bytes = cxDataPrimSize(data);
  num_bytes *= cxDimsProd(nDim,dims,nDataVar);
  bcopy(inDataArray,outDataArray,num_bytes);
}
```

Fortran Version:

```
      SUBROUTINE GENER(INLAT,OUTLAT)
C
C     fortran example for UDT's
C
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
      INCLUDE '/usr/explorer/include/cx/DataOps.inc'
      INCLUDE '/usr/explorer/include/cx/tstLattice.api.inc'
C
C     .. Parameters ..
```

```
      REAL             ZERO, ONE, EMPTY
      PARAMETER        (ZERO=0.0,ONE=1.0,EMPTY=-999.0)
      INTEGER          LSIZE, MAXCHR
      PARAMETER        (LSIZE=4,MAXCHR=10)
C     .. Scalar Arguments ..
      INTEGER          INLAT, OUTLAT
C     .. Local Scalars ..
      INTEGER          COORD, CTYPE, DATA, ERR, HASCRD, HASDAT, I, IER,
     *                 NBYTES, NCVAR, NDIM, NDVAR, P0, PTYPE, TDATA
C     .. Local Arrays ..
      INTEGER          DIMS(1)
      CHARACTER        INARR(1), OUTARR(1)
      CHARACTER*(MAXCHR) LABELS(1)
C     .. External Functions ..
      INTEGER          MEMCLN
      EXTERNAL         CXDATAMALLOC, CXDATAPRIMSIZE, CXDIMSPROD, MEMCLN,
     *                 TSTDATAALLOC, TSTDATADATAARRAYALLOC,
     *                 TSTDATADATALABELSGET, TSTLATTICEALLOC
C     .. External Subroutines ..
      EXTERNAL         CXLATDESCGET, CXLATPTRGET, TSTDATADATAARRAYSET,
     *                 TSTDATAEMPTYSET, TSTDATAMAXIMUMSET,
     *                 TSTDATAMINIMUMSET, TSTLATTICECOORDSTRUCTURESET,
     *                 TSTLATTICEDATASTRUCTURESET
C     .. Pointers to Lattice Structures ..
      POINTER (PDIMS,DIMS)
      POINTER (PIN, INARR)
      POINTER (POUT, OUTARR)
      POINTER (PLAB, LABELS)
C     .. Executable Statements ..
C
C     Get the lattice description for the incoming lattice
C
      IER = CXLATDESCGET(INLAT,NDIM,PDIMS,HASDAT,NDVAR,PTYPE,HASCRD,
     *                   NCVAR,CTYPE)
C
C     Make the complete tstLattice structure and all its components
C     using the incoming lattice characteristics
C     Return if any of the allocations fail
C
      OUTLAT = TSTLATTICEALLOC(NDIM,DIMS)
      IF (OUTLAT.EQ.0) RETURN
      TDATA = TSTDATAALLOC(NDIM,DIMS,NDVAR,PTYPE)
      IF (MEMCLN(TDATA,0,OUTLAT).GT.0) RETURN
      CALL TSTLATTICEDATASTRUCTURESET(OUTLAT,TDATA,ERR)
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
      POUT = TSTDATADATAARRAYALLOC(TDATA)
      IF (MEMCLN(POUT,0,OUTLAT).GT.0) RETURN
      CALL TSTDATADATAARRAYSET(TDATA,POUT,ERR)
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
      P0 = 0
      IER = CXLATPTRGET(INLAT,DATA,PIN,COORD,P0)
      CALL TSTLATTICECOORDSTRUCTURESET(OUTLAT,COORD,ERR)
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
      CALL TSTDATAMINIMUMSET(TDATA,ZERO,ERR)
```

```
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
      CALL TSTDATAMAXIMUMSET(TDATA,ONE,ERR)
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
      CALL TSTDATAEMPTYSET(TDATA,EMPTY,ERR)
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
C
C     We cannot use the Fortran API function TSTDATADATALABELSGET
C     So will allocate our own array and replace it in the structure
C     Free the LABELS array, after it has been copied into TDATA
C
      PLAB = CXDATACALLOC(NDVAR,MAXCHR)
      IF (MEMCLN(PLAB,ERR,OUTLAT).GT.0) RETURN
      DO 20 I = 1, NDVAR
         WRITE (LABELS(I),FMT='(''Label '',I2)') I-1
 20   CONTINUE
      CALL TSTDATADATALABELSSET(TDATA,LABELS,ERR)
      IF (MEMCLN(TDATA,ERR,OUTLAT).GT.0) RETURN
      CALL CXDATAFREE(PLAB)
C
C     Copy the rest of the lattice into the tstLattice
C
      NBYTES = CXDATAPRIMSIZE(DATA)*CXDIMSPROD(NDIM,DIMS,NDVAR)
      DO 40 I = 1, NBYTES
         OUTARR(I) = INARR(I)
   40 CONTINUE
C
      RETURN
      END
      INTEGER FUNCTION MEMCLN(PTR,ERCODE,TSTLAT)
C
C     cleanup routine
C
C     .. Scalar Arguments ..
      INTEGER                 ERCODE, PTR, TSTLAT
C     .. External Subroutines ..
      EXTERNAL                CXDATAREFDEC
C     .. Executable Statements ..
C
      IF ((PTR.EQ.0) .OR. (ERCODE.NE.0)) THEN
         CALL CXDATAREFDEC(TSTLAT)
         MEMCLN = 1
      ELSE
         MEMCLN = 0
      END IF
      RETURN
      END
```

8.11.2. Test Module 2

This module reads in a *tstLattice* user–defined type and prints out some specific information from it. The code resides in */usr/explorer/src/MWGcode/UserTypes/C/print.c*. A resources file may also be found in this directory. As the Fortran wrapper to the C function *tstDataDataLabelsGet* is incorrect (this function returns a *char***), the equivalent Fortran example would require a user–generated Fortran wrapper. The Fortran example program is not supplied.

```
#include <cx/DataAccess.h>
#include <cx/DataTypes.h>

#include <cx/tstLattice.api.h>

void prt( tstLattice *inlat )
{
  char **f;
  tstData *outData;
  long i, nDV;
  float mt;
  cxErrorCode err;

  outData = tstLatticeDataStructureGet(inlat,&err);
  f = tstDataDataLabelsGet(outData,&err);
  nDV = tstDataNumDataVariablesGet(outData,&err);
  for(i=0;i<nDV;i++)
  {
    printf("label %d = %s\n",i,f[i]);
  }
  mt = tstDataEmptyGet(outData,&err);
  printf(" Empty value = %f\n",mt);
}
```

8.11.3. Auto−generated Modules

There are two more modules in directory */usr/explorer/src/MWGcode/UserTypes/C*, *ReadtstLattice* and *WritetstLattice*, that allow the user to read and write ASCII or binary versions of the user−defined type. The user function for these modules is a simple interface to the two auto−generated routines *tstLatticeWrite* and *tstLatticeRead.*

Chapter 9
Advanced Module Writing

This chapter discusses some advanced aspects of module writing. It covers some of the more esoteric functions of the IRIS Explorer product, which you may be interested in if you are writing complex and powerful modules that push the capabilities of IRIS Explorer beyond everyday levels. Some of the topics describe advanced techniques, such as writing hook functions, and some are more explanatory, such as the section on the firing algorithm. An in−depth knowledge of how IRIS Explorer operates allows you to make the most of the benefits that IRIS Explorer offers.

You need not read this chapter unless you are interested in learning more about IRIS Explorer or in implementing these techniques.

9.1. Controlling Module Functions

IRIS Explorer provides the module writer with many opportunities to make a choice between letting IRIS Explorer control a particular process in the module, or taking charge and running that process directly from the user function. As a result, you can write modules that operate at varying levels of independence from IRIS Explorer. Where the module writer settles on the continuum from total control to very little depends on current needs and programming expertise.

The major divide is between modules with a Module Data Wrapper (MDW) and those without. If you build a module without an MDW, you must make sure that your user function performs all the actions that the MDW would otherwise have done.

Figure 9−1 shows the range of module complexity. The modules in division 1 contain code that works anywhere, and is reusable in other programs and systems. Division 2 modules contain code, possibly written for other purposes, that can be made to work exclusively in IRIS Explorer without a lot of hard work, by adding *cx** API routines and the IRIS Explorer data type structures to the basic function or subroutine. Modules in division 3 are written specifically for IRIS Explorer, designed to take advantage of all the flexibility and potential in the system, and the code is not reusable at all in other systems.

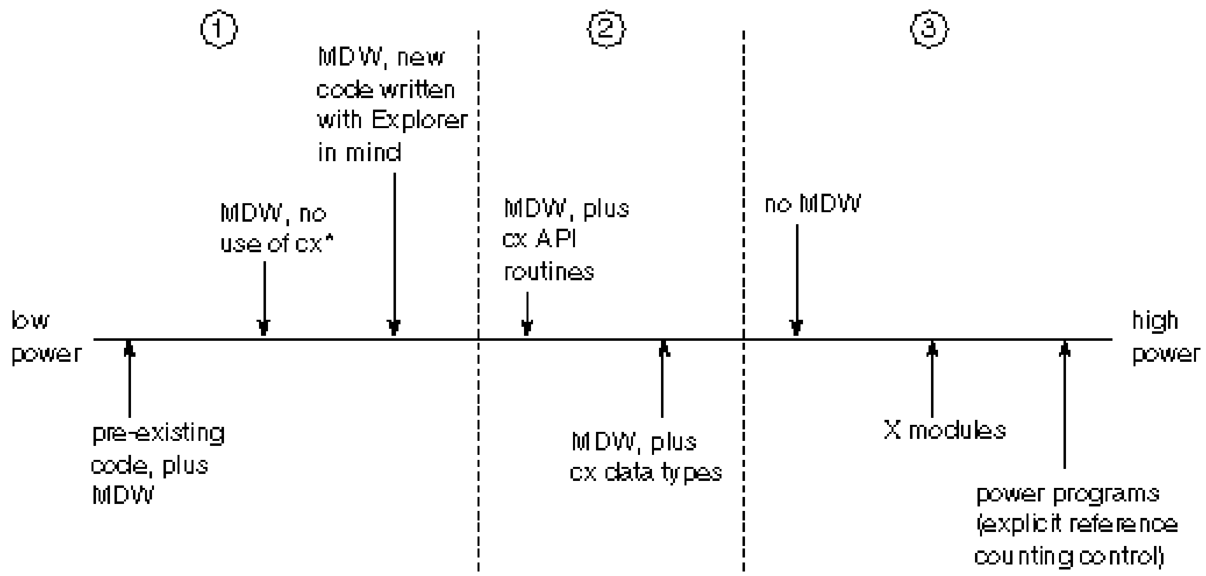


Figure 9−1 Spectrum of Module Complexity

9.1.1. The Module Wrappers

IRIS Explorer uses three controlling interfaces or "wrappers" to mediate between the module's user function and the rest of the system. They are the Module Control Wrapper (MCW), the Module Data Wrapper (MDW), and the Generic Wrapper. The relationship between wrappers and user or computational function is shown in **Figure 9−2**.

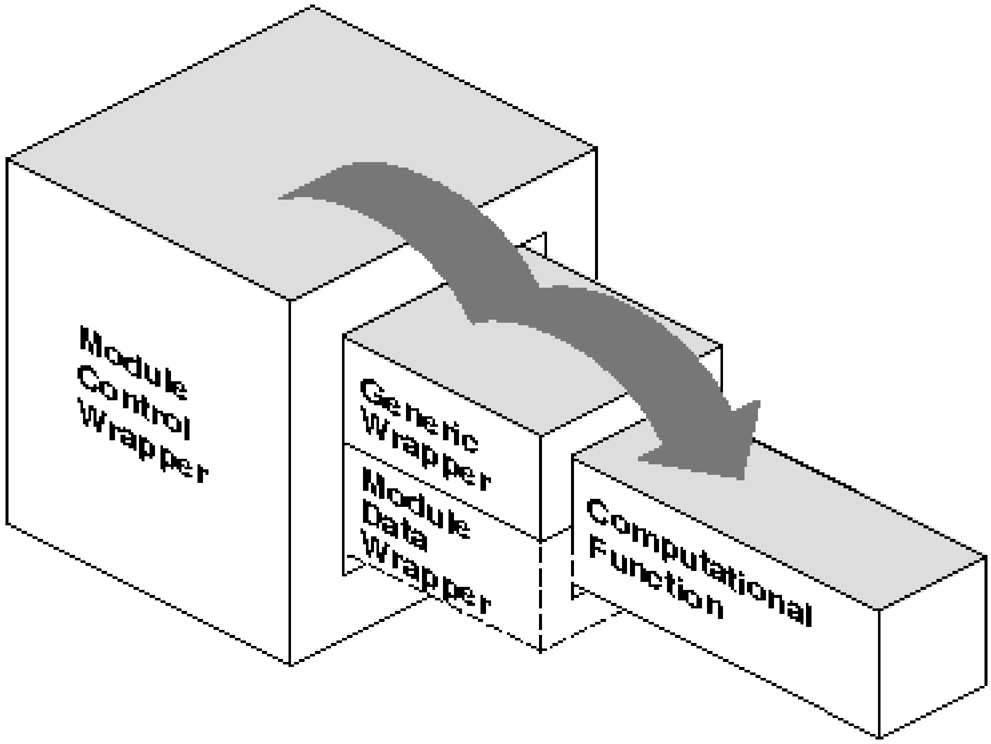


Figure 9−2 The Module Structure

Module Control Wrapper

Every module has a Module Control Wrapper (MCW), which is the module data manager. The MCW contains the firing algorithm and provides interfaces to input and output functions, including port and widget settings. The MCW also manages all communication with other modules.

The firing algorithm is described in **Appendix B, "The Firing Algorithm"**. Information about the MCW's interface capacity is contained in the API routines for all the input and output functions.

The Module Builder constructs two kinds of MCWs, the default and a special form for X modules. The X−MCW uses the X mechanism for handling scheduling.

Module Data Wrapper

The Module Data Wrapper (MDW) performs conversions between IRIS Explorer and user−defined data types on the ports and the data format required by the user function. You can thus develop modules with customized interfaces without having to interact with the IRIS Explorer data types. Since you can create IRIS Explorer modules from existing code, it is easy to fold a function into a module without writing a line of source code yourself. Similarly, it is possible to convert a library of routines into a library of modules with minimal effort.

The connection between the data type and the subroutine is mediated by the Module Data Wrapper (MDW), which translates input data into a form the user function can process, and then translates the result into a suitable form for output. The Module Builder generates an MDW for a module by default. However, you can choose not to have an MDW, in which case you must explicitly incorporate the MDW functions in your user function. These include:

- Extracting data from input ports and placing data on output ports.
- Controlling the interface with the Module Control Wrapper for special callback events.
- Coercing input data into a form that the module can handle.

The Generic Wrapper

The Generic Wrapper is not accessible by the module writer, except indirectly through the Hook Functions menu. It contains:

- Hook function interfaces
- User function table (where the name of the user function or MDW routine is declared to the MCW)
- Declaration of the metatypes (IRIS Explorer's internal form of a data type) for the necessary IRIS Explorer and user−defined types, for example, *cxLatticeMetaType.*
- Port checking code for run−time data constraint checking on lattices.

9.1.2. Working with Module Wrappers

In general, when you build a module, the Module Builder writes code for the Module Data Wrapper and generates a linkage to a default Module Control Wrapper (MCW), or an X Window System MCW if so selected. You make these choices using the Build Options window.

To display the Build Options window, select *Options* from the Build menu on the Module Builder window. The Build Options window offers you options for expanding individual control over the interface between the user function and IRIS Explorer.

Modules Without MDW

If you stipulate no automatically generated MDW, you must construct the interface between the user function and the input and output ports yourself. To build a module that has no Module Data Wrapper, follow these steps.

1. Create the basic module, including input and output ports, the control panel, and the documentation, in the Module Builder.

   You must define the user function name in the Function Arguments window, but you can skip the function definitions and the Connections window; the information in these panes is used to generate the MDW.

2. Open the Build Options menu from the main Module Builder window and select *Options.*

3. Toggle the *Write wrapper code* button to *No.*

The user function must create the access to input and output ports using API subroutines, such as:

- *cxInputPortOpen* to open each port
- *cxInputDataGet* to get data from an input port
- *cxOutputDataSet* to put data on an output port
- *cxInputDataChanged* to check for modified data on an input port
- *cxFireDataChanged* to check for modified data on the *Fire* port
- *cxInputConnectsGet* to determine the number of inputs to a port
- *cxFireConnectsGet* to determine the number of inputs to the *Fire* port
- *cxOutputNoSync* to prevent data being put on the *Firing Done* port

For details on these subroutines, see the *IRIS Explorer Reference Pages* or the on−line *man* pages.

Using the X−MCW Option

When you build a module, you can choose whether to generate a default Module Control Wrapper or an X Window System MCW. You will require an X−MCW only if you have a DrawingArea widget, or otherwise need access to the X server.

The X−MCW provides IRIS Explorer with the ability to recognize and manipulate X Window widgets in a module. You can use a drawing area in a module as an X input window, to hold an X Window or Motif widget, or as a general X graphics drawing area. IRIS Explorer modules that contain X Window drawing areas and use an X−MCW include *GenerateColormap* and *Histogram*.

The code examples in **"Examples Using X Widgets"** later in this Chapter illustrate how to incorporate some of these X widgets into an IRIS Explorer module.

These are the advantages of using the X Window System for customized graphics:

- It is the only way to incorporate such graphics into an IRIS Explorer module.
- You may get better performance from the graphics because there is no traffic from the IRIS Explorer user interface to the module.
- You have complete freedom in creating the user interface. You may even be able to use X Window System code that has already been written for another application.

These are the disadvantages:

- You have to manage the entire user interface for the X Window yourself, which takes a lot of work.
- You cannot use parameter functions with widgets managed directly by the module.
- The module binary file will be larger than that for an IRIS Explorer module, since it must be linked with the X Window System, and that consumes memory.
- Your module may look different from other IRIS Explorer modules, and a different interface may confuse your users.

IRIS Explorer provides some API subroutines to help you handle X Window System widgets efficiently. They are described in the *IRIS Explorer Reference Pages* and the on−line *man* pages and summarized in **Table 9−1**.

Subroutine	Function
cxXtAreaInitialize	Creates a drawing area that can later be bound to a module control panel
cxXtAreaAttach	Attaches a widget hierarchy to a control panel drawing area
cxXtAreaResize	Updates the size of a widget hierarchy in the drawing area of a module control panel
cxXtAreaCallbackAdd	Registers a callback function for a *cxXtArea*
cxXtGLAreaInitialize	Creates an OpenGL window inside a control panel drawing area
cxXtGLAreaRedraw	Registers a redraw procedure with a *cxXtGLArea*

Table 9−1 X Window API Subroutines

9.1.3. Examples Using X Widgets

This example shows how an X drawing area can be used as an input mechanism. It places a Motif button in an X drawing area and reports whenever the button is pressed. The code is in */usr/explorer/src/MWGcode/Advanced/C/XDrawInput.c*. The resources file *XDrawInput.mres* may be found in the same directory.

Note: It is not possible to write this module in Fortran.

```
/*
 * This example shows how an X drawing area can be used
 * as an input mechanism.
 */
```

```
#include <X11/Intrinsic.h>
#include <X11/Xm/PushB.h>

#include <cx/XtArea.h>

void xcallback(Widget w, void *client)
{
     printf("button pressed\n");
}

void xwidget(long window)
{
     static int first = 1;
     static Widget top;

    Widget button;

    /*
     * Do initialization stuff
     */
    if (first) {
 first = 0;

 /*
  * Get a Xt form and put a button on it
  */
 top = cxXtAreaInitialize();
 button = XtVaCreateManagedWidget("button",
     xmPushButtonWidgetClass, top,
     XmNleftAttachment, XmATTACH_FORM,
     XmNrightAttachment, XmATTACH_FORM,
     XmNtopAttachment, XmATTACH_FORM,
     XmNbottomAttachment, XmATTACH_FORM,
     NULL);

 /*
  * Add a callback then attach the form to the window
  */
 XtAddCallback(button, XmNactivateCallback,
     (XtCallbackProc) xcallback, (XtPointer) NULL);

 cxXtAreaAttach(top, window);
    }
    cxXtAreaResize(top, window);
}
```

Graphics Library (OpenGL) Example

This example uses a GLXWidget to create a drawing area. The code is in */usr/explorer/src/MWGcode/Advanced/C/GLXwidget.c* and in */usr/explorer/src/MWGcode/Advanced/Fortran/GLXwidget.f*

Footnote: The Fortran examples were created for use on Silicon Graphics machines. An extended form of Fortran 77 is used.

. Resources files may also be found in these directories.

Note that to create a module that calls OpenGL directly it is necessary to have a linkable version of the OpenGL libraries for your platform.

C Version:

```
/*
 * Example module using a GLXWidget
 */

#include <stdio.h>

/* X includes */
#include <X11/Intrinsic.h>
#include <X11/Xm/Xm.h>
#if defined(__sgi) && ! defined(IRIX_4)
#include <Sgm/GlxMDraw.h>
#else
#include <X11/Xirisw/GlxMDraw.h>
#endif

/* GL includes */
#include <gl/gl.h>

/* Explorer includes */
#include <cx/PortAccess.h>
#include <cx/DataAccess.h>
#include <cx/UserFuncs.h>
#include <cx/XtArea.h>

Widget top; /* top level form widget */

/*
 * Widget action routine
 */
void xaction(void *client, XEvent *ev)
{
  float p[2];     /* temporary storage for coordinates */
  short ysize;    /* height of GLX widget */

  switch (ev->type)
    {
    case ButtonPress:
      /*
       * IMPORTANT NOTE: X origin is *upper* left;
       * GL origin is *lower* left.
       */
      XtVaGetValues(top, XmNheight, &ysize, NULL);
      printf("button pressed at %d, % d\n",
      ev->xbutton.x, ysize - 1 - ev->xbutton.y);
      break;
    case Expose:
      /*
       * some sample GL stuff
       */
```

```
      reshapeviewport();
      cpack(0x00b070a0);
      clear();

      perspective(400, 1.0, 0.5, 5.0);
      translate(0.0, 0.0, -4.0);
      cpack(0xffffffff);
      bgnclosedline();
      p[0] = 0.0;
      p[1] = 0.0;
      v2f(p);
      p[0] = 1.0;
      p[1] = 1.0;
      v2f(p);
      p[0] = 1.0;
      p[1] = -1.0;
      v2f(p);
      p[0] = -1.0;
      p[1] = -1.0;
      v2f(p);
      p[0] = -1.0;
      p[1] = 1.0;
      v2f(p);
      endclosedline();
      swapbuffers();
      break;
    }
}

/*
 * Module initialization routine
 *
 * This routine must be listed as the "initialization" hook function
 */
void xex2_init()
{
  long wid;                     /* window id from UI */
  Widget da;                    /* drawing area widget */
  XtTranslations translations;  /* X translations structure */

  /* X actions record */
  static XtActionsRec actions[] ={"xaction",(XtActionProc) xaction};
  /* GL configuration */
  static GLXconfig config[] =
    {
      GLX_NORMAL, GLX_RGB, TRUE, GLX_NORMAL, GLX_DOUBLE, TRUE,
      GLX_NORMAL, GLX_ZSIZE, GLX_NOCONFIG, 0, 0, 0
    };

  /* Initialize an Xt area, get an empty form widget */
  top = cxXtAreaInitialize();

  /* Get the UI drawing area widget id  */
  wid = cxParamLongGet
```

```
    ((cxParameter *) cxInputDataGet(cxInputPortOpen("window")));

  /* Define some actions and translations for our widget */
  XtAddActions(actions, XtNumber(actions));
  translations = XtParseTranslationTable
    ("<BtnDown>: xaction()\n\ <Expose>: xaction()");

  /* Put a drawing area widget in the form */
  da = XtVaCreateManagedWidget("da", glxMDrawWidgetClass, top,
          GlxNglxConfig, config,
          XmNtranslations, translations,
          XmNleftAttachment, XmATTACH_FORM,
          XmNtopAttachment, XmATTACH_FORM,
          XmNrightAttachment, XmATTACH_FORM,
          XmNbottomAttachment, XmATTACH_FORM,
          NULL);

  /* Attach the form to the drawing area window  */
  cxXtAreaAttach(top, wid);
}

/*
 * Module execution function
 *
 */
long xex2()
{
  int windowPort;  /* port number of window parameter */
  long wid;   /* window id from UI */

  /* Get window port number and the widget ID of the drawing area */
  windowPort = cxInputPortOpen("window");
  wid = cxParamLongGet
    ((cxParameter * )cxInputDataGet(cxInputPortOpen("window")));

  /* Resize the form widget to the UI drawing area size */
  cxXtAreaResize(top, wid);

  return 0;
}
```

Fortran Version:

```
      SUBROUTINE FGL(WIN)
C
C     User function
C
      INCLUDE '/usr/explorer/include/cx/XtGLArea.inc'
C
C     NOTE -- the redraw and input procedures MUST be declared external
C
C     .. Scalar Arguments ..
      INTEGER        WIN
C     .. Scalars in Common ..
```

```
      INTEGER        WIDGET
C     .. Local Scalars ..
      LOGICAL        FIRST
C     .. Local Arrays ..
      INTEGER        CONFIG(12)
C     .. External Functions ..
      INTEGER        CXXTGLAREAINITIALIZE
      EXTERNAL       CXXTGLAREAINITIALIZE
C     .. External Subroutines ..
      EXTERNAL       CXXTAREACALLBACKADD, CXXTAREARESIZE,
     *               CXXTGLAREAREDRAW, INPUT, RFRESH
C     .. Common blocks ..
      COMMON         WIDGET
C     .. Data statements ..
      DATA           FIRST/.TRUE./
C     .. Executable Statements ..
C
      IF (FIRST) THEN
C
C        First time initialization
C
         FIRST = .FALSE.
C
C        Set up the GL configuration information
C
         CONFIG(1) = GLX_NORMAL
         CONFIG(2) = GLX_RGB
         CONFIG(3) = 1
         CONFIG(4) = GLX_NORMAL
         CONFIG(5) = GLX_DOUBLE
         CONFIG(6) = 1
         CONFIG(7) = GLX_NORMAL
         CONFIG(8) = GLX_ZSIZE
         CONFIG(9) = GLX_NOCONFIG
         CONFIG(10) = 0
         CONFIG(11) = 0
         CONFIG(12) = 0
C
C        Create the GL area
C
         WIDGET = CXXTGLAREAINITIALIZE(WIN,CONFIG)
C
C        Register the redraw and input procedures with the widget
C
         CALL CXXTGLAREAREDRAW(WIDGET,RFRESH,0)
         CALL CXXTAREACALLBACKADD(WIDGET,'inputCallback',INPUT,0)
      END IF
C
C     Match the GL area size with the window size
C
      CALL CXXTAREARESIZE(WIDGET,WIN)
C
      RETURN
      END
```

```
      SUBROUTINE RFRESH(WIDGET,CLIENT)
C
C     Redraw procedure
C
C     .. Scalar Arguments ..
      INTEGER           CLIENT, WIDGET
C     .. Local Arrays ..
      REAL              P(2)
C     .. External Subroutines ..
      EXTERNAL          BGNCLO, CLEAR, CPACK, ENDCLO, PERSPE, RESHAP,
     *                  SWAPBU, TRANSL, V2F
C     .. Executable Statements ..
C
C     Configure the viewport
C
      CALL RESHAP
C
C     Clear the screen
C
      CALL CPACK(11563168)
      CALL CLEAR
C
C     Do some example GL stuff
C
      CALL PERSPE(400,1.0,0.5,5.0)
      CALL TRANSL(0.0,0.0,-4.0)
      CALL CPACK(-1)
      CALL BGNCLO
C
      P(1) = 0.0
      P(2) = 0.0
      CALL V2F(P)
C
      P(1) = 1.0
      P(2) = 1.0
      CALL V2F(P)
C
      P(1) = 1.0
      P(2) = -1.0
      CALL V2F(P)
C
      P(1) = -1.0
      P(2) = -1.0
      CALL V2F(P)
C
      P(1) = -1.0
      P(2) = 1.0
      CALL V2F(P)
C
      CALL ENDCLO
      CALL SWAPBU
C
      RETURN
      END
```

```
      SUBROUTINE INPUT(WIDGET,CLIENT,INFO)
C
C     Input procedure
C
C     .. Scalar Arguments ..
      INTEGER          CLIENT, WIDGET
C     .. Array Arguments ..
      INTEGER          INFO(5)
C     .. Local Arrays ..
      INTEGER          XEV(32)
C     .. Pointers to Lattice Structures ..
      POINTER (XEVPTR, XEV)
C     .. Executable Statements ..
C
C     Get the pointer to the X event
C     xev will now point to the XEvent structure.
C     Be *very* careful extracting fields from this C struct.
C     This is definitely not portable code.
C
      XEVPTR = INFO(2)
C
C     Switch on the kind of event
C     Event numbers may be found in /usr/include/X11/X.h.
C     Here are a few examples:
C     KeyPress               2
C     KeyRelease             3
C     ButtonPress            4
C     ButtonRelease          5
C
      IF (XEV(1).EQ.4) THEN
C
C        IMPORTANT NOTE: X origin is *upper* left;
C        GL origin is *lower* left.
C
         PRINT *, 'button pressed at ', XEV(9), ', ', XEV(10)
      END IF
C
      RETURN
      END
```

XtGLArea Examples

These example programs use XtGLArea routines. The code is in */usr/explorer/src/MWGcode/Advanced/C/XtGLArea.c* and in */usr/explorer/src/MWGcode/Advanced/Fortran/XtGLArea.f*. Resources files may be found in the same directories.

C Version:

```
/*
 * This program creates a GL window and draws inside it.
 *
 * In order to link this code in a module, the proper set of geometry
 * and X Window System libraries must be defined in the user Imakefile
 */
```

```
#include <X11/Intrinsic.h>
#include <gl/glws.h>

#include <cx/XtArea.h>
#include <cx/XtGLArea.h>

/*
 * Redraw procedure
 */
void        refresh(void *w, void *client)
{
    float        p[2];    /* coordinate vector */

    /*
     * Configure the viewport and clear the display
     */
    reshapeviewport();
    cpack(0x00b070a0);
    clear();

    /*
     * Now do some sample GL stuff
     */
    perspective(400, 1.0, 0.5, 5.0);
    translate(0.0, 0.0, -4.0);
    cpack(0xffffffff);
    bgnclosedline();
    p[0] = 0.0;
    p[1] = 0.0;
    v2f(p);
    p[0] = 1.0;
    p[1] = 1.0;
    v2f(p);
    p[0] = 1.0;
    p[1] = -1.0;
    v2f(p);
    p[0] = -1.0;
    p[1] = -1.0;
    v2f(p);
    p[0] = -1.0;
    p[1] = 1.0;
    v2f(p);
    endclosedline();
    swapbuffers();
}


#include <cx/DataTypes.h>

#ifdef __cplusplus
extern "C" {
#endif
```

```
/*
 * User function
 */
void        cgl ( long win )
{
    static          first = 1;
    /* GLX configuration information */
    static GLXconfig config[] ={
 GLX_NORMAL, GLX_RGB, TRUE,
 GLX_NORMAL, GLX_DOUBLE, TRUE,
 GLX_NORMAL, GLX_ZSIZE, GLX_NOCONFIG,
 0, 0, 0
    };
    static Widget   widget;

    /*
     * First time, do initialization
     */
    if (first) {
 first = 0;

 /*
  * Create the GL area and register the redraw proc
  */
 widget = cxXtGLAreaInitialize(win, config);
 cxXtGLAreaRedraw(widget, refresh, 0);
    }

    /*
     * Match the GL area size with user interface window size
     */
    cxXtAreaResize(widget, win);
}

#ifdef __cplusplus
}
#endif
```

Fortran Version:

```
      SUBROUTINE FGL(WIN)
C
C     This example creates a GL window and draws inside it.
C
      INCLUDE '/usr/explorer/include/cx/XtGLArea.inc'
C
C     NOTE -- the redraw procedure MUST be declared external
C
C     .. Scalar Arguments ..
      INTEGER        WIN
C     .. Scalars in Common ..
      INTEGER        WIDGET
C     .. Local Scalars ..
      LOGICAL        FIRST
```

```
C     .. Local Arrays ..
      INTEGER        CONFIG(12)
C     .. External Functions ..
      INTEGER        CXXTGLAREAINITIALIZE
      EXTERNAL       CXXTGLAREAINITIALIZE
C     .. External Subroutines ..
      EXTERNAL       CXXTAREARESIZE, CXXTGLAREAREDRAW, RFRESH
C     .. Common blocks ..
      COMMON         WIDGET
C     .. Data statements ..
      DATA           FIRST/.TRUE./
C     .. Executable Statements ..
C
      IF (FIRST) THEN
C
C        first time initialization
C
         FIRST = .FALSE.
C
C        Set up the GL configuration information
C
         CONFIG(1) = GLX_NORMAL
         CONFIG(2) = GLX_RGB
         CONFIG(3) = 1
         CONFIG(4) = GLX_NORMAL
         CONFIG(5) = GLX_DOUBLE
         CONFIG(6) = 1
         CONFIG(7) = GLX_NORMAL
         CONFIG(8) = GLX_ZSIZE
         CONFIG(9) = GLX_NOCONFIG
         CONFIG(10) = 0
         CONFIG(11) = 0
         CONFIG(12) = 0
C
C        Create the GL area
C
         WIDGET = CXXTGLAREAINITIALIZE(WIN,CONFIG)
C
C        Register the redraw procedure with the widget
C
         CALL CXXTGLAREAREDRAW(WIDGET,RFRESH,0)
      END IF
C
C     Match the gl area size with the window size
C
      CALL CXXTAREARESIZE(WIDGET,WIN)
C
      RETURN
      END
      SUBROUTINE RFRESH(WIDGET,CLIENT)
C
C     Redraw procedure
C
C     .. Scalar Arguments ..
```

```
      INTEGER            CLIENT, WIDGET
C     .. Local Arrays ..
      REAL               P(2)
C     .. External Subroutines ..
      EXTERNAL           BGNCLO, CLEAR, CPACK, ENDCLO, PERSPE, RESHAP,
     *                   SWAPBU, TRANSL, V2F
C     .. Executable Statements ..
C
C     Configure the viewport
C
      CALL RESHAP
C
C     Clear the screen
C
      CALL CPACK(11563168)
      CALL CLEAR
C
C     Do some example GL stuff
C
      CALL PERSPE(400,1.0,0.5,5.0)
      CALL TRANSL(0.0,0.0,-4.0)
      CALL CPACK(-1)
      CALL BGNCLO
C
      P(1) = 0.0
      P(2) = 0.0
      CALL V2F(P)
C
      P(1) = 1.0
      P(2) = 1.0
      CALL V2F(P)
C
      P(1) = 1.0
      P(2) = -1.0
      CALL V2F(P)
C
      P(1) = -1.0
      P(2) = -1.0
      CALL V2F(P)
C
      P(1) = -1.0
      P(2) = 1.0
      CALL V2F(P)
C
      CALL ENDCLO
      CALL SWAPBU
C
      RETURN
      END
```

Figure 9–3 shows what the sample output from this module looks like.

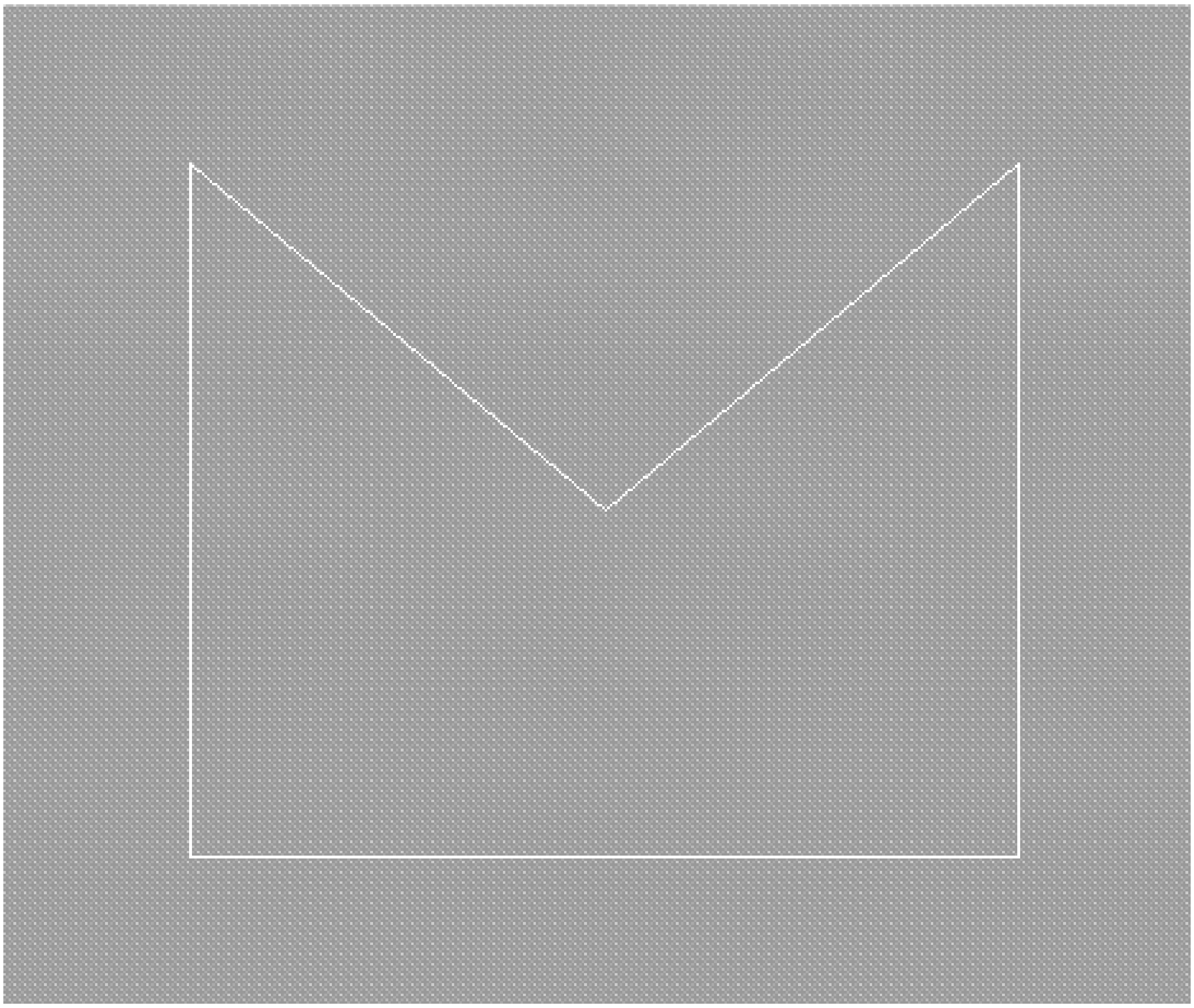


Figure 9–3 Output from the XGLArea Routine

9.2. Sharing Module Executables

You can save a significant amount of disk space by combining several related modules into a single executable image. This means you have only one executable file for all the related modules. If these modules were combined into one in the GUI, you could end up with a very complicated module control panel. To avoid this, IRIS Explorer allows each module sharing a single executable to have its own control panel. Each control panel is defined in a separate module resources file with the same name as the module itself.

This section describes how to create a single executable, or combination module, in the Module Builder and suggests what part of the library API to call from the combined module code.

9.2.1. Types of Shared Executable

A combination module determines its actions in two ways. It can:

- keep a table of known actions and make a selection from the action table based on the module name which invoked the executable. ImageVision Library modules such as *AddImg* and *XorImg* act in this way.

- make use of an interpreter module that reads a script or program provided on an input port, then performs the actions dictated by the script. The *DataScribe* and *LatFunction* modules function on this principle.

Both *AddImg* and *XorImg* have module resource files that describe their control panels and ports. In addition, on the Build Options menu, *AddImg* has *Alternate Executable* selected, with the name *XorImg* displayed in the executable name field. The *XorImg* module does not have *Alternate Executable* selected, so its name field is identical to the module name. it is clear that both *AddImg* and *XorImg* share the *XorImg* executable.

When the modules are built, the *XorImg Makefile* rules cause the *XorImg* executable to be compiled, linked, and installed along with *XorImg.mres.* The rules for *AddImg* cause only its module resources file to be installed, thus referring to the *XorImg* binary. When either module is started, the same executable is run. Enough information is provided to the *XorImg* binary that it can determined whether it was invoked as *AddImg* or *XorImg*, and, consequently, which set of functionality to provide.

The *XorImg* user function includes a call to the API routine *cxModuleNameGet*, which returns the name of the module that was invoked: in this case, *AddImg* or *XorImg.* The user function then compares the name to a table of known operations to determine whether to *add* or *xor* two images. The module can also use this name to determine its expected port names and to access its port data.

The DataScribe modules provide another example of the use of alternate executables. Each customized DataScribe module is simply a module resource file that expects an input script and knows which set of ports to query, based on the contents of the script. Every DataScribe module refers to the generic DataScribe module for its executable. The DataScribe module uses the input script, not *cxModuleNameGet*, to direct its activities, which are to interpret the input script and carry out specific actions on data files or IRIS Explorer ports.

9.2.2. Using an Alternate Executable

An ordinary module, called *MyModule* for example, can use an executable name derived from its *mres* file, *MyMod.mres*. But some modules, notably the ImageVision Library™ modules, refer to a different executable. Since many modules can refer to a single executable, those modules need not build the executable; some other similarly named module does that. For example, *XorImg* is the module that builds the executable *XorImg*; all the other ImageVision Library modules refer to it.

The *Alternate Executable* item in the Build Options window of the Module Builder lets you:

- Select whether you wish to have such an alternate executable.
- Type in the name of the executable file.

In the case of a combined executable, each module has its own module resources file, but only one of the modules creates and installs an executable. The executable is then responsible for determining which operation to invoke upon firing.

9.3. Building Loop Controller Modules

There is no difference in programming between the user code of a loop controller and that of an ordinary module. The loop control work is handled in the MCW. There are certain API routines, discussed below, that simplify writing loop controller modules.

However, there are some capabilities a loop controller should have. The module must have at least one input and at least one output port. It needs to know that the loop continues by default if new data is sent into the loop, and the loop stops if no new data enters the loop. If the module is to continue or terminate a loop, it must be able to evaluate the condition for the loop. The module should also be able to break its loop, to prevent the loop from iterating indefinitely.

Note: A module that flushes multiple data sets, such as *Streakline*, should not be given loop controller status. Multiple data flushes from a controller module can lead to an exponential increase in loop data that will paralyze your system.

9.3.1. Using the Loop Controller API

There are four API subroutines you can use when writing a loop controller module: *cxLoopBreak*, *cxLoopControlArc*, *cxLoopCtlr* and *cxLoopIteration.* The three routines, *cxLoopCtlr*, *cxLoopControlArc* and *cxLoopIteration*, are Boolean queries that deal respectively with loop controller status of the module, control arc status of a connection, and looping, as opposed to non−looping, invocation of an iteration. You can use these calls to get information about the actions and state of a loop controller module.

The *cxLoopBreak* routine halts a current loop iteration, as well as marking all pending iterations for cancellation by the loop controller. If the controller puts out "end−of−loop" data, those datasets should be assigned to the output ports when *cxLoopBreak* is called, as the user function will not be called again for that loop iteration. In programming a loop controller module, you must use the API routine *cxLoopBreak* to terminate iteration of a loop. This causes the MCW to cancel all pending iterations of the loop and return the module to a quiescent state.

Data is put on the synchronisation port *Loop Ended* by the MCW when the iteration of a loop is terminated. You can prevent data being put on this port, as well as the *Firing Done* port, by calling *cxOutputNoSync*.

You may also terminate a loop by sending out a sync frame, which is a frame with no new data, on the looping outputs. You must take care here, as the Map Editor user might wire a loop on ports you did not expect to be used, causing the module not to terminate the loop. If the synchronisation port *Firing Done* is wired into the loop, as data is always put on this port when the module fires successfully, then the only way to terminate the loop is by calling *cxLoopBreak.* It is safest to use *cxLoopBreak* in all cases.

9.3.2. Using Generic Controller Modules

The *While* and *Repeat* loop controller modules simply examine the condition for the loop, which is a parameter value, and copy a pointer over from the input to the output port. These are "typeless" modules which have *cxGeneric* input and output ports, and do not interrogate their generic inputs.

You can copy the *modulename.mres* file of these modules from directory */usr/explorer/modules* and modify each one slightly to create loop controllers that pass additional data sets through, or accept particular data types. Just rename the module and augment the port list. Under the Build Options menu, set the alternate executable to be the name of the module you copied, e.g. *While*, (see **"Using the Build Menu"**) and run mbuilder on the new module.

While module has the ports listed in **Table 9−2**:

Input Ports	Output Ports
Condition	Final Value 0
Initial Value 0	Final Value 1
...	...
LoopIn Value 0	LoopOut Value 0
LoopIn Value1	LoopOut Value1

Table 9−2 While Module Ports

Substitute your own name for the "Value N" part of the port name. The module must match initial/final and loop in/loop out values. It can do this by finding the keywords "Initial", "Final", "LoopIn" and "LoopOut" on the port names.

The For Module

The For loop is a specific case of the While loop, and the *For* module is designed to obviate the need for parameter functions. The *For* module has sliders for changing initial, final and current parameter values and parameter logic, and it also has a *Refire* button. The module evaluates a current value and waits for the return value. The loop index uses double precision floating point values. The current value is put out even when the loop terminates, so that other downstream modules can query it.

9.3.3. The cxGeneric Data Type

The *cxGeneric* data type is found on input and output ports in the generic loop controller modules, which are*For*, *While*, *Repeat* and *Trigger*. You can wire any port types into or out of a *cxGeneric* port.

The purpose of *cxGeneric* is to allow passage of all data types through a module. It is actually more of an encapsulation system than a true type, since it does not operate on the data at all.

9.4. Installing Interpreter Modules

When you install an interpreter module, you can list any extra files you want installed with the module in the Module Builder. The Build Options window has a text slot for listing them (see **Figure 2−17**). Any module can cause extra files to be sent to *$EXPLORERUSERHOME/modules* when a *make install* command is issued.

The Module Builder will check for information pertaining to interpreter modules, that is, those with an alternate executable name. Such a module may have a file *$EXPLORERHOME/lib/modulename.interp* which lists the interpreter module, optional file suffix to use, and widget normally containing the name of the interpreter script file.**Table 9−3** lists three examples:

Interpreter Module	Module Executable
LatFunction.interp	LatFunction
GenericDS.interp	GenericDS
MyInterp.interp	MyInterp

Table 9−3 Interpreter Module Files

These are the rules that the Module Builder follows during installation.
- If there is no file, then no addition is made to the list of files to install.
- If it finds a *modulename.interp* file, then all alternate executable modules relying on *modulename* will have the contents of the text or file−browser widget added to their file list.
- If the *modulename.interp* file has anything other than NULL for suffix, then in the absence of the named widget, the file *modulename.suffix* will be added to the installation list.
- If the *modulename.interp* file has NULL for suffix, then no other files will be added to the installation list.

9.5. Hook Functions

Hook functions allow you to gain control of a module when specific events occur in the Module Control Wrapper (MCW). The MCW passes control to the hook function whenever these events occur. A hook function can be linked to any one of the following events in an IRIS Explorer session. These events are:
- creating the module
- initializing the module
- destroying the module
- making a connection to an input or output port in the module
- breaking a connection to an input or output port in the module

Once you have control through the hook function, you can call a subroutine to carry out an activity related to the main user function. For example, the *Render* module has an initialization hook function that is triggered when the module is launched in the Map Editor. This function initializes the Open Inventor Library and the OpenGL drawing area, among other things. When the subroutine has completed its task, control is returned to the MCW.

A hook function table is contained in the Generic Wrapper (GW) of every module. The table is empty by default. To fill in the default table with your own hook functions, you must:
- Write the hook functions you want and define them in the user function
- List each hook function in the Hook Functions window, located on the Build menu.

Hook functions written in C++ must have a C linkage. You can do this by declaring the function:

```
extern "C" my_hook_fcn( )
{ ...
}
```

9.5.1. Hook Function Calling Sequence

There are two forms of the calling sequence, one used in the initialization and removal hook functions, and one used in the connection and disconnection hook functions.

For the latter, the MCW passes the name of the affected port (`portName`) and an integer that identifies that particular connection (`linkID`). The `linkID` values are unique with respect to a given port and increase throughout a single IRIS Explorer session.

The calling sequence of each type of hook function is given below. You can give a hook function or subroutine any name you like; these are merely examples.

Creating a module:

C:	*void CreateHook(void);*
Fortran:	*subroutine CreateHook*

The MCW calls this function when the module is created, before connections are made and parameters are delivered. Parameter values, particularly drawing area window IDs, are meaningless at this time.

Connecting input ports:

C:	*void ConnectInput(char*portName,int linkID);*
Fortran:	*subroutine ConnectInput(portName,linkID)* *character*(*) portName* *integer linkID*

The MCW calls this function after the user has connected input `portName` to some output.

9.5.2. Writing Hook Functions

Hook functions are ordinary procedures, which you must write before the MCW can use them. The hook functions and user function must both have either C or Fortran linkage; mixing linkages is not permitted.

The easiest way to generate hook function prototypes is to define them in the Module Builder as follows:

1. Select *Hook Funcs* from the Build menu in the Module Builder window to open the window (see **Figure 9–4**).
2. Type the name you have given each hook function in the slot next to its description. For example, you may type "DeleteHook" as the function name in the slot next to "Remove Hook Function".
3. When you have finished listing the hook functions for the module, click on *OK.* When you select *Build*, the Module Builder creates the hook function table.

   It also creates hook function prototypes in the correct language and places them in the prototype user function file created when you select *User Function* from the Prototypes menu. You need merely fill in the computation.

   Then create the function prototype in the Module Builder (see **"Hook Function Prototypes"**).

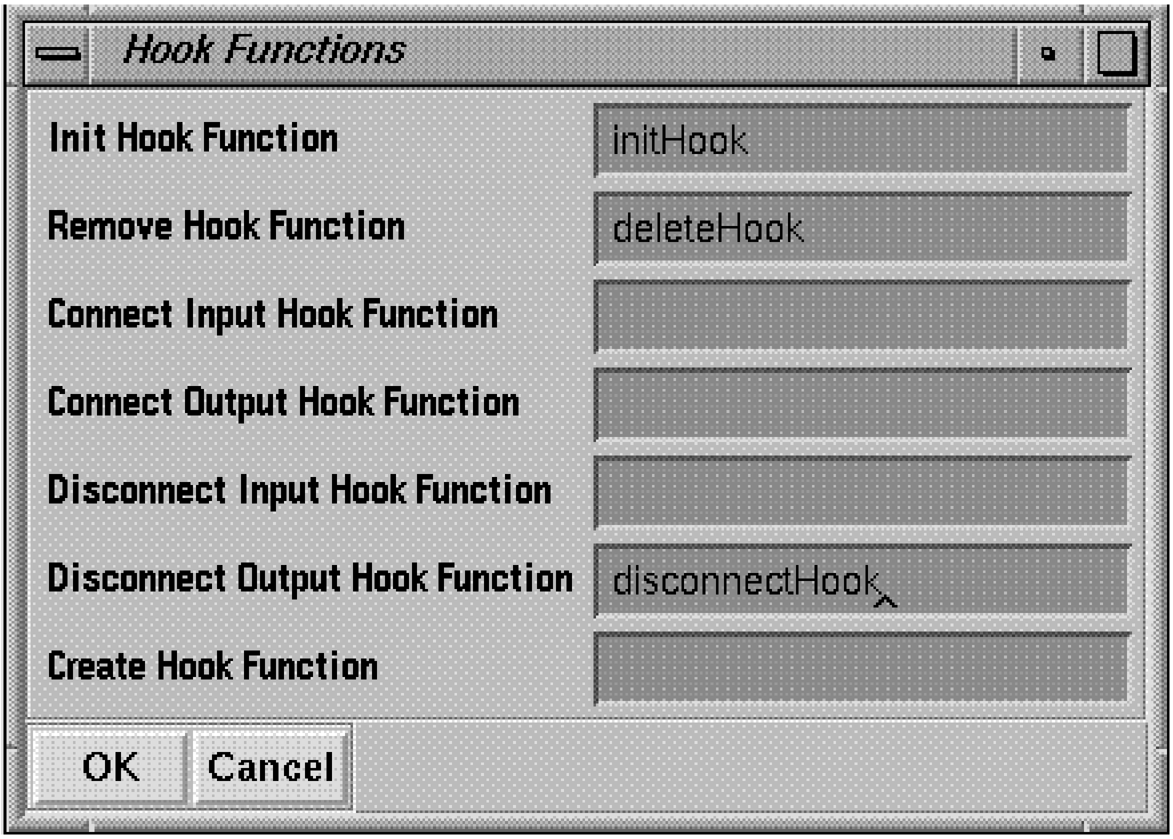


Figure 9–4 The Hook Function Window

Here are some points to note when you are writing a hook function.

- When you request that a module be deleted, a "destroy" request is sent to the module. If the module is firing, the "destroy" request is not processed until the firing is completed, at which point the removal hook function will be called. However, if you delete the module a second time *before* it finishes firing, you kill the module's process and the removal hook function cannot be called.
- If the module retains cached data, you must decrement the reference counts when the module is deleted. The module removal hook is responsible for removing any persistent data in shared memory. If the process is killed, however, this cleanup does not take place.

Note: If you have old modules that define *cxHookTable*, you must remove this definition from your source code. If you do not, you will see a multiply–defined symbol. This may make debugging more difficult, and on machines such as the Sun, the module will not build.

9.5.3. Examples

The following example shows the use of hook functions to allocate storage space at module initialization, to get information about port connections, and to delete persistent lattice data in shared memory when the module is destroyed. Code and resources files may be found in the directories */usr/explorer/src/MWGcode/Advanced/C* and */usr/explorer/src/MWGcode/Advanced/Fortran.*

C Version:

```
#include <cx/DataAccess.h>

cxLattice *permanentLattice = NULL;

/*********************************************************************
 * This module shows how hook functions may be used
 *
 * Each hook function generates output, so the calling of each may be
 * followed as the module starts up, is connected to other modules,
 * is fired, and gets deleted.
 *********************************************************************
 */
void HookFuncModule(cxLattice *lat)
{
  /* You may complete the user function by adding lines of code here
   * This output will appear every time the modules is fired
   */
  printf("User function called.\n");

  /* A private copy of the input lattice is stored by the user
   * This storage would be leaked on module destruction, unless
   * we increment the reference counter for this storage, and
   * ensure to delete it in the hook destroy function
   */
  if (permanentLattice == NULL)
    {
      permanentLattice = cxLatDup(lat,1,1);
      cxDataRefInc(permanentLattice);
      printf("increase the reference count for the private lattice\n");
    }
}

#include <cx/DataOps.h>

int *myLinkArray = NULL;

/*********************************************************************
 * Initialisation hook function in C
 *
 * Some storage is allocated for use while the module exists
 *********************************************************************
 */
void myInitFunc()
{
  /* You may complete the initialisation function by adding code here
   * for example to initialise the geometry library
   * This output will only appear when the module is started up
   */
  printf("Initialisation hook function called.\n");

  /* Initialise the link array */
  if (myLinkArray == NULL)
    myLinkArray = (int *) cxDataCalloc(100, sizeof(int));
}
```

```
/*******************************************************************
 * Connection hook function in C.
 *
 * This shows how one might save the port name associated with every
 * connection. Storage allocation of "myLinkArray" is done in the
 * initialisation hook function.
 */

void myConnectFunc(portName, linkID)
     char *portName;
     int linkID;
{
  extern int *myLinkArray;

  /* Save the association of port and linkID. The link tag may be
   * negative, indicating a widget connection or a parameter function
   * (pfunc) connection. If so, ignore it.
   */
  printf("linkID=%d\n", linkID);
  if (linkID >= 0 && linkID < 100)
    {
      myLinkArray[linkID] = portName;
      printf("link %d is named %s\n",linkID, myLinkArray[linkID]);
    }
}

#include <cx/DataAccess.h>

/*******************************************************************
 * Destroy hook function in C.
 *
 * This procedure shows how one might write a "desctruction" hook
 * function.  When the map editor user destroys a module, this
 *  procedure, if defined, will be called.
 *
 * Typically, this is useful if the module caches old data inputs.
 *  This provides a place to put the code that can decrement the
 * reference counts on the data.
 *******************************************************************
 */
void myDestroyProc(void)
{
  extern cxLattice *permanentLattice;
  int i;

  /* Here, one might adjust the reference counts on any cached data
   * In this case, there is only one lattice
   */
  cxDataRefDec(permanentLattice);
  printf("User destroy hook function called.\n");
}
```

Fortran Version:

```
      SUBROUTINE HKFUNC(LAT)
C
      INCLUDE '/usr/explorer/include/cx/DataAccess.inc'
C
C     This module shows how hook functions may be used
C
C     Each hook function generates output, so the calling of each may be
C     followed as the module starts up, is connected to other modules,
C     is fired, and gets deleted.
C
C     .. Scalar Arguments ..
      INTEGER           LAT
C     .. Scalars in Common ..
      INTEGER           PRVLAT
C     .. Local Scalars ..
      INTEGER           FIRST
C     .. External Subroutines ..
      EXTERNAL          CXDATAREFINC
C     .. Common blocks ..
      COMMON            /CACHE/PRVLAT
C     .. Data statements ..
      DATA              FIRST/1/
C     .. Executable Statements ..
C
C     You may complete the user function by adding lines of code here
C     This output will appear every time the modules is fired
C
      PRINT *, 'User function called.'
C
C     A private copy of the input lattice is stored by the user
C     This storage would be leaked on module destruction, unless
C     we increment the reference counter for this storage, and
C     ensure to delete it in the hook destroy function
C
      IF (FIRST.EQ.1) THEN
         PRVLAT = CXLATDUP(LAT,1,1)
         CALL CXDATAREFINC(PRVLAT)
         PRINT *, 'increase the reference count for the private lattice'
         FIRST = 0
      END IF
C
      RETURN
      END

      SUBROUTINE MYINIF
C
C     Initialization hook function in Fortran
C
C     .. Arrays in Common ..
      LOGICAL          MLSTAT(100)
C     .. Local Scalars ..
      INTEGER          I
C     .. Common blocks ..
      COMMON           /CONN2/MLSTAT
```

```
C     .. Executable Statements ..
C
C     You may complete the initialisation function by adding code here
C     for example to initialise the geometry library
C     This output will only appear when the module is started up
C
      PRINT *, 'Initialisation hook function called.'
C
C     Initialise the link tags to .FALSE.
C
      DO 20 I = 1, 100
         MLSTAT(I) = .FALSE.
   20 CONTINUE
C
      RETURN
      END

      SUBROUTINE MYCONF(PRTNAM,LNKTAG)
C
C     This connect hook function shows how one might save the
C     port name associated with every connection.
C     The name is saved in a common variable.
C     The initialisation hook function sets array MLSTAT to .FALSE.
C
C     .. Scalar Arguments ..
      INTEGER           LNKTAG
      CHARACTER*(*)     PRTNAM
C     .. Arrays in Common ..
      LOGICAL           MLSTAT(100)
      CHARACTER*32      MLARAY(100)
C     .. Local Scalars ..
      INTEGER           I
C     .. Common blocks ..
      COMMON            /CONN1/MLARAY
      COMMON            /CONN2/MLSTAT
C     .. Executable Statements ..
C
C     The LNKTAG may be negative, indicating that the connection is
C     either a widget or a parameter function (pfunc) in which case
C     this code ignores it.
C
      IF (LNKTAG.GE.0 .AND. LNKTAG.LT.100) THEN
         MLARAY(LNKTAG+1) = PRTNAM
         MLSTAT(LNKTAG+1) = .TRUE.
      END IF
C
      DO 20 I = 1, 100
         IF (MLSTAT(I)) THEN
            PRINT *, 'Link number', I - 1, ' is named ', MLARAY(I)
         END IF
   20 CONTINUE
C
      RETURN
      END
```

```
      SUBROUTINE MYDSTF
C
C     Destroy hook function in Fortran.
C
C     This procedure shows how one might write a "desctruction" hook
C     function.  When the map editor user destroys a module, this
C     procedure, if defined, will be called.
C
C     Typically, this is useful if the module caches old data inputs.
C     This provides a place to put the code that can decrement the
C     reference counts on the data.
C
C     The name of this procedure must be entered into the hook function
C     table in the module builder.
C
C     .. Scalars in Common ..
      INTEGER          PRVLAT
C     .. External Subroutines ..
      EXTERNAL         CXDATAREFDEC
C     .. Common blocks ..
      COMMON           /CACHE/PRVLAT
C     .. Executable Statements ..
C
C     Here, one might adjust the reference counts on any cached data
C     In this case ther is only one lattice
C
      CALL CXDATAREFDEC(PRVLAT)
   20 CONTINUE
      PRINT *, 'User destroy hook function called.'
C
      RETURN
      END
```

9.6. Understanding Reference Counting

IRIS Explorer transfers data between modules using shared memory if the modules are on the same machine and the machine supports shared memory. The advantage of using shared memory is that large quantities of data can be transferred from one module to another with very little communication overhead. All modules access the shared memory arena simultaneously, so only the address of the data has to be transferred, not the data itself. However, the data must be managed somehow, and this is done by means of *reference counting*.

The reference count can be thought of as the number of places that wish to keep a persistent copy of specific data. These places are also responsible for decrementing the count when they have finished with the data.

9.6.1. Managing Data in Shared Memory

All modules are responsible for collectively managing the data in shared memory. One module creates data and sends it to other modules. When all the modules are finished with that data, the space it occupies must be reclaimed. IRIS Explorer uses reference counting to manage this process. With reference counting, a module does not need to know anything about which other modules use the data. It must simply perform certain bookkeeping actions on the data it references, so that the system knows when the data should be reclaimed.

Reference counting is essential when using shared memory, but the mechanism is also general enough to be used by a single process on private data. Thus, modules executing on a machine that does not support shared memory can still use the same code for managing data memory.

As long as every module does its bookkeeping correctly and consistently, there will be no memory leaks, and data will not be reclaimed while a module still refers to it.

Caution: The automatically generated Module Data Wrapper cannot completely recover if the shared memory arena becomes full. It is possible that, if there is not enough memory for the computational function or for the MDW itself, the MDW may leak memory.

9.6.2. How Reference Counting Works

An integer is kept with each data object, to record the number of places where the data is used. When a module receives data, it increments the integer count to show that the data is being used in an additional place. When the module is finished with the data, it decrements the reference count. When the count becomes zero, the data is no longer needed and that module deletes the data.

For example, the module's input and output ports store data for the module, so they must manipulate the reference count. When the module puts data on the output port, the output port decrements the count of the old data on the port and increments the count of the new data. The output port retains use of the data so it can be sent to modules as they are connected to the port.

When a module receives new data, the input port decrements the count on the old data on that port and increments the count on the new data. The input port retains use of the data until it is replaced, which means the data is available every time the module fires, even if the data has not changed.

The API subroutines *cxDataRefInc* and *cxDataRefDec* are used to increment and decrement the reference count.

9.6.3. How IRIS Explorer Implements Reference Counting

IRIS Explorer implements a less rigid version of reference counting than that defined above. Data objects are created with a reference count of 0 because nothing has claimed responsibility for decrementing the count. If the count on such an object is decremented immediately, it becomes −1. The object is deleted and no error is reported.

The default count of 0 simplifies coding of the user function cases. A data object is created and (usually) placed on the output port. The output port increments the count and assumes responsibility for maintaining the count, thus freeing the user function from that responsibility and allowing it to ignore the data. If data objects were always created with a reference count of 1, then you would have to decrement the count of each object after placing it in the output port.

A user function may want to allocate a data object for temporary storage so that the same data can be output more than once. In this case, the user function does need to increment the count, ensuring that the data will persist as long as the user function needs it. When it is no longer required, the user function should decrement the count.

A module may have more than one reference to any data, so the value of the reference count may be greater than the number of modules using it. It is easier to code a module when you can allow it to have multiple references to the data, because the user function does not have to maintain exactly one reference count.

9.6.4. Reference−Counting Data Types

All IRIS Explorer system and user−defined data types are reference−counted. Data objects may contain references to other reference−counted objects. For example, the *cxLattice* and *cxPyramid* data types contain references to *cxData*, *cxCoord*, *cxConnection*, and *cxLattice* objects. The reference−counting scheme handles arbitrary sharing of IRIS Explorer data objects within other objects, just as data objects are shared between modules. For example, a*cxData* object can be contained in more than one *cxLattice*. Likewise, a *cxLattice* object can be contained in more than one *cxPyramid* and referenced by more than one module.

Reference Counts for Contained Data Types

The IRIS Explorer routines that set the contents of *cxLattice* and *cxPyramid*, for example, *cxLatPtrSet* and *cxPyrSet*, correctly manage the reference counts of the contained data types. When the count of a *cxLattice* or *cxPyramid* reaches 0 or −1, the generic data deletion routine uses the data type information to decrement the reference count of each contained data type.

The automatically generated interface routines *cxLatPtrSet*, *cxPyrSet*, and *cxPyrLayerSet* correctly manage the reference counts for the data objects they contain. You are strongly encouraged to use them.

The *cxGeometry* data type contains references to the field data of the IRIS Explorer nodes within an Open Inventor scene graph. This reference counting is managed within the geometry API and the IRIS Explorer node classes and you will never need to manipulate these directly. Note that this is in addition to the Open Inventor reference counting for nodes and referencing and deleting nodes should be programmed as usual for an Open Inventor scene graph, when using Open Inventor directly instead of the geometry API. It is important to ensure that a scene graph is correctly deleted when a module exits, lest it leak the shared memory in its nodes. .

Manipulating Reference Counts

Module writers rarely have to manipulate reference counts, but there are times when it is necessary. This happens most commonly when the module needs to retain data that the system would normally have released. For example, a module that outputs a moving average of the last three lattices delivered to it might need to keep the lattices from two previous firings. When a new lattice arrives on a wire, it decrements the reference count on the lattice that it replaces. This can result in the memory for the old lattice being deallocated. To stop this happening, the module should increment the reference count of the lattices it wants to save. This process applies to all data types.

If a module increments the reference count to save data, it is very important that it decrement the reference count to free the memory when the data is no longer needed. Not properly balancing reference count increments and decrements can result in memory leaks which degrade performance.

9.6.5. Shared Memory Arena

The shared memory arena is a fixed−size resource. It is at most 256 MB, and may not reach that size. Because it is stored in a memory−mapped file, its size is limited by the available disk space and the command line and configuration options that are specified when the user starts IRIS Explorer; therefore, it is possible for data memory allocation routines to fail, no matter how large the arena might be, because the arena size may exceed the amount of available memory.

Allocating Memory Efficiently

It is very important to keep the limitations of memory allocation in mind when you write modules. The memory requirements of the system fluctuate, so many out−of−memory situations are transient. IRIS Explorer tries to allocate memory several times over a short period before returning an error flag, at which time an error message is sent to the Map Editor, which in turn generates a pop−up dialog box.

Modules should check the data allocation flag after every IRIS Explorer routine that allocates data memory. The *cxDataAllocErrorGet* routine returns TRUE (non−zero) if data could not be allocated. See *cxDataAllocErrorGet* for a list of the routines that allocate data memory. (For reasons of simplicity such error checking has been omitted from many of the example programs.)

If a module ignores allocation errors, it may crash due to a bad pointer. Even worse, it may cause another module to crash. This kind of module failure can be very hard to diagnose. If a module crashes, it cannot decrement reference counts to data to which it has references, so the data will be leaked. This makes a bad situation even worse.

Recovering from Memory Errors

The IRIS Explorer modules provided with the system do not attempt memory allocations after an error has been detected. They simply reclaim any memory they have already allocated and return from the user function. Unfortunately, the module may have successfully allocated several data objects, which it must reclaim before returning. The recovery code can get quite complicated, especially for a module that creates many data objects or creates *cxPyramid* or *cxGeometry* objects. It helps a great deal if all data is allocated in one place, where it is easier to keep track of what should and should not be recovered. This is not always possible, but grouping code in modules to streamline data allocation helps to eliminate programming errors.

If the memory requirements are smaller when no modules are firing, the *Fire Now* option on the module pop–up menu will cause it to execute again. If the module still does not have enough memory, you can free some data memory by deleting modules that are consuming large portions of memory.

If a module runs out of memory, it can:

- retry the request itself, in the hope that memory has become available
- reduce its memory requirements before attempting a new request

However, each failed attempt still generates the pop–up dialog box in the Map Editor.

9.6.6. Examples of Memory Recovery

The following examples show how to recover from allocation errors for some of the IRIS Explorer data types. They also illustrate some coding styles for recovery.

Note: These examples are in C only because they are meant to illustrate specific techniques, rather than provide reusable code. Templates for these routines may be found in directory */usr/explorer/src/MWGcode/Advanced/C*.

Example 1

The first example allocates a *cxParameter* and two *cxLattices*. It uses a single memory recovery routine, *memCleanup*, which takes advantage of the fact that *cxDataRefDec* accepts a `NULL` pointer; therefore, *memCleanup* can be called at any point in the program.

```
#include <cx/DataAccess.h>
#include <cx/DataOps.h>

cxParameter *parm = NULL;
cxLattice *lat1 = NULL;
cxLattice *lat2 = NULL;

void memCleanup()
{
  cxDataRefDec( parm );
  cxDataRefDec( lat1 );
  cxDataRefDec( lat2 );
}

void userFunction()
{
  parm = cxParamNew();
  if (cxDataAllocErrorGet())
    return;                   /* no cleanup required */
  lat1 = cxLatNew(...);
  if (cxDataAllocErrorGet())
    {
      memCleanup();
      return;
    }
```

```
  lat2 = cxLatNew(...);
  if (cxDataAllocErrorGet())
    {
      memCleanup();
      return;
    }
  /*
   * Et cetera...
   */
}
```

Example 2

The second example shows the use of recovery labels in the reverse order of the data allocation. Execution falls through the labels, unwinding the successful allocations.

```
#include <cx/DataAccess.h>
#include <cx/DataOps.h>

void        userFunction()
{
  cxParameter *parm = NULL;
  cxLattice *lat1 = NULL, *lat2 = NULL;

  parm = cxParamNew();
  if ( cxDataAllocErrorGet() )
    goto parm_Failed;
  lat1 = cxLatNew(...);
  if ( cxDataAllocErrorGet() )
    goto lat1_Failed;
  lat2 = cxLatNew(...);
  if ( cxDataAllocErrorGet() )
    goto lat2_Failed;

  /* ... */

  /* normal return */
  return;

 lat2_Failed:
  cxDataRefDec( lat1 );
 lat1_Failed:
  cxDataRefDec( parm );
 parm_Failed:
  return;
}
```

Example 3

The third example shows the memory for lattice data and coordinates allocated separately. Recovery becomes more complicated in this case. Once *cxLatPtrSet* stores *data* and *coords* into *lat1, lat1* has reference counts on *data* and *coords* and is responsible for decrementing their counts. Once succesfully stored in the lattice, they should*not* be decremented by the user function.

```
#include <cx/DataAccess.h>
```

```
#include <cx/DataOps.h>

void userFunction()
{
  cxLattice *lat1, *lat2;
  cxData *data;
  cxCoord *coords;

  lat1 = cxLatRootNew(...);
  if( cxDataAllocErrorGet() ) return;
  data = cxDataNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( lat1 );
      return;
    }
  coords = cxCoordNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( lat1 );
      cxDataRefDec( data );
      return;
    }
  /* cxLatPtrSet doesn't allocate memory */
  cxLatPtrSet( lat1, data, NULL, coords, NULL );

  lat2 = cxLatNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( lat1 ); /* NOT data or coords */
      return;
    }
  /* Etcetera .. */

  /* normal return */
  return;
}
```

Example 4

The fourth example shows a recovery strategy for the *cxPyramid* data type. It is similar to that required by *cxLattice.*

The base lattice and the lattice/connection pair are managed similarly to the data and coordinates for *cxLattice.* Once successfully stored in the pyramid, they should *not* be decremented by the user function.

Also, *cxPyrLayerSet* allocates memory if the number of pyramid layers is to be increased, so *cxDataAllocErrorGet* should be checked after a call to *cxPyrLayerSet* if there is any chance that the number of layers will increase. If *cxPyrLayerSet* fails, you must decrement the lattice and connection manually because they were not successfully stored in the *cxPyramid.*

```
#include <cx/DataAccess.h>
#include <cx/DataOps.h>

void userFunction()
{
  cxPyramid     *pyr;
```

```
  cxLattice    *base, *layer, *other;
  cxConnection    *conn;

  base = cxLatNew(...);
  if( cxDataAllocErrorGet() ) return;
  pyr = cxPyrNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( base );
      return;
    }
  cxPyrSet( pyr, base );         /* no allocation */

  layer = cxLatNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( pyr );        /* not base */
      return;
    }

  conn = cxConnNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( pyr );
      cxDataRefDec( layer );
      return;
    }

  cxPyrLayerSet( pyr, i, conn, layer );
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( pyr );
      cxDataRefDec( layer );       /* layer and conn not in pyr */
      cxDataRefDec( conn );
      return;
    }

  other = cxLatNew(...);
  if( cxDataAllocErrorGet() )
    {
      cxDataRefDec( pyr );
      return;
    }
  /* Etcetera .. */

  /* normal return */
  return;
}
```

Example 5

The *cxGeometry* data type complicates error recovery even more, since it contains a buffer of transcribed geometry specification commands. Each geometry specification routine adds commands to this buffer, which may cause the buffer to be enlarged, so *cxDataAllocErrorGet* must be checked after each command. In addition, the actual buffer location is not associated with the geometry object while primitives are added, so the *cxGeoBufferClose* routine must be called before the geometry object can be reclaimed.

In this example, the module performs operations if the error flag is not set. It can then check the flag at key points and perform cleanup if required.

The buffer may be expanded more than once for each geometry specification command, so it may contain a partial command when the data allocation fails. The partial command may confuse downstream modules so seriously that the contents of the geometry must be thrown away.

Because the geometry interface is a *delta protocol* (only changes are sent), any *cxGeoDelete* command will be lost after the local data structures have been modified. The downstream modules will misinterpret future geometry inputs and will appear corrupted. The best way to recover is to disconnect and reconnect the geometry output that ran out of memory.

Note: *cxGeoBufferClose* should *not* be called twice on a *cxGeometry* object.

```
#include <cx/DataAccess.h>
#include <cx/DataOps.h>
#include <cx/Geometry.h>

void        userFunction()
{
  cxGeometry *geo;

  geo = cxGeoNew();
  if ( cxDataAllocErrorGet() )
    return;
  cxGeoBufferSelect( geo );

  if ( !cxDataAllocErrorGet() )
    cxGeoRoot();
  if ( !cxDataAllocErrorGet() )
    cxGeoDelete();
  if ( cxDataAllocErrorGet() )
    goto cleanup_selected;

  cxGeoXformPush();
  if ( !cxDataAllocErrorGet() )
    cxGeoLinesDefine(...);
  if ( !cxDataAllocErrorGet() )
    cxGeoColorAdd(...);
  if ( !cxDataAllocErrorGet() )
    cxGeoXformPop();
  if ( cxDataAllocErrorGet() )
    goto cleanup_selected;

  cxGeoBufferClose( geo );
  if ( cxDataAllocErrorGet() )
    goto cleanup;

  /* ... */

  /* normal return */
  return;
```

```
 cleanup_selected:
  cxGeoBufferClose( geo );
 cleanup:
  cxDataRefDec( geo );
  return;

}
```

9.7. Extending the Power of Modules

Several subroutines in the API allow you to increase the flexibility of your module. You can:

- communicate with other devices and programs outside the IRIS Explorer environment, using *cxInputAdd*
- time the occurrence of events, using *cxTimerAdd*
- expand a filename into the directory path, using *cxFileNameExpand*

The use of these routines is described briefly in the next three sections.

9.7.1. Communicating with UNIX Processes

The subroutine *cxInputAdd* registers an open UNIX file descriptor that the module will monitor. The file descriptor can reference any UNIX device on the system that has a file descriptor. The module can request that it be informed when the device is ready for reading or writing, or when an exception condition exists. When one of these conditions is met, the Module Control Wrapper calls the user callback routine, which passes the data collected from the UNIX device to the IRIS Explorer module. *cxInputAdd* returns a unique handle, which allows the module to distinguish more than one such input device.

When you use *cxInputAdd*, you should note that the callback routine is not called when the module is firing, only when it is quiescent.

The related subroutine, *cxInputRemove*, cancels the UNIX file descriptor monitoring process. You can call it from the callback routine specified in *cxInputAdd*.

cxInputAdd is a handy tool for monitoring, and collecting data from, processes outside the IRIS Explorer environment. For example, you can start up and communicate with a data base server from your IRIS Explorer module.

An Example Module

This module demonstrates the use of *cxInputAdd*. The module has a single text type–in slot on its control panel, into which the user types a UNIX command. The module runs that command and routes the output to the file descriptor `fd`. When the output is available on the file descriptor, the `feedme` routine is called. This routine simply copies this output to the module's standard output. The code is in */usr/explorer/src/MWGcode/Advanced/C/InputAdd.c*. The resources file may be found in the same directory. To test the module type a command, for example *ls*, in the text slot and observe the results.

Note: There is no equivalent Fortran version of this module.

```
/*
 * Sample module to demonstrate the use of cxInputAdd()
 *
 * This module takes a single text type-in, into which should be typed a Unix
 * command.  It runs that command, and routes the command's output to its own
 * standard output.
 */

#include <stdio.h>
```

```
#include <cx/DataTypes.h>
#include <cx/ModuleCommand.h>

#ifdef __cplusplus
extern "C" {
#endif

void        selectTestFunc  (
unsigned char *command )
{
    FILE            *fd;
    void            *handle;
    void             feedme();

    if (strlen(command) == 0) {
 return;
    }

    /*
     * First, send the user's command off for invocation, and hang on to a
     * descriptor for the pipe.
     */
    if ((fd = popen(command, "r")) == NULL) {
 cxModAlert("Command failed.");
 return;
    }

    /*
     * Now, add this file descripter to the scheduler's select set. When
     * data arrives from the command, "feedme" will be called by Explorer.
     */
    handle = cxInputAdd(fileno(fd), cx_InputReadMask,
   feedme, (void *) fd);
    return;
}

/*
 * Input available callback.
 *
 * This is called when the file descriptor from the popen() call above is
 * available.  The cient data (data) argument is just the FILE * form of the
 * descriptor.
 */

void        feedme( data, fd, handle )
void       *data;
int         fd;
void       *handle;
{
    char             buf[BUFSIZ];
    int              len;

    /*
     * Read from the pipe.  If no data remains on the file descriptor
```

```
     * (EOF) then remove the file descriptor from the select set.
     * Otherwise, just write the data to the window.
     */
    len = read(fd, buf, sizeof buf);

    if (len <= 0) {
 printf("closing pipe\n");
 pclose((FILE *) data);
 cxInputRemove(handle);
    } else {
 write(1, buf, len);
    }
}

#ifdef __cplusplus
};
#endif
```

9.7.2. Timer Events

You can use *cxTimerAdd* to time the occurrence of certain events that relate to the module function. The module can request callback routines to be called after a delay of a specified number of milliseconds, or at specified intervals. For example, you can set the timer to operate once and then switch off, or to reset itself and continue to run.

The callback is made when the timer expires, but the user function must be quiescent at the time. If the timer expires while the user function is executing, the callback will not be made until it has completed execution, and the timing schedule will be thrown off.

You can use *cxTimerRemove* to cancel this routine.

The code for this example is in */usr/explorer/src/MWGcode/Advanced/C/Timer.c* and */usr/explorer/src/MWGcode/Advanced/Fortran/Timer.f*. Resources files may be found in the same directories.

C Version:

```
/*
 * Trigger module -- demonstrates cxTimerAdd()
 *
 * This module fires every X seconds, where X is the value on
 * the dial widget on its control panel
 */

#include <cx/DataTypes.h>
#include <cx/Timer.h>


/*
 * Callback routine for the timer.
 * All this does is causes the module to fire.
 */

void        myTrigger(void *handle)
{
    printf("Timer trigger called\n");
    cxFireASAP();
```

```
}

/*
 * Main user function.
 *
 * secs -- number of seconds to delay (from a dial)
 * newtime -- non-zero if the firing resulted from the dial changing
 */

void        trigger  ( long secs, long newtime )
{
    static void    *lastTimer = NULL;

    /*
     * If the dial changed, remove any old interval timers and register a
     * new one.
     */

    if (newtime) {
 if (lastTimer != NULL) {
     cxTimerRemove(lastTimer);
 }
 lastTimer = cxTimerAdd(secs * 1000, 1, myTrigger, &lastTimer);
    }
    printf("triggered\n");
}
```

Fortran Version:

```
      SUBROUTINE TRIGGR(SECS,NEWTIM)
C
C     User function.
C     Trigger module -- demonstrates cxTimerAdd()
C
C     This module fires every X seconds, where X is the value on
C     the dial widget on its control panel
C
C     SECS   - number of seconds to delay (from a dial)
C     NEWTIM - non-zero if the firing resulted from the dial changing
C
C      include "cx/Timer.inc"
      INCLUDE '/usr/explorer/include/cx/Timer.inc'
C
C     .. Scalar Arguments ..
      INTEGER           NEWTIM, SECS
C     .. Local Scalars ..
      INTEGER           LSTTIM
C     .. External Subroutines ..
      EXTERNAL          CXTIMERREMOVE, MYTRG
C     .. External Functions ..
      EXTERNAL          CXTIMERADD
C     .. Save statement ..
      SAVE              LSTTIM
C     .. Data statements ..
```

```
      DATA              LSTTIM/0/
C     .. Executable Statements ..
C
C     If the dial changed, remove any old interval timers
C     and register a new one.
C
      IF (NEWTIM.NE.0) THEN
         IF (LSTTIM.NE.0) THEN
            CALL CXTIMERREMOVE(LSTTIM)
         END IF
         LSTTIM = CXTIMERADD(SECS*1000,1,MYTRG,0)
      END IF
C
      PRINT *, 'TRIGGERED'
      RETURN
      END
      SUBROUTINE MYTRG
C
C     Callback routine for the timer.
C     All this does is causes the module to fire.
C
C     .. External Subroutines ..
      EXTERNAL         CXFIREASAP
C     .. Executable Statements ..
C
      PRINT *, 'TIMER TRIGGER CALLED.'
      CALL CXFIREASAP
C
      RETURN
      END
```

9.7.3. Expanding Filenames

You can use *cxFilenameExpand* to expand the name of a file fully to include the directory path. You can give it a string including, for example, ~ or *$<environment variable>*, and it will expand the variable into the full name. This is particularly useful because there is no standard C library routine for doing this.

The code for the following example is in */usr/explorer/src/MWGcode/Advanced/C/FilenameExpand.c* and the resources file *FilenameExpand.mres* may be found in the same directory.

Note: There is no equivalent Fortran version for this module.

```
#include <stdio.h>
#include <string.h>

#include <cx/DataAccess.h>

void expand(char *filename)
{
 /* terminate early if filename is empty */
 if ( strlen(filename) == 0 )
  return;

 /*
```

```
  The string returned by cxFilenameExpand must be copied
  if it is to be saved for future use.  Subsequent calls
  to cxFilenameExpand will write over this buffer.
 */
 printf("delivered filename: %s\n",filename);
 printf("expanded filename:  %s\n",cxFilenameExpand(filename));

 /* increment counter for %n */
 cxFilenameIndexIncrement();
}
```

9.8. Making Subdirectories for Module Development

This section explains how to set up a user directory in which modules can be made. It assumes some knowledge of UNIX program development, specifically, the *make(1)* program. IRIS Explorer uses an extension to *make* called *Imake* which allows us to distribute a portable development mechanism. With *Imake*, you do not directly create the *Makefile* files used by *make*. Instead, *Makefile* files are generated automatically from a file named *Imakefile* (see **Appendix A, "Using Makefiles"**). IRIS Explorer takes this one step further.

Within the subdirectories where you develop modules, the Module Builder creates the *Imakefile* for you from the information in the module resource file, and then it creates a *Makefile* from that.

There are two paths to follow at this point:

1. Try building the sample IRIS Explorer modules provided in source format within your own module tree.
2. Set up your own build system, skipping the attempt to build the IRIS Explorer modules.

Path 1 provides a good test of the installation of IRIS Explorer on your machine and shows a working example of module compilation. Path 2 is more direct, allowing you to begin module development immediately. In either case, you must first set up a new, empty module build tree. Each method is described in detail in the following sections.

9.8.1. Creating the Directory

The plan is to create a directory where you have write permission (for example, beneath your home directory) and then develop modules in subdirectories of that directory. One or more modules may be developed in each subdirectory and you can build one or all of them with a single command.

First, reset the *EXPLORERUSERHOME* environment variable to the directory you have chosen as the IRIS Explorer installation directory; and place *$EXPLORERHOME/bin* in your execution path:

```
setenv EXPLORERUSERHOME /usr/local/explorer
set path = ($EXPLORERHOME/bin $path)
```

Note: Individual users do not always have write access to */usr/local/explorer* on a multi–user system. In this case, set *EXPLORERUSERHOME* to a directory in which you do have write permission.

Decide where you want to do module development and create that directory. *~/explorer/modules* is a good choice.

```
mkdir ~/explorer/modules
```

If *~/explorer* does not already exist on your system, create it by typing:

```
mkdir ~/explorer
```

From here, you can proceed through one or the other, or both, of the following sections, depending on your purpose.

9.8.2. Building Sample IRIS Explorer Modules

If you decide to build the sample IRIS Explorer modules provided in source format, you can compile them in your own source directory. To do this, first copy the sources into your directory:

```
cp -r /usr/explorer/src/MWGcode/* ~/explorer/modules
```

This assumes that *explorer/modules* is a directory in your home directory where you will build these modules. Next, make a *Makefile* in that directory:

```
cd ~/explorer/modules
cxmkmf
```

Next, make sure that your EXPLORERUSERHOME environment variable is set to a place where you want these modules installed, for example:

```
setenv EXPLORERUSERHOME /usr/tmp/modules
mkdir $EXPLORERUSERHOME
```

Now, create *Makefiles* for those module subdirectories:

```
make Makefiles
```

Finally, you compile, link and install these modules:

```
make install
```

At the end, all of the sample modules should be compiled, linked, and installed.

9.8.3. Setting Up a Personal Module Tree

If you decide to set up your own build system, you create subdirectories in which you can develop one or more modules. First copy the template *Imakefile* into your development directory and make it writable:

```
cp /usr/explorer/lib/Imakefile.modules Imakefile
chmod +w Imakefile
```

Then you can make a subdirectory (whose name may or may not match the module name you wish to use), and add that subdirectory to the list in *Imakefile* which starts with the line

```
MODULESUBDIRS = \
```

and is followed by a list of subdirectories containing modules. Any number of directory names can be listed, as long as they exist and are IRIS Explorer module development subdirectories.

You then build a new top–level *Makefile*, remake subdirectory *Makefiles*, and then proceed with development of the new module.

For example,

```
cd DIR
mkdir mycontour
vi Imakefile

 [add mycontour to the MODULESUBDIRS list]

cxmkmf
make Makefiles
cd mycontour
make all
```

There are several environment variables that affect the way modules are built and installed. See **"Configuring the Build Environment"** for more information on these variables.

Each time you add a directory to the MODULESUBDIRS list, you must recreate the top−level *Makefile* with *cxmkmf.* Do not be tempted to edit the *Makefile* directly.

Once you have done this, the following commands will work in each module subdirectory:

`make all`	compiles and links all modules
`make clean`	deletes object files
`make depend`	adds dependencies to your *Makefile*

The typical development cycle is to write your module user function, use the Module Builder to describe it, then use the Build option in the Module Builder to build it. Alternatively, you can describe the module first, then use the Module Builder to generate the function prototype. You can then fill in the code for the user function, and after that, build the module.

From this point on, you can rebuild the module at any time using the *make all* command. For more information, refer to **Appendix A, "Using Makefiles"**.

Chapter 10
Module Prototyping with Shape

This chapter describes how to use the Shape language and the *LatFunction* module to create prototypes of new modules very quickly without having to use the Module Builder. It describes the Shape language syntax in detail.

10.1. Overview

Each module in an IRIS Explorer map performs a well−defined, often complicated, function. On occasion, you will find that no module exists for the lattice−based operation you want to perform. You can place a *LatFunction* module in your map, then read in a program written in *LatFunction*'s *Shape* language, to perform this operation. The module's interpreter carries out the actions you specify.

The *LatFunction* module is an interpreter for a lattice manipulation language that makes it easy to operate on lattices. It lets you interactively change the way *LatFunction* acts on the data it receives, which provides a quick way to prototype a new module without going through the Module Builder or writing C or Fortran code. It also gives you programmatic control beyond the range you would have with dials and sliders, and it eliminates the need to compile, link, and install a module each time you modify it.

As a language interpreter designed to work with lattices and arrays, the *LatFunction* module runs programs written in Shape and fed to it from *LatFunction*'s *Program File* or *Program Text* input ports. Shape's syntax is similar to the C programming language. Its main advantages are that it allows you to work directly with arrays and to change programs dynamically.

LatFunction is structured like other modules except that it accepts only lattice and parameter data. Its ports are wired in the same way as other module ports, and *LatFunction* makes data on those ports available to the Shape language interpreter as internal variables. You can:

- Perform simple arithmetic on lattices, much as the P−Func Editor (described in Section 4.5 of the *IRIS Explorer User's Guide*) lets you perform arithmetic on parameters.
- Test an idea for a full−blown module by creating a prototype of it.
- Change your prototype interactively and view its computational results without building or loading a new module.
- Mimic the effects of several other modules in one to obtain a general idea of the results of the combination.

10.2. The LatFunction Module

LatFunction has two forms. The basic *LatFunction* module has a fixed set of ports, uses fixed variable names for its input and output data, and does not accept parameters. The second form of *LatFunction* is analogous to the DataScribe. You create your own module resources to describe input ports, output ports, and control panel, and you write a Shape program. The generic *LatFunction* executable then executes in the guise of your module, according to its Shape program. This is called a *LatFunction*−based module. The main differences between the two are the port names and control panels. Both interpret the Shape language identically.

10.2.1. The LatFunction Module

The *LatFunction* module (see **Figure 10−1**) has three input ports, two parameters, and a lattice that controls its program execution, plus data ports.

Program File — The name of a program file chosen with the file selector. The file selector is a text type−in widget. The filename overrides any program typed in on the *Program Text* parameter port. This port is required on all *LatFunction* modules.

Debug — Can be *On* or *Off* (the default). When it is *On*, the value of each program statement is printed after it is executed. This widget is optional on generic *LatFunction* modules.

Program Text — A Shape program in the form of a null–terminated byte lattice. This program is executed if no *Program File* is specified; otherwise it is ignored. This port is optional on generic *LatFunction* modules.

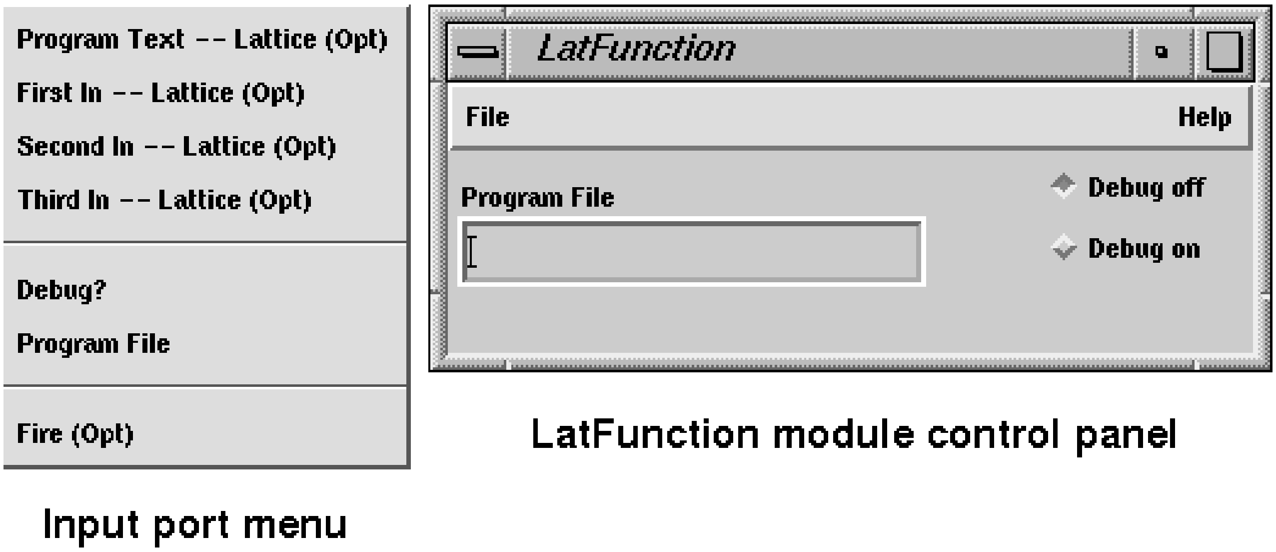


Figure 10–1 The *LatFunction* Module Control Panel

You can either enter programs in the *Program Text* input port or load them from a file by typing a filename into the *Program File* file selector.

LatFunction has fixed names for its data ports. On input, data ports are F*irst In, Second In*, and *Third In*. On output, they are *First Out, Second Out*, and *Third Out.*

10.2.2. The LatFunction–based Module

A *LatFunction*–based module can accept any number of input parameters and lattices and can output any number of lattices. It does not output parameters.

Input Ports

The *LatFunction*–based module uses the name of an input port, such as *Pressure*, to create an internal variable with the same name, `Pressure`, in the Shape language. If the input port contains a non–alphanumeric character (a space is considered a non–alphanumeric character), the *LatFunction*–based module turns that character into an underscore. For instance, an input port called *Temperature Array* is assigned the internal variable name `Temperature_Array`.

A *LatFunction*–based module creates internal array variables only for those input ports that have data. It does not create internal variable names for ports that do not contain data. This means that you will be able to test your Shape programs only after all ports that are required for the module execution have been wired up. The size and shape of the internal array depend on the port data type, which can be either *parameter* or *lattice*.

Parameters on input ports can be of two types:

- a *scalar* (a single value consisting of an integer or a double–precision real number) or
- a *character string.*

A scalar, such as the value of a slider widget, is sent to Shape as a scalar. A character string is sent to Shape as a 1D array of integers representing the character string in the ASCII collating sequence.

A *LatFunction*–based module sends a lattice to Shape in each of three forms: the data array, the coordinate array, and the combined lattice. The variable name indicates the form: the suffixes _lat and _coord signify the combined lattice and the coordinate array, respectively. The variable name itself signifies the data part of the lattice only. For instance, a *Pressure* port yields data `Pressure` coordinates `Pressure_coord`, and lattice `Pressure_lat`.

The Shape ports themselves have names derived from the IRIS Explorer ports. Input ports take the prefix `iport_` followed by the IRIS Explorer port name; output ports use the prefix `oport_`. For instance, the port names `iport_Pressure` and `oport_Pressure_Gradient` correspond to the example in **Figure 10–2**.

Output Ports

A *LatFunction*–based module assumes it must produce data for all of its output ports. Thus, for each output port name, the *LatFunction*–based module uses the internal variable of the same name to create an output lattice.

Name mapping differs for input and output ports. For input ports, only the data goes into the variable; for output ports, the variable contains data and may also contain coordinates.

A *LatFunction*–based module distinguishes the different output possibilities as follows:

- If a given variable is a single array, Shape assumes this array contains data and appends default, uniform coordinates to it. (See the *IRIS Explorer Reference Pages* for a description of *cxCoordDefaultNew*, which creates these coordinates.)
- If the variable is a full lattice (containing two arrays), Shape uses those two arrays for coordinates and data in the output lattice. For instance, in the case of the `Pressure` input port, the *LatFunction*–based module creates the internal variable Pressure to contain the data array, and `Pressure_coord` to hold the location coordinates.

Thus,`Pressure_lat` can be seen as a list of two arrays, one that contains data and one that contains location coordinates. If the output array `Pressure_Gradient` has two arrays, they are assumed to be coordinates and data (see **Figure 10–2**).

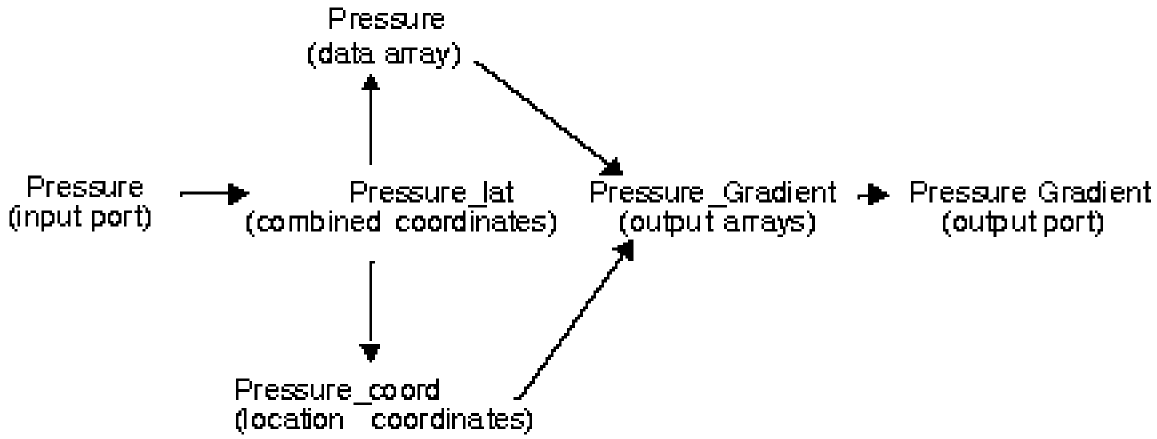


Figure 10–2 How Shape Assigns Variable Names

To get coordinates from input to output ports, you must build them into the Shape internal variable by using `make_lat(coordinates, data)`. The order of the arguments is significant: coordinates come first.

For example, suppose a reactor vessel has radially symmetrical pressure coordinates with pressure data at the nodes. To compute the gradient of the pressure using *LatFunction*, use the following procedure:

1. Create the module resources for a *LatFunction–based* module with an input lattice port named *Pressure* and an output port named *Pressure Gradient* (see **"Building a LatFunction–based Module"**).

   Create a gradient program in Shape, using `Pressure` and `Pressure_coord` to compute a gradient field called `Field`.

2. Use `make_lat` to compose the gradient lattice (the *define* operator := is described in **"Shape Language Particulars"**):

   ```
   Pressure_Gradient := make_lat(Pressure_coord, Field);
   ```

When the *LatFunction* script finishes, the output port contains the pressure gradient based on the same coordinate system as the input pressure. An output port can use `output_data_flush` to send multiple data sets downstream during one module firing.

10.2.3. Interfacing to IRIS Explorer Data Types

The *LatFunction* module takes lattices as its inputs and produces lattice as its outputs, but Shape operates only on arrays. Shape cannot operate directly on lattices because each lattice contains two arrays, one for the data and one for its location coordinates, thus requiring each lattice to be split into its component parts.

Lattices can have coordinates of three types: *uniform*, *curvilinear* and *perimeter*, all three of which are acceptable in Shape. For uniform and curvilinear lattice coordinates, the coordinate array passed through to Shape consists of the floating point coordinates of the lattice in whatever type was present on input. This array looks exactly like the values array of the comparable lattice coordinates structure.

For example, the program to scale an array of data by 3.0 while maintaining coordinate positions is:

```
First_Out := make_lat(First_In_coord, 3.0* First_In);
```

The program to scale a lattice's coordinate positions by 3.0 while maintaining data values is:

```
First_Out := make_lat(3.0* First_In_coord, First_In);
```

Note: This program works only for uniform and curvilinear coordinates. Perimeter coordinates are handled separately (see **"List Operators"**).

10.2.4. Building a LatFunction−based Module

You can give your *LatFunction*−based module any control panel you want, with any widgets and any port names. However, every *LatFunction* module must have a widget named "Program File". This widget can be either a File Browser or a Text Type−in widget but it must deliver a text string. Your module may also have a radio button widget named "Debug?" This widget generates debugging printouts when its value is non−zero.

To build a custom L*atFunction*−based module, follow these steps:

1. Use the Module Builder to construct a module with the desired input and output port names and control panel (see **Chapter 2**).
2. In the "Program File" type−in, save the name of the file that will contain the Shape program for this module.
3. Click on the *Yes* radio button for "Alternate Executable" on the Build Options panel and type in **`LatFunction`**. This tells IRIS Explorer to look for the executable under *LatFunction* instead of under the name of the module you have just built.
4. Save the module resources, and build and install the module.
5. Write the Shape program, saving it under the same filename as you gave the module in Step 2.

You can now load the module into an IRIS Explorer map.

10.3. Prototyping on the Fly

If you bring up an IRIS Explorer map with a *LatFunction*−based module in it, you can now change that module's execution simply by changing the contents of the program file. Pressing the **`Enter`** key with the cursor in the "Program File" type−in forces the *LatFunction* module to reread and interpret that file. This, in turn, forces the module to refire based on a new program.

Note: A refiring caused by the module receiving a different value from a control panel widget does not cause the program to be reinterpreted.

Think of a channel selector that takes a three−color RGB (red, green, blue) image as input. Based on your selection, it sends either red, green, or blue downstream. You can change the selected channel on a control panel but, by modifying the program, you can also change the representation of output data. It is clear that prototyping on the fly has definite advantages when compared to the more laborious process of creating a module from scratch.

For instance, the following channel selector has three input ports:

Multi Channels	Lattice data
Selection	Parameter channel
Program File	Parameter type−in with program file name

and produces one lattice as output:

Channel Out	Lattice data out

Its action is to select the red, green, or blue channel of the input data (`Multi_Channels`), depending on the value of `Selection`, and output a gray−scale image on the output port.

The Selection parameter can be any widget that produces an integer between 0 and 2, inclusive, including a dial, slider, option menu, radio box, or even a type−in. Values below 0 and above the number of channels of the input are clipped to the valid range. The program creates a single−channel output. The shape program *channelselect.shp* and resources file may be found in directory */usr/explorer/src/MWGcode/LatFunc/Shape.* Test the shape program by connecting the modules: *ReadImg* to *channelselect* to *DisplayImg.* Read in the image file */usr/explorer/data/image/flowers2.rgb* and the shape program *channelselect.shp* and select each of the color channels, 1 to 3, in turn.

```
// Select a specified data channel from the input lattice
// and copy it to the output lattice

// Define sz to be the shape array of the input lattice Multi_Channel
// This is the dimension vector giving the shape of the lattice
// For example [3, 2, 4] for a 3*2 lattice with 4 data values
sz := shape(Multi_Channels);

// Get the rank of the shape array
rank := shape(sz)[0];

// Maximum number of data channels
nChan := sz[rank-1];

// Maximum selectable channels
chan := max(0,min(Selection,nChan-1));

int outsz[rank] = sz;
outsz[rank-1]=1;
Channel_Out := allocate((byte)1, outsz, fill_zero);
Channel_Out = inside(Multi_Channels[chan]);
```

Note: Leading // (double slash) characters indicate a Shape comment line.

10.4. Shape Language Particulars

The instructions for working with input data are given to *LatFunction* in the form of a Shape program.

10.4.1. Resemblances to C

As an array−oriented language with a C−like syntax, Shape has many similarities to C. Shape includes all of C's math library functions (see **Table 10−2**), as well as some other common operations. Like C, Shape function names and variable names are case−sensitive, and Shape has zero−based indexing. Shape extends the power of C by operating on arrays rather than on single scalar values.

For instance, if lattice input ports *First In* and *Second In* are of the same size and shape, you can add them to produce a lattice output port, *First Out*, with

```
First_Out:= First_In + Second_In;
```

Note: Using the *define* operator **:=** is analogous to assigning a pointer to a structure or to an array in C. Here it makes *First_Out* a reference to the temporary array created by the addition of *First_In* and *Second_In*.

Note: This statement creates a lattice `First_Out` which adds the data portions of the lattices `First_In` and `Second_In`, and is given default coordinates.

To try this and the next example, wire the output of *GenLat* to ports "First In" and "Second In" of *LatFunction* and wire *LatFunction*'s "First Out" port to *IsosurfaceLat* and then to *Render*.

Multiplication is:

```
First_Out:= First_In * Second_In;
```

You can compile the array of maximums with the *max* function, a Shape function analogous to the Fortran function *max*:

```
First_Out:= max(First_In, Second_In);
```

You can produce a double precision output lattice with:

```
First_Out:= (double) First_In;
```

Most other C arithmetic and logical operations work analogously on Shape arrays. Try:

```
First_Out:= sin(First_In);
```

10.4.2. Differences from C

Shape differs from C in several important ways, including the use of the *define* operator (:=), and the way it manipulates lattices.

Lattice Manipulations

In Shape, lattices behave much like arrays in C or Fortran, and they also resemble arrays in Fortran 90. You can enquire the value of a Shape variable *varname* by inserting the following line in your Shape program:

```
write(varname);
```

The examples show the Shape syntax of the computed result with an arrow symbol (for instance, `==>[[1,2],[3,4],[5,6]]`). They also show Shape arrays printed in a more conventional array layout, such as:

The array `[[[1,2], [3,4]], [[5,6], [7,8]]]` could be displayed equivalently as

$$\begin{bmatrix} [1,2] & [3,4] \\ [5,6] & [7,8] \end{bmatrix}$$

For example, in this fragment, the vectors `u`, `v`, and `r` are declared and allocated, then individual values are assigned as in C. Finally, the sum of `u` and `v` is copied (with the = operator) to `r`.

```
int u[3];    int v[3];    int r[3];
u[0] = 1;    u[1] = 5;    u[2] = 10;
v[0] = 2;    v[1] = 3;    v[2] = -3;
r = u + v;
write(r);
==>[3, 8, 7]
```

A Shape array is the same as the data array of an IRIS Explorer lattice. Its dimensions are identical to those of the IRIS Explorer lattice, with *nDataVar* treated as the fastest varying dimension. *nDataVar* defines the number of data variables per node in a lattice. For a 2D lattice of three data variables, with dimensions 10 x 20, the shape of the Shape array is `[20, 10, 3]`.

All *n*D lattices in IRIS Explorer are (*n*+1)D Shape arrays by default, with the *nDataVar* size taken as the extent of the fastest varying, *n*+1, dimension.

The *rank* of a lattice is the number of dimensions it has. The *shape* of a lattice is the vector whose entries are the lengths in each dimension. For example, `[1, 2, 3]` has rank 1 and shape `[3]`, and `[[1, 2], [3, 4], [5, 6]]` has rank 2 and shape [3, 2]. Rank 1 arrays are often called vectors and rank 2 arrays are often called matrices. An array with rank 0 is a scalar.

Arithmetic with arrays is almost always done element−wise, as follows:

```
u := [1, 2, 3];
v := [4, 5, 6];
write(u * v);
==>[4, 10, 18]
```

Arrays are automatically promoted to higher rank as required. Promotion entails augmenting the rank of an array and copying array values to fill in the new array elements. Lattices promote on the right, in the most rapidly varying dimension. That is, arrays with the following shapes are compatible with respect to promotion: `[5, 4, 3]`, `[5, 4]`, and `[5]`. Data values are copied to consecutive array locations during promotion.

For instance, given `u` from the above example:

```
write(u + 2);
==>[3, 4, 5]

m := [[1, 2], [3, 4]];
write(m * [3, 4]);
==>[[3,6],[12,16]]
```

Note: As m is a 2D Shape array it is equivalent to a 1D IRIS Explorer lattice, and you may also view the result of the computation by wiring up *LatFunction* to *PrintLat*:

```
First_Out := m * [3, 4];
==>[[3,6],[12,16]]
```

$$\begin{bmatrix} 3 & 6 \\ 12 & 16 \end{bmatrix}$$

Note: The fragment above *is not* matrix−vector multiplication

```
First_Out := m + 1;
==>[[2,3],[4,5]]
```

$$\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

```
m := [[1, 2], [3, 4]];
m = [1, 2];
First_Out := m;
==>[[1,1],[2,2]]
```

Note: The first line defines m as a 2D Shape array, and assigns values to it. The second line assigns the Shape vector [1, 2] to m, updating the values of m.

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$

You can change the direction in which lattices promote: refer to the *Inside*, *Outside*, and *InsideOut* functions listed under "Miscellaneous" in **Table 10−3** as well as in **"Promotion of Arrays"** and **"Reduction of Arrays"**.

It is an error to operate on arrays of differing lengths in any dimension. For example, `[1, 2, 3] + [3, 4]` is an error.

The Define Operator (:=)

The *define* operator, denoted `var:= value`, binds a variable without requiring that the variable be previously defined. *Define* is not present in C, but it provides functionality similar to that of C pointers.

The `:=` operator, unlike =, copies no data. Instead, the value is simply given a new alias, or name. The *define* operator is important because:

- *Define* lets you name only a part of an object. Consider the following code fragments:

  ```
  L := [1, 2, 3, 4, 5];
  a := L[1];
  a = 5;
  write(L);
  ==>  [1, 5, 3, 4, 5]
  ```

  and

  ```
  int a;
  L := [1, 2, 3, 4, 5];
  a = L[1];
  a = 5;
  write(L);
  ==>  [1, 2, 3, 4, 5]
  ```

  In the first example, a is defined by reference to a part of L, and has no storage of its own. Allocating the value `5` to a also changes the value of L[1]. In the second example a must be declared before the value L[1] may be assigned to it. As a is now a separate variable with its own storage, the value of L[1] is not changed when a is changed.
- *Define*(`:=`) is more efficient than copy (=). Define attaches a name to a piece of data rather than moving data to the name. For a single integer, the difference does not matter, but it is significant if the data is a large, multidimensional array.

  For example, you could make the statement

  ```
  a:= First_In;
  ```

  The data from the input port may be a whole lattice, and simply labelling the lattice instead of moving each piece of the lattice to the label is less work.
- *Define* obviates the need for some variable declarations. It is not necessary to preallocate or type the defined variable in Shape, since the alias picks up the type and size of its source.
- *Define* lets you refer to a regularly−strided portion of a larger array (see **"Subshape"**).

Define is commonly used to change a small part of a lattice. For example, after:

```
int m[5, 5];        // 5 by 5 array (1D lattice with 5 data variables)
d := diagonal(m);   // point to diagonal of m
d += 1;             // increment the diagonal of m
First_Out := m;     // output lattice
==>  [[1 0 0 0 0],[0 1 0 0 0],[0 0 1 0 0],[0 0 0 1 0],[0 0 0 0 1]]
```

m is now a 5 x 5 identity matrix. In the first line, m is declared to be a 5 x 5 matrix and is initialized with zeros. In the second line, the vector d is defined as the diagonal of the matrix. Finally, the scalar 1 is promoted to a vector and added to each entry of d. Since d is an alias for part of m (the diagonal), the result is that 1 has been added to each entry of the main diagonal of m.

Promotion of Arrays

Arrays promote on assignment and during binary operations. Arrays are promoted by padding the shape on the right, that is, in the fastest varying dimension. The promotion iterates each element of the source until the newly padded dimensions have been filled with the value, then moves on to the next source value.

```
int b[2,3,4];               // declare a 3D array of zeros
a := [[1,2,3],[4,5,6]];     // define a 2D array
```

```
b = a;
First_Out = b;
==> [[[1,1,1,1],[2,2,2,2],[3,3,3,3]],
     [[4,4,4,4],[5,5,5,5],[6,6,6,6]]]
```

$$\begin{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix} \\ \begin{bmatrix} 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 \end{bmatrix} \end{bmatrix}$$

```
First_Out := b + a;
==> [[[2,2,2,2],[4,4,4,4],[6,6,6,6]],
     [[8,8,8,8],[10,10,10,10],[12,12,12,12]]]
```

$$\begin{bmatrix} \begin{bmatrix} 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 \\ 6 & 6 & 6 & 6 \end{bmatrix} \\ \begin{bmatrix} 8 & 8 & 8 & 8 \\ 10 & 10 & 10 & 10 \\ 12 & 12 & 12 & 12 \end{bmatrix} \end{bmatrix}$$

Reduction of Arrays

Arrays are reduced by compressing the shape on the right, that is, in the fastest varying dimension. The reduction selects the first element of the source array in all of the newly reduced dimensions on assignment. On updating assignment (for example, +=) the dimensions are compressed by the reduction operator.

Reduction assignment by addition adds the data of the extra dimensions together. For example:

```
int b[2, 3, 4];
int c[2, 3];
a := [[1, 2, 3], [4, 5, 6]];
b = a;
c += b;
First_Out := c;
==>  [[4,8,12],[16,20,24]]
```

$$\begin{bmatrix} 4 & 8 & 12 \\ 16 & 20 & 24 \end{bmatrix}$$

Assignment reduction looks like a sum over `i` of `inside(A = B[i])` on the correct number of dimensions. The following example shows a few updating operator reductions:

```
// Addition ....
int b[2, 3, 4];
a := [[1, 2, 3], [4, 5, 6]];
long s;
write(s);
==>0
b = a;
s += b;
write(s);
==>84

// Bitwise conjunction of an array ....
s = 1;
write(s);
==>1
s &= b;
write(s);
==>0

s = 1;
write(s);
==>1
s &= [1, 1, 1, 1];
write(s);
==>1

// Logical conjunction of an array ....
s = 1;
write(s);
==>1
s &= (b>0);
write(s);
==>1
```

10.4.3. Other Operators

The *update* operators, such as += and *=, behave somewhat differently in Shape than in C. For example, take the expression `x += y`.

If `x` has lower rank than `y`, then `y` is reduced into `x`, as follows:

```
int x;
y := [1, 2, 3];
x += y;
write(x);
==>6
```

Note: The fragment above assigns to `x` the sum of the entries of `y`.

```
int v[3] = 1;
m := [[1, 2], [3, 4], [5, 6]];
v *= m;
write(v);
==>[2, 12, 30]  // the products of each row of m
```

Reduction occurs along the same dimension as promotion, on the right of the array's dimension vector.

10.4.4. Data Types

The base data types in Shape differ from those in C. Shape lacks `char` and all the C−type modifiers, such as `extern, static, unsigned, signed, short,` and `long`, but it has the additional data types `byte`, `complex` and `double complex`.

All the base data types in Shape match the IRIS Explorer data types, except the two complex types. The complex numbers consist of a pair of floats, and the double complex numbers consist of a pair of doubles. Complex numbers must be read into IRIS Explorer as floats or doubles, converted in Shape, and read out again as floats or doubles. **Table 10−1** compares the data types in the two languages.

Language	Data Types
C	char, short, int, long, float, double
Shape	byte, short, int, long, float, double, complex, double complex

Table 10−1 Data Types in C and Shape

10.4.5. Function Declaration Syntax

Functions are declared differently in Shape than in C. For example, the C−style declaration

```
float f(float x, int y) {...; return z;}
```

is expressed in Shape as

```
function f(x, y) {...; return z;}.
```

To summarize:

- Functions and their arguments are not typed.
- They start with the keyword "function".
- The keyword return signals the value to be returned from the function. In the absence of a return statement, the last computed value in the function is returned.

Table 10−3 lists the built−in functions available in Shape.

10.4.6. Loop Syntax

Shape's loop syntax is fundamentally identical to that of C. The following examples illustrate correct usage.

For Loop

Shape's *for* loop construction is identical to that of C. The three expressions in parentheses following *for* give the initial setting, test logic, and loop counter changing, respectively. The code block for the loop can be one statement or a sequence of statements; if the block is a sequence of statements, it is enclosed in curly braces. For instance:

```
int i;
double a[5];
for (i=0; i<5; i+=1)
     a[i] = sin(i);            // single statement -- no curly braces
write(a);
==> [0, 0.841471, 0.909297, 0.14112, -0.756802]
```

Caution: Because the increment operator ++ is not yet implemented, use the construction i += 1 instead of i++ in loop iterations, as shown above. Using the ++ operator will produce a "Not Yet" error message.

While Loop

Shape's *while* loop construction is identical to that of C. The loop iterates while the conditional expression evaluates to true, or non−zero. For instance, the *for* loop of the previous example could be rewritten as:

```
int i;
int a[5];
i = 0
while (i < 5)
  {
     a[i] = sin(i);
     i += 1;
  }
write(a);
==> [0, 0.841471, 0.909297, 0.14112, -0.756802]
```

Continue and Break

Shape's *continue* and *break* statements are identical to those of C. The *continue* statement causes execution in a *for* or *while* loop to branch to the next iteration of the loop. The *break* statement causes execution in a loop to branch out of the loop to the next higher loop level or to exit from a singly−nested loop.

10.4.7. Issues of Scope

Shape has the same concept as C of automatic variables, which are freed when the scope of their declaration is exited. This is useful for memory allocation, since it relieves you of having to worry about freeing variables. However, it can be inconvenient at times, particularly when you want to use a conditional assignment to an array of a variable size. For instance, the following program fragment is in error:

```
if (n==1)
  {
    array := index_fill([3,1]);
  }
else if (n==2)
  {
```

```
    array := index_fill([6,6,2]);
  }
First_Out := array;
```

This Shape program fails to set First_Out because array is freed when its scope (an if or else clause) is exited. You can circumvent this problem by creating a function that returns the array, as follows:

```
function ret_array(n)
{
  if (n==1)
     return index_fill([3,1]);
  else if (n==2)
     return index_fill([6,6,2]);
}

First_Out := ret_array(n);
```

10.4.8. Additional Features

Shape offers some features that are not available in C. These include:

- subarray selection and assignment, including scatter/gather operations
- permutation of array dimensions
- input/output to IRIS Explorer lattices
- a "garbage collection" service incorporated into the language that automatically frees memory that has become inaccessible

See **Table 10–2** and **Table 10–3** for lists of Shape functions.

Copy

The *copy* function implements a copy on write (COW) functionality for its single argument. Its action lies between that of *define* (:=) and *assign* (=) in that it creates a pointer to its argument only if no writes are done to that pointer. However, when the first write to the returned pointer occurs, the *copy* function copies the data values of the source. This late binding of copying is efficient for routines that access input port data; it prevents Shape programs from writing over input data, but delays making a copy until a *write* is attempted,as shown:

```
a := First_In;        // get the input lattice
write(a);             // assume that a byte lattice has been attached
==> [[200],[171],[73],[107]]

b1 := a[0];           // b1 points to the array element a[0]
b2 := copy(a[0]);     // b2 points to the array element a[0]
                      // but will copy the original value of a[0]
                      // if a[0] is updated
write(b1);
==> [200]
write(b2);
==> [200]

b1 += (byte)3;        // update b1 (assumed to be byte)
write(b1);
==> [203]
write(a);             // this will also update a[0]
==> [[203],[171],[73],[107]]
write(b2);            // b2 will still be equal to the original a[0]
==> [200]
```

In order not to overwrite the input lattice, b2 rather than b1 should be used for updating any values related to the input lattice.

Scatter and Gather

Shape's *scatter* and *gather* functions extract data from one array and copy it to another array. The extracted data need not be arranged in a regular pattern, as is required of *subshape*, below. However, the values are copied rather than simply assigned a pointer, as they would be with *subshape*. Thus, *scatter* and *gather* are more expensive than *subshape* .

scatter takes three arguments: the *destination* (dest), *index*, and *source* arrays. *scatter*'s functionality implements the mathematical operation:

```
dest(index)= source
```

which copies each value from source into the dest position specified by index. Source and dest must have the same type.

The shapes of `index`, `source`, and `dest` must be such that the last dimension of `index` equals the rank of `dest`, and the remaining dimensions of index equal their respective dimensions in `source`. For instance, if source has shape [2,3] and dest has shape [4,1,6], a valid index would have shape [2,3,3] and might be `[[0,0,0], [0,0,2], [0,0,4]], [[2,0,0], [2,0,1], [2,0,2]]`.

$$\begin{bmatrix} [0,0,0] & [0,0,2] & [0,0,4] \\ [2,0,0] & [2,0,1] & [2,0,2] \end{bmatrix}$$

With a `source` of `[[1,2,3], [4,5,6]]`, the resulting `dest` would be `[[[1,x,2,x,3,x]], [[x,x,x,x,x,x]], [[4,5,6,x,x,x]], [[x,x,x,x,x,x]]]`.

$$\begin{bmatrix} [[1,x,2,x,3,x]] \\ [[x,x,x,x,x,x]] \\ [[4,5,6,x,x,x]] \\ [[x,x,x,x,x,x]] \end{bmatrix}$$

where x denotes the original value from `dest` before the *scatter* operation was performed.

gather is a function that returns a gathered array, with two arguments, *index* and *source*, interpreted according to the mathematical operation `source(index)`.

gather returns the array of selected values in the same shape as index. For example, with the same `source` array as before, and an `index` [[0,2], [1,1]] the following result is achieved:

```
index := [[0,2], [1,1]];
source := [[1,2,3], [4,5,6]];
```

```
a := gather(index, source);
write(a);
==> [3 5]
```

Shape

The *shape* function returns a dimension vector giving the shape of its input argument. For instance, a 128 x 256 lattice with three channels of data `(dims = (128,256), nDataVar = 3)` would yield an array of shape `[256,128,3]`.

The *shape* function is commonly used to discover the rank of an array. An array's rank is its number of dimensions − essentially, the shape of the shape. The correct syntax is:

```
int rank = shape(shape(val))[0];
```

Writing only

```
int rank = shape(shape(val));
```

would fail, because the returned shape would be a vector of length 1 rather than a scalar rank. The zero−ordered slice extracts the scalar from its vector position.

Allocate

The *allocate* function allocates an array of the given shape and type (such as long or float, as given in the example below) and fills it with values according to the *fill_type*. The basic syntax is:

```
allocate(base_type_example, shape, fill_type)
```

The fill_type must be one of:

`fill_none`	leave uninitialized
`fill_zero`	fill with zeros
`fill_index`	fill with uniform coordinates (with integer indices)
`fill_NaN`	fill with IEEE NaN symbols

```
a := allocate(1, [3, 1], fill_index);
write(a);
==> [[0],[1],[2]]
```

$$\begin{bmatrix}0\\1\\2\end{bmatrix}$$

```
a := allocate(1, [3, 3, 2], fill_index);
write(a);
==> [[[0,0],[0,1],[0,2]], [[1,0],[1,1],[1,2]], [[2,0],[2,1],[2,2]]]
```

$$\begin{bmatrix} [0,0] & [0,1] & [0,2] \\ [1,0] & [1,1] & [1,2] \\ [2,0] & [2,1] & [2,2] \end{bmatrix}$$

The type of the first argument is used to fix the type of the allocated array. You may use a known array or value, use a cast value, such as `(byte)1`, or use an explicitly typed number, such as `1L` for long or `1.0L` for double.

Slice

Slice extracts a multi−dimensional slice from an array. It has two syntactical forms. The first one is x[y], where y is a scalar that selects the `y`th subarray of `x`, exactly as in C. When you use this form, the sliced array has rank one less than the source. For example, take a 2D array:

```
a := [[1,2,3], [2,4,6], [4,8,12], [8,16,24]];
```

Then `a[0] = [1, 2, 3]`, a three−length vector derived from A in the ordinary C manner.

The second form of slice is `a[x:y]`, where `x` and `y` are scalars. In this form an array section, or subarray, of a is extracted. The subarray has the same rank as a, but contains only slices `x` through `y`. For example, given `a`, the statement `a[0:1]` extracts two vectors from the original array, and the shape of the array is [2,3]. The rank is `2`.

In C and Shape, you can write `a[0][0]` to get the first element of a. In Shape, however, `a[0:2][0:1]` is different from `a[0:2, 0:1]`. The respective shapes are [2,3] and [3,1], because `a[0:2][0:1]` is the same as `a[0:1]`.

```
a := [[1,2,3],[2,4,6],[4,8,12],[8,16,24]];
write(a);
==> [[1,2,3],[2,4,6],],[4 8 12],[8 16 24]]

write(a[0]);
==> [1 2 3]

write(a[0:1]);
==> [[1 2 3],[2 4 6]]

write(a[0:2][0:1]);
==> [[1 2 3],[2 4 6]]

write(a[0:2,0:1]);
==> [[1,2,3]]
```

Subshape

The *subshape* function takes three or four arguments and returns a reference to a subarray (or subsection) of its input array. It does not copy any data. The syntax is:

```
subshape (start, end, source)
```

or

```
subshape (start, end, stride, source)
```

where *start, end*, and *stride* are vectors of size `shape(`*source*`)` that signal the indices of the beginning and end of the array subsection and the strides between successive elements in each dimension. The default stride is 1. For example, with a 10 x 10 `source`, you can extract the central 2 x 2 square with `subshape([4,4],[5,5], source)` and extract every other row with `subshape([0,0],[10,11],[1,2], source).`

An ending index greater than the size of source is legal only if the actual indices used in the computation do not exceed the array bounds. In this example, rows 0, 2, 4, 6, 8, 10 would be extracted, but row 11 would not.

Subshape does not copy data; it returns a reference to the selected subarray.

Stack

The stacking operator concatenates arrays of compatible shape to yield a new array of rank one higher. The stacking syntax is:

```
[expr, expr, ...]
```

The shape of the stacked array is equal to the shape of the stacked components, with an added component in the slowest varying dimension. That component is the number of stacked arrays. For instance, let:

```
a:= [1,2,3];
b:= [4,5,6];
c:= [a,b];
==> [[1,2,3],[4,5,6]]
```

In this case, both a and b have shape `[3]`, while c has shape `[2,3]`. You can change the stacking dimension with *inside.*

Send_indices

The *send_indices* function takes an index permutation vector and a lattice. The permutation controls where each coordinate ends up. As a special case, if permutation indices agree, then a diagonal is taken in those coordinates. For example:

```
a := [[1,2],[3,4]];
b := [1,0];
write(send_indices(b,a));
==> [[1,3],[2,4]]
```

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

The 2 x 2 matrix a is transposed because indices 0 and 1 were swapped by the index vector b.

```
a := [[1,2],[3,4]];
b := [1,1];
write(send_indices(b,a));
==> [1,4]
```

The new matrix is a diagonal of the 2 x 2 matrix a because the indices were equal in the index vector b.

Min and Max

Min and *max* take one or more arguments. With two or more, *min* does an element−wise minimum over all its arguments. With one argument, it performs a reduction by one coordinate. For example, the first statement takes the pointwise minimum of three vectors, while the second finds the minimum element of one vector:

```
min([1, 2, 3], [4, -2, 6], [7, 8, 9]);
==> [1, -2, 3]

min([1, 2, -2, 3]);
==> -2
```

DotProduct (+.*)

The *dotproduct* operator +.* may be used to form the dot product between vectors and/or arrays. This is matrix or tensor multiplication.

```
[1, 2, 3] +.* [4, 5, 0];
==> 14

[[1, 2], [3, 4]] +.* [5, 6];
==> [17, 39]

[5, 6] +.* [[1, 2], [3, 4]];
==> [23, 34]

[[1, 2], [3, 4]] +.*[[5, 9], [10, 13]];
==> [[25 35] [55 79]]
```

$$\begin{bmatrix} 25 & 35 \\ 55 & 79 \end{bmatrix}$$

The form `A+.*B` is identical to the function *dot_product(A,B). dot_product* has an optional third argument, which directs some number of leading coordinates *not* to be involved in the product, but, rather, to indicate that several matrix multiplications of lower rank are to be performed. For instance, if A and B are `[3, 3, 3]` dimensional arrays, then *dot_product(A, B)* has shape `[3, 3, 3, 3]`, while `dot_product(A, B, 1)` has shape `[3, 3, 3]`. The former is a single, 3D tensor product, while the latter consists of three 2D matrix products.

BitProduct (|.&)

Bitproduct resembles *dotproduct*, except that it substitutes bit−wise "or" and "and" functions for addition and multiplication. The *bitproduct* operator is |`.&`.

Conditional Evaluation (?:)

The conditional evaluation operator ?: is similar to that of C. When working with scalar values, its operation is identical to that of C, so that

```
x ? a : b
```

evaluates to a if `x` is true (or non−zero) and to b if `x` is false (or zero). The operator goes beyond the C interpretation in that it can accept arrays as well. In this case, the arrays `x`, `a`, and `b` must be compatibly shaped. For instance, all three could have the same shape, or either a or b could be a scalar.

The array operation of `x ? a : b` creates an array the size of x but with values chosen from `a` or `b`, depending on the values of `x`. Scalars a and b are promoted to constant−filled arrays the shape of `x`, but no additional storage is allocated.

10.4.9. Changing Promotion and Reduction Order

The *inside*, *outside* and *insideout* forms are used.

Inside

The argument to the inside form is evaluated with different indexing rules than normal. The four array operations— indexing, concatenation (stacking), promotion, and reduction— start from the opposite end of the shape vector. For instance, `A[0]` selects the first slice of `A` in the slowest varying dimension, while `inside(A[0])` selects the first slice of `A` in the fastest varying dimension. The former is a *blocked* selection, while the latter is an *interleaved* selection. For example:

```
v := [1, 2];
w := [3, 4];
write([v,w]);
==> [[1,2],[3,4]]
```

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$$

```
write(inside([v, w]));
==> [[1,3],[2,4]]
```

$$\begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix}$$

```
m := [[1, 2], [3, 4]];
write(inside(m[0]));
==> [1, 3]

v := [0, 0];
write(inside(v += m));
==> [4, 6]
```

```
m := [[1, 2], [3, 4]];
write(inside(m + [5, 6]));
==> [[6,8],[8,10]]
```

$$\begin{bmatrix} 6 & 8 \\ 8 & 10 \end{bmatrix}$$

Outside

The *outside* argument evaluates its argument with the default indexing rules. For example:

```
v := [1, 2];
w := [3, 4];
write(outside([v, w]));
==> [[1,2],[3,4]]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
m := [[1, 2], [3, 4]];
write(outside(m[0]));
==> [1, 2]

v := [0, 0];
write(outside(v += m));
==> [3, 7]

m := [[1, 2], [3, 4]];
write(outside(m + [5, 6]));
==> [[6 7],[9 10]]
```

$$\begin{bmatrix} 6 & 7 \\ 9 & 10 \end{bmatrix}$$

InsideOut

The *insideout* argument evaluates its argument with the opposite of the current indexing rules. Thus, `insideout(inside(expr))` and `outside(expr)` are equivalent.

10.5. Data Output from Shape

Shape provides some input and output capabilities in addition to creating array variables from input ports automatically and sending data out on output ports. For instance, it lets you flush multiple arrays to output lattices during one computation, and you can also write files.

Before writing a file, you must open it. The file opening commands return a port identifier as a Shape variable. For instance,

```
globe := open_output_file ("earthData.rgb");
```

would open an output file.

Note: *LatFunction* automatically creates port identifiers, with names such as iport_<*portname*> and oport_<*portname*>, for all the input and output IRIS Explorer ports on *LatFunction* or a *LatFunction*–based module. You can treat *LatFunction* file port identifiers interchangeably with IRIS Explorer port identifiers.

You can write an array *data* to a port identified *by portID* using the *write*command:

```
write(data, portID);
```

If you want this data to enter the IRIS Explorer map immediately, rather than waiting until module firing finishes, you can use the*flush_output_port* command, as in

```
flush_output_port(portID);
```

In the absence of a flush, the lattice propagates to the map after the *LatFunction* module finishes firing.

You can also write an array to standard output, which can be useful for debugging, with

```
write(data);
```

Also useful for debugging is the *dump* command, which prints the shape, type, contents, and other information for an array.

You can write characters on ports with the *write_char* functions, which transmits a single character.

```
file:=open_output_file("test.out");
byte a=(byte)65;
write_char(a,file);
==> A (in file "test.out")
```

10.5.1. Scalar_lattice_in and Scalar_lattice_out

Because IRIS Explorer lattices use their *nDataVar* length as one of their array dimensions when they are converted to and from Shape arrays, it can be inconvenient to work with scalar lattices (nDataVar = 1) in LatFunction. For instance, a 2D scalar lattice of size 10 x 20 in IRIS Explorer would have shape [20,10,1] and be 3D in Shape. The *scalar_lattice_in* function removes the last dimension of its input argument. In fact, *scalar_lattice_in* removes the last dimension even if its input is not a scalar lattice; in this case, it returns the first data variable at each node.

The function *scalar_lattice_out* adds a trailing unit dimension to an array. It can be used to convert a 2D Shape array into a 2D IRIS Explorer data array (with nDataVar = 1).

10.5.2. List Operators

Shape can work with lists of arrays, as well as handle arrays. A list is an ordered collection of items, each of which can be an array or another list. For instance, a lattice is represented in Shape as a list of two arrays:

```
(coord, data)
```

Note: Lists are delimited by parentheses, not by square brackets.

The perimeter coordinates of a lattice are stored as individual vectors in a list. For a three–dimensional lattice, this would be:

```
(Xcoord, Ycoord, Zcoord)
```

Thus the entire lattice would be:

```
((Xcoord, Ycoord, Zcoord), data)
```

Most Shape commands do not work on lists. Instead, you must either use one of the special list processing commands or extract an array from the list and work with it directly. The list processing commands are:

`list`	creates a list
`list_first`	returns the first list element
`list_rest`	returns the tail of the list, that is, all but what is returned by `list_first`.
`map`	operates on list elements
`apply`	operates on list elements
`pair_p`	returns true if the list is non–empty (that is, a list of at least one element)
`null_p`	returns true if the list is a null list

The empty list has its own predefined symbol, `nil`.

List

The *list* function is used to concatenate two or more arrays in Shape. This can be useful when passing several arguments to a user function, or when assembling arguments for use in a map or apply call. The syntax for this function is:

```
list(array [, array] ...)
```

list uses parentheses `()`, not square brackets `[]`. It can build arbitrarily complex, heterogenous structures. `list(1,2,3)` is not equal to `[1,2,3]`. For example, to create and then concatenate two scalars and a two–vector:

```
x := 1;
write(x);
==> 1

y := 2;
write(y);
==> 2

z := [3,4];
write(z);
==> [3,4]

list(x,y,z);
write(list(x,y,z));
==>(1 2 [3 4])
```

List_first and List_rest

The two functions *list_first* and *list_rest* decompose lists, extracting the first and all but the first list elements, The two functions *list_first* and *list_rest* decompose lists, extracting the first and all but the first list elements, respectively. For example:

```
a := (1, 2, 3);
write(list_first(a));
==> 1
write(list_rest(a));
==> (2,3)
```

Handling Perimeter Lattice Coordinates

For perimeter coordinates, the Shape array structure is a list of vectors, one for each coordinate dimension of the lattice. For instance, 3D data with perimeter coordinates has three vectors in a list: the first vector represents the `X` dimension, and the second and third represent the `Y` and `Z` dimensions, respectively.

You can access the list of vectors with the list processing commands, `list_first and list_rest`, which return the first element of a list and the remainder of the list, respectively. For instance, to access the `X`, `Y`, and `Z` vectors of a perimeter lattice, you could use:

```
X_vector := list_first(Perim_Data_coord);
Y_vector := list_first(list_rest(Perim_Data_coord));
Z_vector := list_first(list_rest(list_rest(Perim_Data_coord)));
```

On output, such a list is interpreted as perimeter coordinates for a lattice. You can construct a list using the list function and the nil token. You will need either to write *LatFunction* routines to handle all coordinate types or else restrict inputs to a known type, using the Module Builder.

Map and Apply

The functions *map* and *apply* are used to run a Shape function on a set of inputs. The syntax for these functions is:

```
map(func, list...)
apply(func, list...)
```

The effects of the calls are different in that *map* runs the named function on each item in the list of inputs, whereas *apply* runs the named function on the concatenated list as a single input:

```
map(sin, list(0, 1, 2, 3));
==> (0 0.841471 0.909297 0.14112)
```

This is equivalent to concatenating several calls to *sin.*

```
sin(0);
==> 0

sin(1);
==> 0.841471

sin(2);
==> 0.909297

sin(3);
==> 0.14112
```

When the list has several arrays in it, the *sin* function works on each input:

```
map(sin, list([1, 2, 3], [4, 5], [1.1]));
==> ([0.841471, 0.909297, 0.14112], [-0.756802, -0.958924],[0.8912007])
```

This is equivalent to concatenating several calls to *sin.*

```
sin([1, 2, 3]);
==> [0.841471 0.909297 0.14112]

sin([4, 5]);
==> [-0.756802 -0.958924]

sin([1.1]);
==> [0.891207]
```

The *apply* function treats the list as the list of inputs to a single call of the named function.

```
apply(min, list(3, 4));
==> 3
```

This is equivalent to the call:

```
min(3, 4);
==> 3
```

but is different from the concatenation of calls:

```
min(3);
==> 3

min(4);
==> 4
```

Pair_p and Null_p

The functions *pair_p* and *null_p* are Boolean predicate functions, each of which indicates the list status of its argument. Thus, *pair_p* returns *true* (non−zero) if its argument is a list of at least one element (non−empty list), while null_p returns true if its argument is the empty list of no elements.

10.6. The Module Prelude and Postlude

The module prelude is a set of predefined operations that runs when the *LatFunction* module begins firing. They include defining certain mathematical constants, such as *pi*; predefined functions for operations that occur every time the module fires; and defining useful constants and functional associations.

The postlude is a set of predefined functions that runs when the *LatFunction* module has finished firing. These functions assemble data from known array names and send it to the output ports. They convert Shape data types back into IRIS Explorer data types.

10.6.1. Prelude Set−up Functions

These prelude functions run only once, when the *LatFunction* module is first launched. They are:

- Mathematical constants:

```
pi       := 3.14159265358979323844;
e        := 2.71828182845904523536;
i        := make_complex(0, 1);
NaN      := sqrt(-1);
```

- Allocation fill mode indicators:

```
fill_NaN
fill_none
```

```
fill_none
fill_index
fill_zero
```

- Extract the data portion of a lattice:

```
function lat_data(lat)
```

- Extract the coordinate portion of a lattice:

```
function lat_coord(lat)
```

- Combine two arrays into coordinate and data portions and return a lattice:

```
function make_lat(coord, data)
```

Predefined Shape functions include.

- Fill an array with 0, NaN, or indices:

```
function index_fill(s) {allocate(1L, s, fill_index);}
```

- Return the transpose or diagonal of a 2D array:

```
function transpose(m) {send_indices([1, 0], m);}
function diagonal(m) {send_indices([0, 0], m);}
```

- Convert a scalar *n*D lattice from or to a Shape *n*D array:

```
function scalar_lattice_out(x) {inside([x]);}
function scalar_lattice_in(x) {inside(x[0]);}
```

All *n*D lattices in IRIS Explorer are (*n*+1)D Shape arrays by default, with the *nDataVar* size taken as the extent of the fastest varying, *n*+1, dimension.

10.6.2. Prelude Execution Functions

These prelude functions run every time the module fires, to organize IRIS Explorer lattice data into Shape arrays. The example shows prelude functions for the *LatFunction* ports "First In" and "First Out". Similar operations occur for the other two ports and for user–defined lattice ports in a *LatFunction*–based module.

- Define names for the input and output ports, as well as for their input lattices, their data parts, and their coordinates:

```
iport_First_In := open_input_lattice("First In");
oport_First_Out := open_output_lattice("First Out");
First_In_lat := read(iport_First_In);
First_In := lat_data(First_In_lat);
First_In_coord := lat_coord(First_In_lat);
```

10.6.3. Postlude Functions

The postlude functions run as the module completes firing. They convert Shape arrays back into IRIS Explorer lattice data for each output lattice port.

- Send the output data onto a known port, for instance:

```
write(First_Out, oport_First_Out);
```

The implication of the prelude section is that `First_In`, `Second_In`, and `Third_In` are Shape variables that contain the input port data for the module. On output, variables `First_Out`, `Second_Out`, and `Third_Out` automatically have their data flushed to output ports.

10.7. Shape Function Tables

Table 10–2 summarizes the Shape functions that resemble C language functions.

Operation	Function Name/Syntax	Notes
Arithmetic		
Addition	*array + array*	
Subtraction	*array – array*	
Division	*array / array*	
Tensor		
Boolean Matrix Multiplication	*array |.& array*	See *Matrix Multiplication*
Matrix Multiplication	*array +.* array*	Performs a tensor product
Casting		
To Byte	(`byte`) *array*	For example, `((byte)1.1)`
To Short	(`short`) *array*	
To Long	(`long`) *array*	
To Single Complex	(`complex`) *array*	
To Double Complex	`(double complex)` *array*	
Integer		
Modulo	*array % array*	
Bitwise And	*array & array*	
Bitwise Or	*array | array*	
Bitwise Exclusive Or	*array ^ array*	
Left Shift	*array << array*	
Right Shift	*array >> array*	
Logical And	*scalar && scalar*	Scalar only
Logical Or	*scalar || scalar*	Scalar only
Logical Not	*! array*	
Preincrement	*++ array*	Not yet implemented
Postincrement	*array ++*	Not yet implemented
Predecrement	*–– array*	Not yet implemented
Postdecrement	*array ––*	Not yet implemented
Relational		
Equal To	*array == array*	
Not Equal To	*array != array*	
Greater Than	*array > array*	
Less Than	*array < array*	
Greater Than or Equal	*array >= array*	
Less Than or Equal	*array <= array*	
Complex		
Real Part	`re`(*array*)	
Imaginary Part	`im`(*array*)	
Make Complex	`make_complex(` *array, array*)	`re(make_complex (a, b))` ==> a `im(make_complex (a, b))`

		==> b
Language Statements		
Conditional statement	`if(` *expr*`)` *Block* `else`	`else` clause is optional
Block	*Block statement;* or `{` *statement; statement;* ... `}`	
Function Declaration	`function(`*array, array, ...*`)` *Block*	
For Loop	`for(`*expr; expr; expr*`)` *Block*	
While Loop	`while(`*expr*`)` *Block*	
Define	*array := expr*	
Assign	*array = expr*	
Conditional array value	*expr? expr: expr*	Evaluates its arguments completely, unlike the C function with same syntax
Return	`return` *expr;* `return;`	
Continue	`continue;`	
Break	`break;`	
Comment	`//` *Comment text*	Precedes a comment (terminated by end−of−line) as in C++ (does not use `/*` as in C)
List	(*array*) or (*list, list, ...*)	C ordered list of arrays and/or lists (s−expression)
Math Library		
Square Root	`sqrt`	(*array*)
Sine	`sin`	(*array*)
Cosine	`cos`	(*array*)
Tangent	`tan`	(*array*) no complex
Arc Sine	`asin`	(*array*) no complex
Arc Cosine	`acos`	(*array*) no complex
Arc Tangent	`atan`	(*array*) no complex
Hyperbolic Sine	`sinh`	(*array*)
Hyperbolic Cosine	`cosh`	(*array*)
Hyperbolic Tangent	`tanh`	(*array*) no complex
Exponentiation	`exp`	(*array*)
Logarithm	`log`	(*array*)
Arc Tangent 2	`atan2`	(*array, array*) no complex
Power	`pow`	(*array*) no complex, also infix(`**`)
Modulo (floating)	`fmod`	(*array, array*) no complex
Remainder	`drem`	(*array, array*) no complex
Conjugate	`conj`	(*array*)
Ceiling	`ceil`	(*array*) no complex
Absolute value	`abs`	(*array*)
Truncation	`floor`	(*array*) no complex
Rounding	`nint`	(*array*) no complex: `nint(a)`

		returns nearest integer rounding
Sign	`sign`	(*array*) no complex, −1, 0, or 1
Minimum	`min`	(*array, array, ...*) takes 1 or more arguments
Maximum	`max`	(*array, array, ...*) see Min

Table 10−2 C−like Functions in Shape

Table 10−3 lists the functions specific to Shape.

Operation	Function Name/Syntax	Notes
Lattice		
Copy	`copy`(*array*)	
Scatter	`scatter`(*dest array, index array, source array*)	
Gather	`gather`(*index array, source array*)	
ShapeOf	`shape`(*array*)	
Allocate	`allocate`(*base type array, shape array, fill type token*)	
Slice	*array*[*scalar*] *array*[*scalar:scalar*]	
SubShape	`subshape`(*start array, end array, source array*) `subshape` (*start array, end array, stride array, source array*)	
Stack	[*expr, expr, ...*]	
SendIndices	`send_indices`(*permutation array, source array*)	
List Operators		
List	`list_first`(*list*) `list_rest`(*list*)	First element of a list List consisting of all but the first element of a list
Map	`map`(*function, array, ...*)	
Apply	`apply`(*function, array, ...*)	
List	`list`(*array, array, ...*)	Creates a list
PairP	`pair_p`(*list*)	A non−empty list?
NullP	`null_p`(*list*)	The nil array?
Miscellaneous		
Inside	`inside`(*expr*)	
Outside	`outside`(*expr*)	
InsideOut	`insideout`(*expr*)	
Port IO		
OpenOutputFile	*portname* `:=` `open_output_file`(*filename*)	
FlushOutputPort	`flush_output_port`(*portname*)	Calls cxOutputDataFlush
Write	`write`(*expr, portname*)	Sends ASCII to port
Write	`write`(*expr*)	Sends ASCII to standard output
Read	`read`(*portname*)	Read from portname

Dump	`dump`(*array*)	Prints array debugging information
WriteChar	`write_char`(*portname*)	Writes one binary byte
Module Prelude		
Constants	pi	3.14
	e	2.71
	i	make_complex(0,1)
	NaN	IEEE constant
Values from	`iport_`*<portname>*	iport_First_In
Predefined	`oport_`*<portname>*	oport_First_In
Operations	*<portname>*`_lat`	First_In_lat full lattice
	<portname>`_coord`	First_In_coord coord part
	<portname>	First_In data part
	`nil`	List terminator value
Predefined Functions	`lat_data`(*lattice*)	Gets data part of lattice
	`lat_coord`(*lattice*)	Gets coord part of lattice
	`make_lat`(*lattice*)	Data + coord = lattice
	`index_fill`(*shape*)	Allocates a long array using `fill_index`
	`transpose`(*expr*)	Works on a matrix
	`diagonal`(*expr*)	Works on a matrix
	`scalar_lattice_in`(*expr*)	Removes an inner coordinate of length 1
	`scalar_lattice_out`(*expr*)	Adds an inner coordinate of length 1
	`conform`(*expr, expr*)	Takes a lattice and a shape and returns a lattice of the passed shape, but whose corner has the values of the passed lattice. Pads with zero the first output to Shape.

Table 10–3 Functions Specific to Shape

10.8. Sample Programs for LatFunction

Here are three sample programs for the *LatFunction* module.

10.8.1. To Create a Test Volume

You can send this program into *VolumeToGeom* and *IsosurfaceLat*, which both feed *Render*, to get an interesting data set; otherwise, look at the map *testVol*. The Shape program may be found in file */usr/explorer/src/MWGcode/LatFunc/Shape/testVol.shp*

```
// EXAMPLE chebyVol surface

   function chebychev(x) {
     xx := x * x;
     xx * xx - 2 * xx + 1;
   }
```

```
   n := 20;

// Allocate vol to be a cube, then fill with 0..n-1 uniform coordinate
// indices in 3D.

   vol := (short)index_fill([n, n, n, 3]);
   float res[n, n, n] = 0;

// Scale vol to lie in the range -2..2 and apply the chebychev() function
// pointwise throughout vol. Perform a sum-reduction over the XYZ vector
// to compute the result, which is returned as a scalar lattice.

   res += chebychev(vol * (4.0/n) - 2);
   First_Out := scalar_lattice_out(res);
```

Figure 10–3 shows a set of contours displaced from a volume in the Render module. The program script above produces this volume.

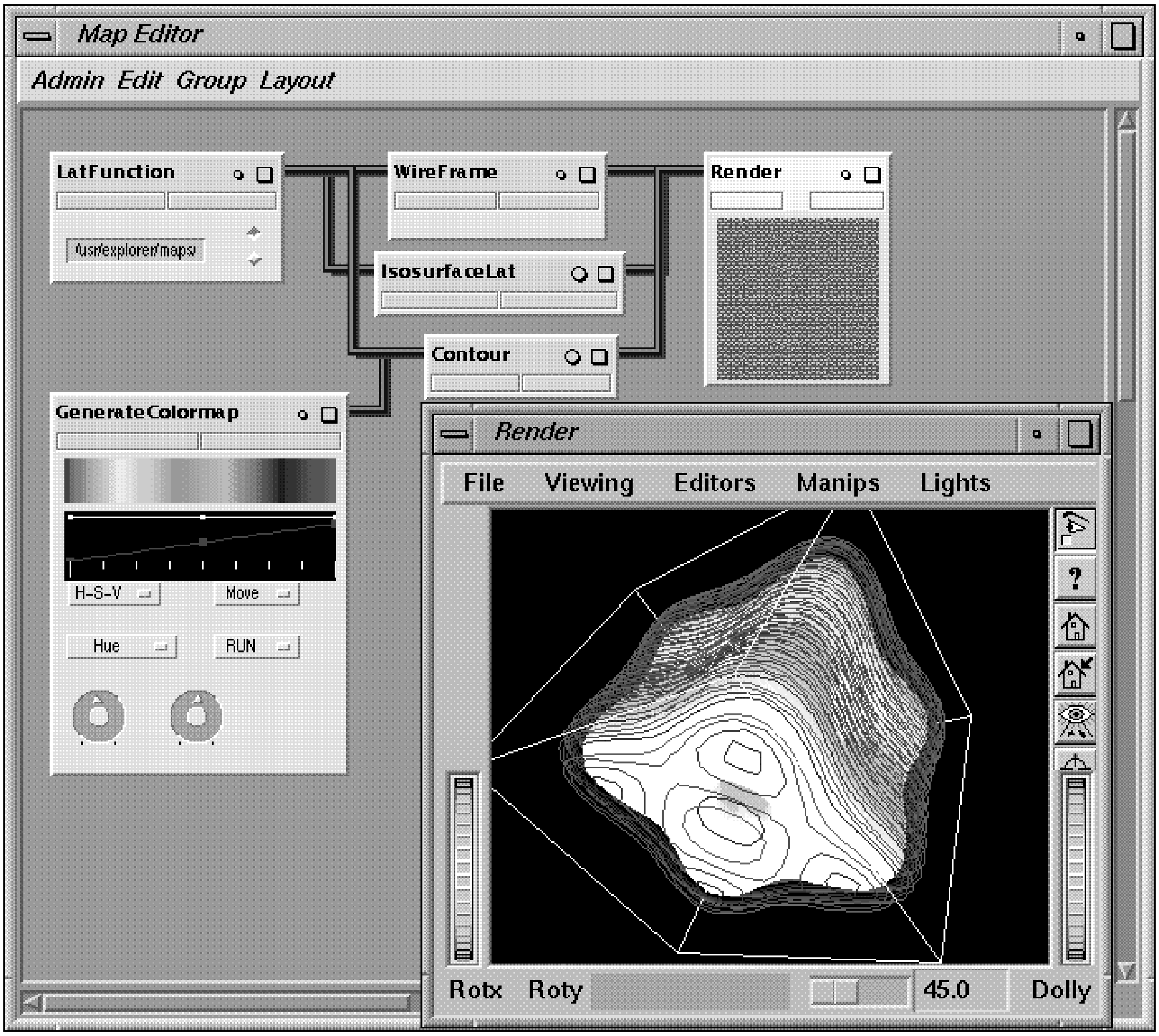


Figure 10–3 *LatFunction* in the *testVol* Map

10.8.2. To Carry Out a Color Transform

This script allows direct manipulation of the coloring of an RGB image. It may be found in */usr/explorer/src/MWGcode/LatFunc/Shape/ColorXForm.shp* Use the map *ColorXForm.map* in the same directory to view the image.

```
// EXAMPLE color transform (gray-scale)
// Create a gray-scale image by taking a linear combination of the RGB
// values of the input image as a gray-level intensity.
//
// Port First_In: 2D Image to transform
// Port Second_In: 4x4 matrix to provide colormap transform
//
// Port First_Out: Transformed 2D byte Image
//

// Extract input transform matrix
   mat := scalar_lattice_in(Second_In);

// Take linear combination of input matrix channels
// using tensor product.
   v := (double) [mat[0][0], mat[0][1], mat[0][2]];
   gray := First_In +.* v;

// Scales the range to 0..255
   mx := max(max(gray));
   mn := min(min(gray));
   if (mx == mn)
        mn -= 1;
   gray = (gray - mn) * (255 / (mx - mn));

// Assign to known output name
   First_Out := (byte) inside([gray, gray, gray]);
```

Figure 10–4 shows the results of the color manipulation in the *color–xform* map. The two images shown in *DisplayImg* and *DisplayImg<2>* can be compared.

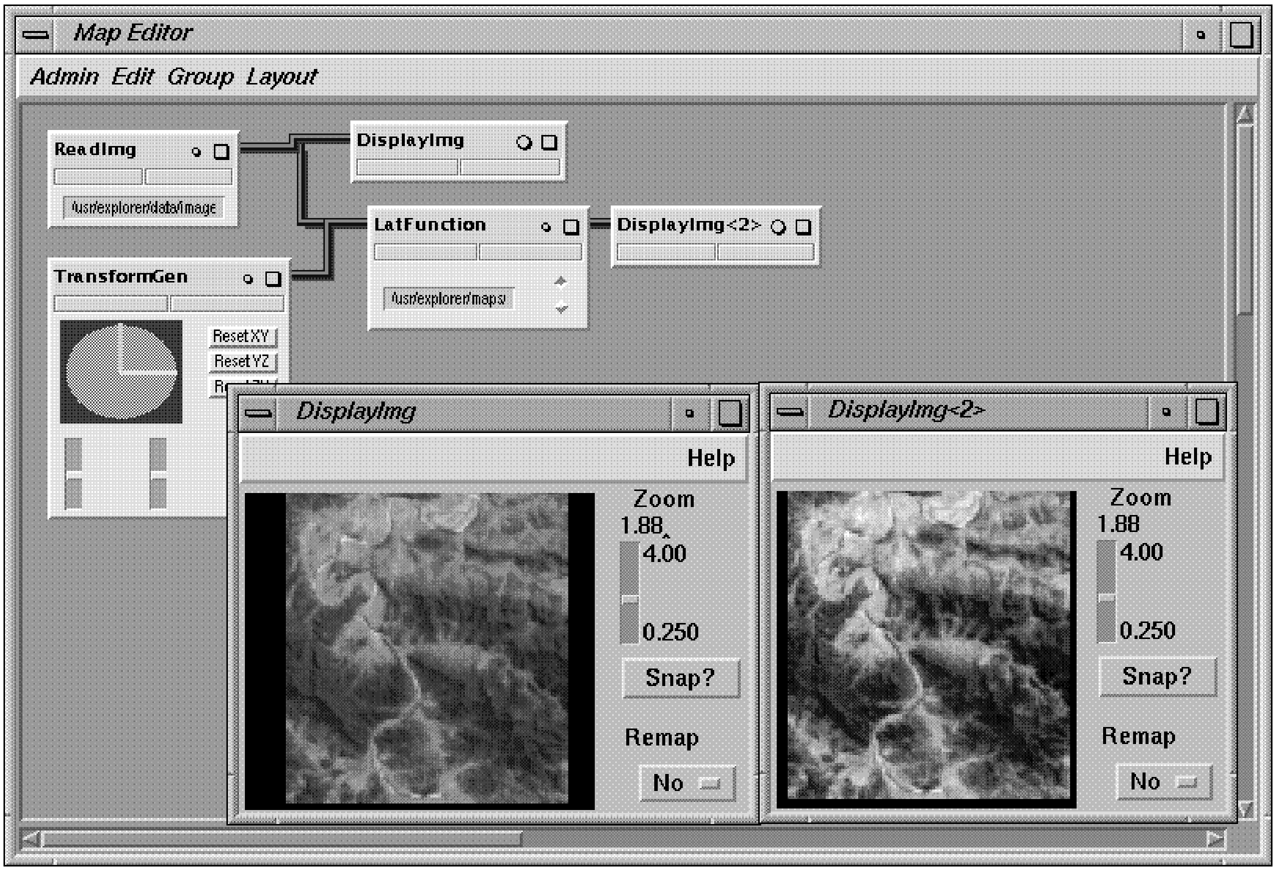


Figure 10−4 Output from the color−xform Map

10.8.3. To Generate a Heat Flux

Use the heat−flux map in the Librarian to view this explicit time−stepping simulation of a heat flow problem. The Shape program may be found in */usr/explorer/src/MWGcode/LatFunc/Shape/Heat_flux.shp*

```
// Animate a heat flow equation for 100 time steps and produce an output
// lattice each 5 time steps. The equation iterated at each timestep is
//              T += 4 h^2 (1-k) Laplacian( T );
// where h is the grid spacing and k is a relaxation parameter.
// T is the temperature.
//
// No input ports
// Port First_Out: Temperatures on square grid every 5 iterations
// After 100 iterations the temperature range is approximately [-80 ... 80].

   int stepPerIter = 5;

   n := 50;

   float next[n, n];            // space for next iteration
   float temps[n, n];           // current temperatures starting at 0.0
```

```
// Alias a heat source and sink in the grid
   source := subshape([20, 30], [29, 44], temps);
   sink   := subshape([30, 20], [39, 24], temps);

// Alias neighboring cells in the four directions plus center
// and the current temperatures.
   M := subshape([1, 1], [n-2, n-2], temps);    // all but edge of temps

   N := subshape([0, 1], [n-3, n-2], next);     // north
   S := subshape([2, 1], [n-1, n-2], next);     // south
   E := subshape([1, 0], [n-2, n-3], next);     // east
   W := subshape([1, 2], [n-2, n-1], next);     // west
   C := subshape([1, 1], [n-2, n-2], next);     // center

// The iteration is T += 4 h^2 (1-k) Laplacian( T );
   k := (float) 0.3;  // weight of C
   int i;
   for (i = 0; i <  100; i+=1) {
     source += (float) 10.;
     sink   -= (float) 10.;
     next = 0;
     N += M; S += M; E += M; W += M;
     next *= (float) 0.25 * (1 - k);
     C += M * k;
     temps = next;
     if (i % stepPerIter == 0) {

// And produce a scalar lattice for rendering.
       First_Out := scalar_lattice_out(temps);

       write(First_Out, oport_First_Out);
       flush_output_port(oport_First_Out);
     }
   }
```

Figure 10–5 shows what the output of the heat–flux program looks like in the *Render* module.

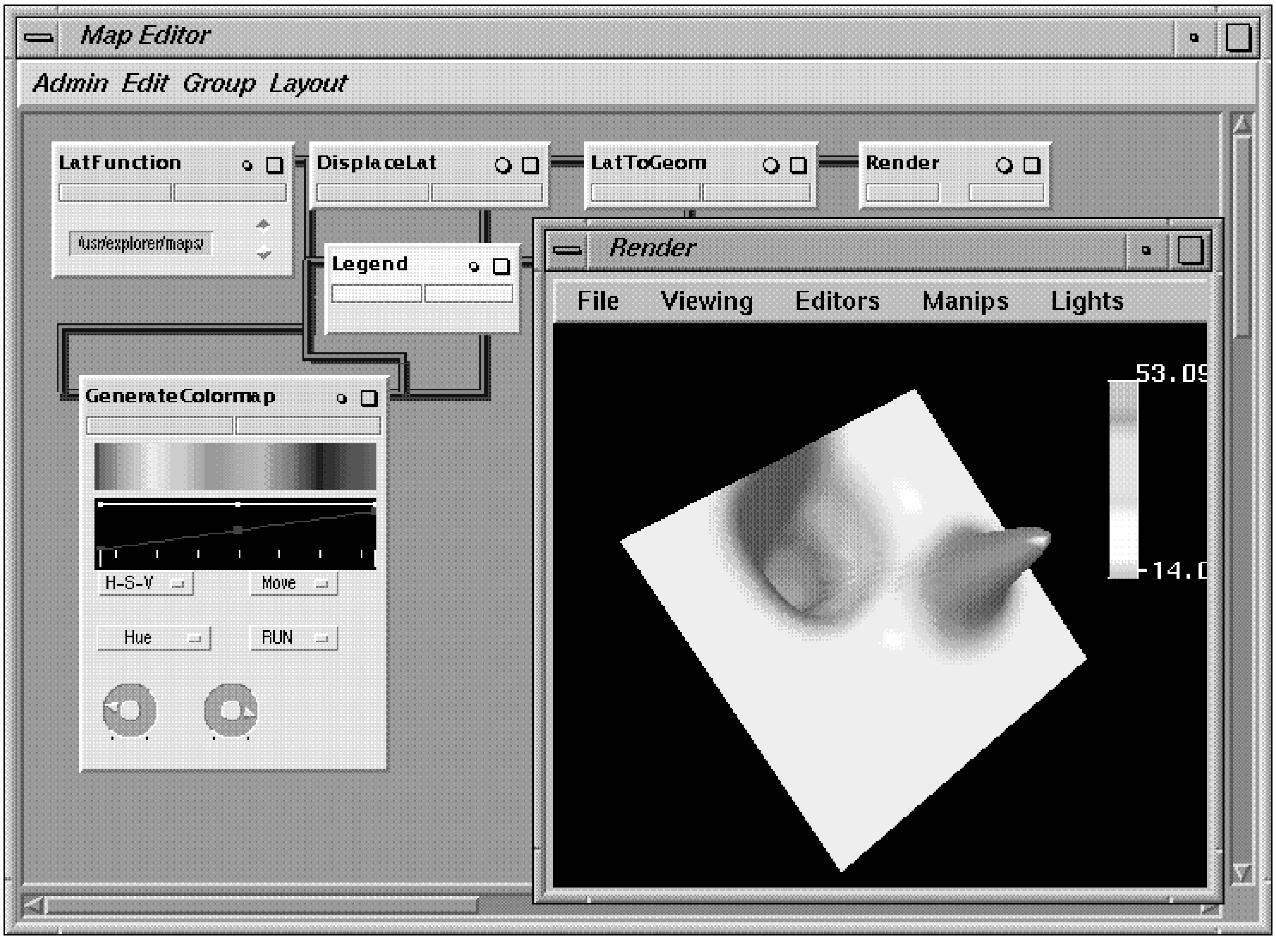


Figure 10–5 Output from the *heat–flux* Map

Appendix A

Using Makefiles

The IRIS Explorer product provides UNIX tools to simplify the automatic building of modules after the Module Builder has created the module resources. This section describes the tools, their use in the creation of a single module, and their integration with the *make* utility in building a source tree that has several modules. It is not necessary to be an expert on *make* to understand this section, although a little knowledge is helpful.

While it is convenient to create the executable image for a module from the Module Builder, it is often necessary to create or modify modules from the UNIX shell without using a graphically based tool. This need is most common in connection with automatic compilation of entire source trees, most often with the *make* utility.

The structure assumed in presenting these tools is that the sources and module resources for one or more modules are kept together in a directory with no other user programs. The Module Builder handles all the necessary features of creating the *Makefile* to govern module building within this directory. It is also possible to have several sibling directories, each containing module programs and resources. The Module Builder tools provide the *Makefile* ingredients necessary to govern the building of all modules in all subdirectories with minimal user intervention.

A.1. Creating a Module

For purposes of illustration, consider the *Contour* module supplied with IRIS Explorer. The source for Contour can be found in directory */usr/explorer/src/Contour.* The same description applies equally to any other IRIS Explorer module.

The general procedure for creating modules is:

- Supplying the module's program sources or objects.
- Creating the module resources in the Module Builder.
- Building and installing the module.

Once the module has been built, you can rebuild the module from the UNIX shell (within the *Contour* module directory) by typing:

```
make Contour
```

Do this whenever you change the source files of the *Contour* module. Running `make` does not result in any change in the interfaces to the Module Data Wrapper or Module Control Wrapper.

Note: When these interfaces do change, for example, when you add or delete a function argument or a port, or change widget types, you must run the Module Builder to update the module resources.

A.1.1. Installing a Module

When running IRIS Explorer, it is useful to keep the key components of all modules, their executables, resources, documentation, and credits files in one directory. This simplifies module selection with the Module Librarian. Each module's *Makefile* (generated by the Module Builder) has an action called *install*, which copies the mentioned files to a definable installation directory. *install* also causes the module to be built, if necessary. To build and install the module, type:

```
make install
```

Note: **`make install`** builds and installs all modules within the current directory; there is no command to allow installation of only a single module from within a multi−module directory.

A.1.2. Build and Installation Options

In some implementations IRIS Explorer can use shared libraries for the Module Control Wrapper and the API library. You can enable this feature by setting the *CXBUILDSHARED* environment variable and remaking the module *Makefile,* as follows.

```
setenv CXBUILDSHARED 1
make Makefile
make all
```

Setting the *CXINSTALLSYMBOLIC* variable when the module's *Makefile* is created causes a module to be installed with a symbolic link from the installation directory back to the module directory. For example, this sequence causes symbolic installation of the module:

```
setenv CXINSTALLSYMBOLIC 1
make Makefile
make install
```

Symbolic installation can save a large amount of disk space, as many modules are half a megabyte or more in size.

The *CXINSTALLSTRIPPED* environment variable controls the stripping of the module's symbol table to save disk space (at the expense of debugging information). Setting *CXINSTALLSTRIPPED* before the module is installed causes the installed module to be stripped. This removes symbol table information and reduces module size significantly.

By default, `make install` installs modules in $*EXPLORERUSERHOME/modules.* Normally, you cannot write directly into that directory. To change the directory where modules are installed, define *EXPLORERUSERHOME* to contain the directory name where modules should be installed. They will be installed in a subdirectory under *$EXPLORERUSERHOME* named *modules.* You can override this on a per–module directory basis with the Module Builder's *User Makefile* option. Choose a filename, such as *myMakefile*, enter its name in the User Makefile slot on the Module Builder main window, and insert the following line in that file:

```
CXMODULES_INST = <pathname>
```

where <*pathname*> is the actual pathname where modules in this directory should be installed.

A.2. Generating Makefiles

In most cases, you do not need to recreate a module's *Makefile* from the UNIX shell. The *Makefile* is saved whenever you select *Save Resources* from the File menu or *Build* or *Build & Install* from the Build menu on the Module Builder's main window. However, it is also possible to recreate the *Makefile* from the UNIX shell, in case you accidentally delete it or change the environment variables discussed above.

The *cxmkmf* command is a shell script located in the */usr/explorer/bin* directory. If you don't have this directory in your path, you must specify the full command name:

```
/usr/explorer/bin/cxmkmf
```

A.2.1. Make Clean

The module *Makefile* also performs another useful action, cleaning automatically generated files out of the directory. To do this, type:

```
make clean
```

This command removes all automatically generated files except the module's *Makefile* and the related *Imakefile*. Cleaning also removes the module executable so that a new build will force recompilation and linking.

A.2.2. The MODULES File

Makefile generation depends on the file MODULES in the module directory. This file contains a list of the modules to be built in that directory. The generated *Makefile* contains rules for building, cleaning, and installing all modules listed in the MODULES file. The MODULES file is automatically updated by the Module Builder, so you never need to touch it. However, you can remove a module from the build process by deleting its name from the MODULES file.

A.3. Using Imake in Module Building

The IRIS Explorer *make* environment relies on the X Window System tool *imake* to provide machine–independent *Makefile* generation and execution. All *Imakefile*s and *Makefile*s in the module directories are generated automatically by the Module Builder, so you never need to edit those files. Any changes you do make are likely to be deleted on the next invocation of the Module Builder.

You can find out more about IRIS Explorer's use of *imake* by looking at the files in */usr/explorer/config*. Most of these files are copies of the standard X Window System *imake* release with minor changes. They include:

- *Explorer.rules*
- *Explorer.tmpl*
- *Project.tmpl*
- *cxModules.rules*
- *cxModules.tmpl*
- *cxTypes.rules*

A.4. Using Generated Makefiles

The Module Builder automatically creates the *Imakefile* for you. You can add custom lines into the generated *Imakefile* by placing them in another file, then entering that filename in the User Makefile slot in the Module Builder main window. You can use this facility to modify the default build and install sequence by defining values for special *make* variables.

Some *make* variables can be changed and some cannot. **Table A–1** and **Table A–2** respectively list the variables.

Table A–1 lists the *make* variables that you can change in a user *Makefile*.

Variable	Meaning
DEFINES	Definitions for cpp use in C, C++, and Fortran compiles, and in `make depend`.
CDEFINES	Passed to the C compiler
C++DEFINES	Passed to the C++ compiler
FDEFINES	Passed to the Fortran compiler
INCLUDES	Include directory specification for cpp use in C, C++, and Fortran, and in `make depend.`
CINCLUDES	Include directory specification for
C++INCLUDES	Include directory specification for cpp use in C++ compiles
FINCLUDES	Include directory specification for cpp use in Fortran compiles
LOCAL_LDFLAGS	Loader flags to be passed when linking modules
LOCAL_LIBRARIES	Library filenames relative to the local directory (or fully qualified). Used in linking modules and in calculating dependencies so that the module will re–link if any of these libraries changes.
SYS_LIBRARIES	System libraries (given in the –L<*dir*> –l<*lib*> form) to be passed to the loader.

Table A–1 User–definable make Variables

Table A–2 lists some of the *make* variables you should not change. They are used by the built–in configuration:

Compiler	Compiler Flags	Defines	Includes
CC	CFLAGS	ALLDEFINES	ALLINCLUDES
C++	C++FLAGS	ALLC++DEFIN ES	ALLC++INCLUD ES
FC	FCFLAGS	ALLFDEFINES	ALLFINCLUDES
LDLIBS	LDOPTIONS		

Table A–2 System–required Variables

A.5. Default Makefile Actions

The generated *Makefile* performs two additional actions when a *make install* command is given at the UNIX shell, or when you select "Build & Install" from the Build menu. If the *modulename.doc* and *modulename.credit* files do not exist, the *Makefile* creates prototype documentation and credits files with those names.

The creation of prototype module documentation is described in **Chapter 2, "Understanding the Module Builder"**.

A.5.1. Installing Credit Files

In general, IRIS Explorer module credit files are image files in TIFF format, though some implementations support other image formats.

Because credit image files are typically large, IRIS Explorer offers a variety of ways to install them, allowing for symbolic links to existing image files to be created automatically. Understanding this procedure will provide maximum flexibility and control over the image presented when you select the "Credits" item on the module menu. It also reduces the amount of disk space required.

Note: The prototype credits file is a link to *UserModule.credit* in the */usr/explorer/lib* directory.

Sometimes, however, you may want a custom credit image file for modules you create. In this case, you can define an environment variable named CXCREDITS to contain the file name of the credit image file. This variable is consulted when the module *Makefile* is created and not consulted again when the credit link is installed. If, when the *Makefile* is made, this variable is defined, it overrides the default location of the credit file given above.

You may want a different credit file for each module development subdirectory. In this case, you need only define a user *Makefile* in the Module Builder main window and place in that file a line, for example:

```
CXCREDITS = filename
```

where filename is the full name of the credit file for the modules in this directory.

For each module in a module development subdirectory, if there exists a file with the same base name as the module but with the suffix *.credit*, this file will be installed as the credit file for that module. So, for example, there may be a module named *Shape* in a development subdirectory. If there were also a file named *Shape.credit*, it would be installed as the credit file for this module.

In all cases but the last, the installation of the credit file is done with a symbolic link, so very little disk space is taken by the credit files.

Appendix B

The Firing Algorithm

This appendix describes the principles of module firing and describes how the firing algorithm works.

B.1. Module Firing

Module firing is controlled by the MCW through the firing algorithm. The task of the firing algorithm is to gather and organize data on each input port and send the data in an orderly fashion to the user function, then to retrieve it from the output port(s) and send it to the next module.Three types of events affect module firing in an IRIS Explorer map. They are:

- Topology changes, which affect how a map fires
- Data arrival, which controls when a module fires
- Module firings themselves, which create data that may in turn cause other modules to fire

Topology changes are caused by these events:

- launching a module
- module deletion or core dump
- wiring connections and disconnections
- enabling or disabling a module or modules

Data arrival includes:

- the transfer of a widget value from the GUI to a module
- the arrival of a dataset from an upstream module

Module firing may result from:

- the arrival of a data frame on a module port
- a spontaneous event, such as an X Window event or a select call
- a repeated call, for example, from *cxFireASAP*
- multiple flushes of output data during a single firing

The map topology controls the propagation of *frames* of data. A frame is the unifying principle into which data arriving on a module port is fitted. Because the firing algorithm is distributed among modules, each module needs to know some part of the map topology in order to decide when to execute.

Each module stores a topology influence table that lists, for each connection, which upstream modules the connection depends on for data. If we think of each connection as storing a bit−vector corresponding to all modules, then that connection marks bits true for each upstream module. The list of modules includes the UI control panel that sends widget values to the module. For example, the map in **Figure B−1** generates the dependency table for *DisplayImg* shown in **Table B−1**.

	Source				
Connection #	ReadImg<2>	ReadImg	BlurImg	SharpenImg	DisplayImg (UI)
1	n				
2		n			
3		n	n		
4		n		n	
5					n

Table B−1 Topology Influence Table for DisplayImg

The connection numbers in the table relate to the numbered wires in the map.

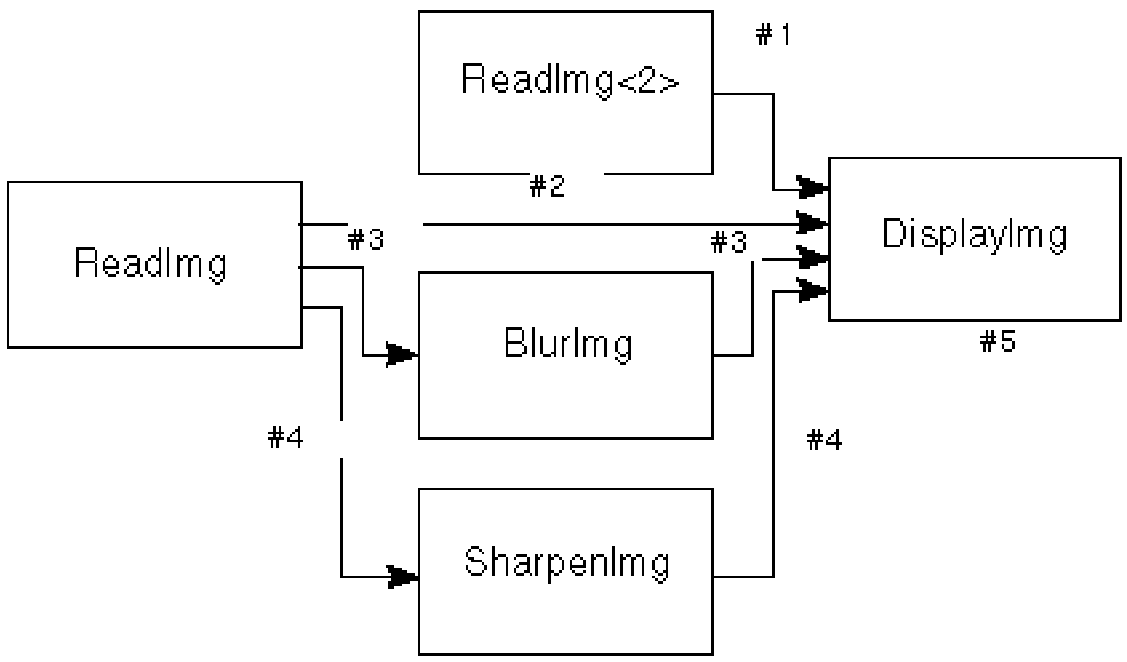


Figure B−1 Map Topology

The module employs the table by ANDing the frame ID bit−vector against each connection bit−vector to see whether data should be expected on that connection. This allows the multi−module frame IDs that are needed by the UI for vector widget settings.

B.1.1. Handling Topology Changes

When map topology changes, modules must be informed of the new topology so that they can correctly update their topology influence tables. No further action is required when topology changes do not coincide with propagation of data frames. However, when there are partial frames in existence, special action is needed to handle them.

On deletion of a connection, all frames waiting for data on that connection can mark that data as arrived (but old). This prevents frames from waiting indefinitely for data that will never arrive. It is especially important when disconnections occur due to module death, because there is no possibility of the upstream module fixing up the frame.

When a new connection is made, the ensuing data that arrives on the connection is marked as special. That data goes into a separate frame and is sent to the input ports. This is so that input connection data updates the module as soon as possible.

B.1.2. Using Frames for Data Ordering

The assembly of multiple independent frames of input data within a module is one of the major innovations of this firing algorithm. The diamond map illustrates the basic concept of the firing algorithm. Under normal circumstance, when we fire module A, for instance through a widget manipulation, we expect B and C both to fire, and D to fire only once. The firing of D waits for the two inputs descended from A (from modules B and C). This firing behavior has these advantages:

- it avoids needless firing of D

- it fires D only on consistent inputs from an upstream event

Figure B–2 shows the possibility for confusion when a map is fired and several datasets move downstream. There are four modules, A, B, C, and D, and six connections that we are interested in. Module A is wired to modules B and C and modules B and C both have two connections to module D. Each dataset flowing through the map carries with it information about the module that initiated it. Each module contains a topology influence table that allows it to compute the set of ports to invalidate in the event that another port receives data from a given upstream module. In this case, D's table says that both ports depend on data from module A, so that arrival of data from module A on port 1 (via module B) should also invalidate port 2 (from C).

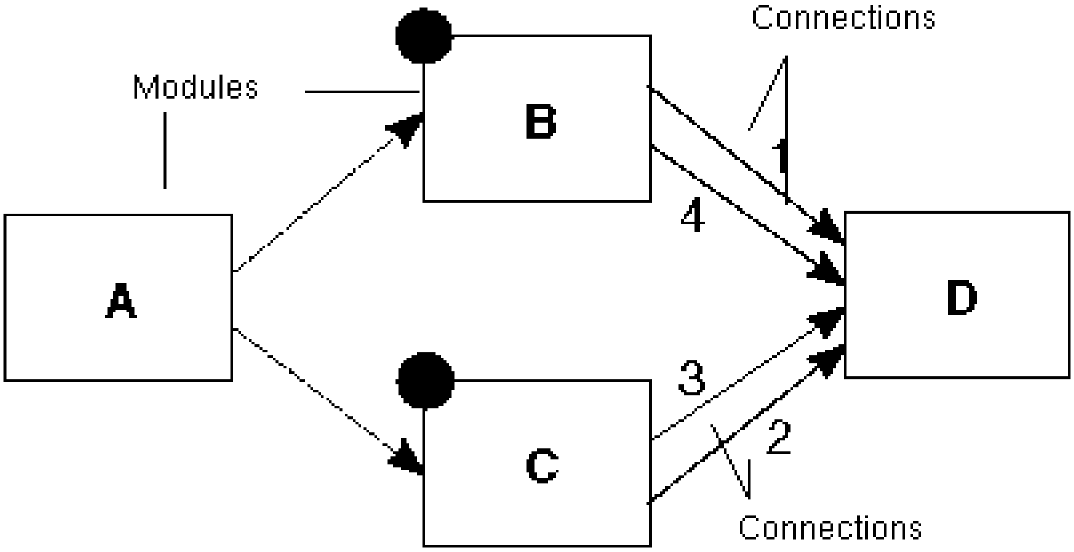


Figure B–2 Multiple Firing in a Map

This version of the diamond map shows two firings, one each of modules B and C. Now B and C each have two data connections to module D. Because B and C are fired almost simultaneously, we could end up with a race condition between dataset arrival in module D. The arrival ordering is numbered on the respective connections. The correct firing behavior of D is to fire twice, once with the outputs of B and once with the outputs of C. It would be wrong to fire only once with all four datasets, which we call frame mixing.

The correct order of firing on D in this example is with datasets (2,3) first because this is the first frame to be completed. This speeds execution of the map. D's inputs on the first firing are `1-previous, 2-current, 3-current, 4-previous.` On the second firing, they are `1-current, 2-current, 3-current, 4-current`. It does not swap out datasets 2 and 3 when firing on datasets 1 and 4.

To summarize:
- frame mixing is to be avoided
- a module must assemble independent frames of data for each upstream firing
- a module fires when a frame is completed (filled)
- a module fires on frames in their order of completion, not order of formation

B.1.3. Identifying Frames

To avoiding data loss, we require multiple independent frames created in D to include the possibility of several frames from the same originator. And in order to correctly associate data with its frame, each dataset in the map, and each frame being collected in a module must carry a frame identifier which must:
- be unique

- identify the originating module
- identify the sequence order of creation on the originating module.

The set of frame identifiers forms a partially−ordered space. Frames with different originators are unordered, while frames from a single originator are ordered by their sequence order of creation.

The information carried in the frame includes:

- the originator ID, that is, the set of modules causing this frame
- the originator sequence number
- an (optional) predecessor ID
- an (optional) predecessor sequence number

The originator ID and sequence number uniquely identify a frame. The extra predecessor information is used to add dependency on a previous frame in the case of multiple output data flushes caused by arrival of an upstream frame. This helps to maintain the partial ordering of frames in downstream modules.

Because data cannot leapfrog through the map, we can be sure that two frames of data from a single originator will arrive in the same order at all downstream modules in the map. The non−leapfrogging of data in a map need not imply that those datasets are in ordered frames, however.

Propagation of Frame IDs

Figure B−3 illustrates how frame identifiers can keep data ordered when there is a multiple data flush. When a module fires due to an upstream event, it copies the frame identifier from its input frame over to its output frame. This allows it to propagate the correct frame identifier through the downstream portion of the map. But when a module sends several sets of output by using the *cxOutputDataFlush* routine (three shaded circles on B.*), assigning that upstream frame identifier to all of its output frames would force the downstream map of out sync, creating a deadlock (module C.1 would expect multiple inputs from A.1 on its first port). Thus the second and all subsequent flushes must create a new frame identifier listing the flushing module as the originator.

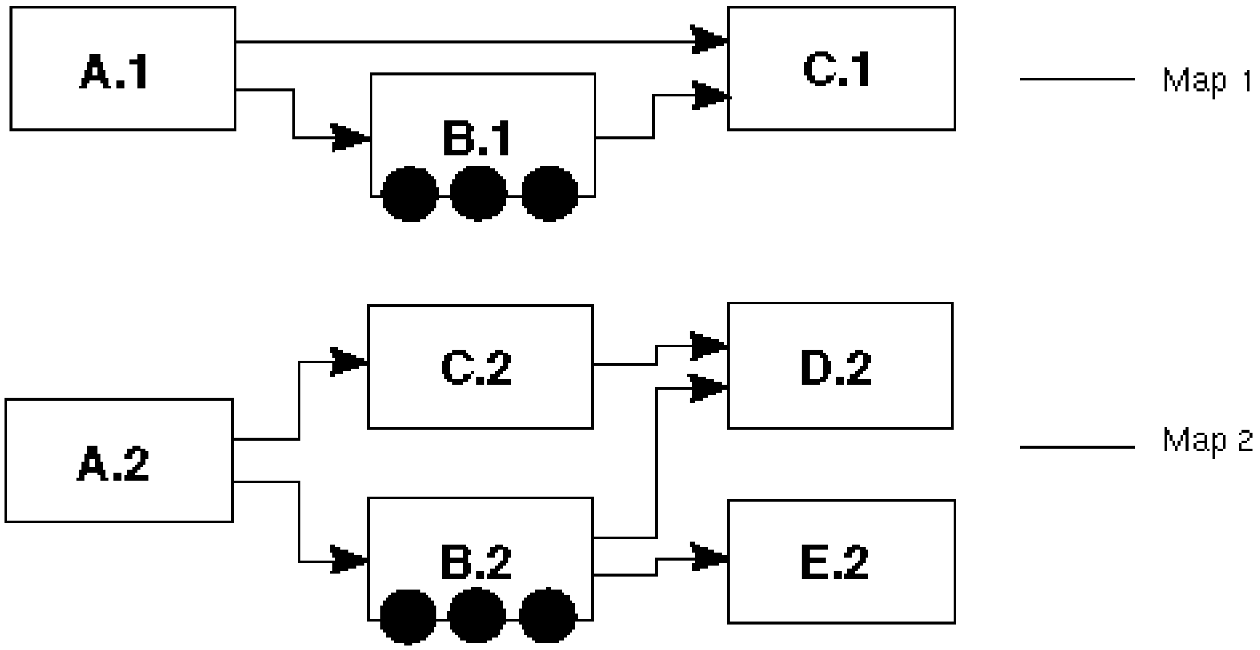


Figure B−3 Frame Identifiers

Frame Ordering

B.*'s frames must be ordered with respect to A.*'s frames. Merely following them in the map along the path to C.1 is not enough, for there could be a race condition if there were other paths to C.1. For instance, in the second map in **Figure B−3**, if C.2 takes a long time to arrive, then multiple complete frames from B.2 may arrive at D.2 before the A.2 frame is completed at D.2.

The B.2 frame is dependent on an A.2 frame because each frame carries dependency information, some implicit, some explicit. The explicit information is of the form "B.2(1) depends on A.2" for the examples above. The implicit information is that $X(n)$ depends on $X(m)$ if $m < n$, the usual execution ordering. A single predecessor is sufficient, because of transitivity of dependencies.

If D.2 fires when it gets a complete frame from B.2, without waiting for the A.2 frame, D.2 could end up with inconsistent data, some from the A.2 and B.2 frames (through B.2) and some from before the A.2 frame (through previous data from C.2).

B.1.4. Frame Construction

Figure B−4 shows a map with two firings. Modules B and C have each output data from two ports, and *MyModule* has received the data on three of its ports.

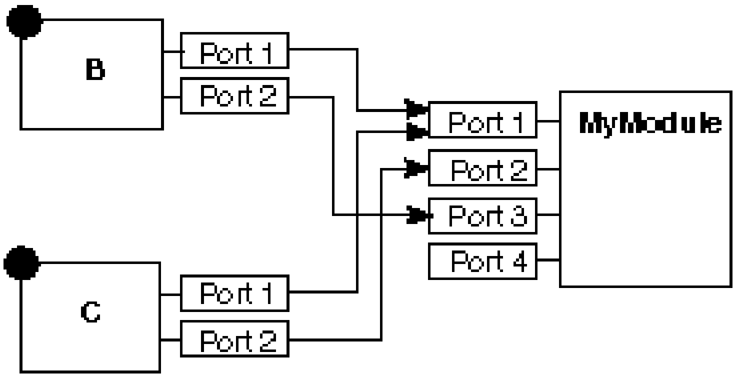


Figure B−4 Map Firings

Figure B−5 shows a snapshot of the firing process. It illustrates the frame assembler after it has received Port 1 data from B and Port 2 data from C. Each frame has a slot for each port, with these three possible states:

`Waiting`	data expected on this connection
`Data`	data has arrived on this connection
`......`	no data expected on this connection

When all "waiting" slots are filled, and if this frame has no dependencies on incomplete frames, the datasets are sent to the ports of the module. No datasets are sent until the frame fills.

The data structure used to represent the incomplete frames in a module is known as a "forest of trees". Each independent tree represents an independent firing upstream of the module. All frames within a single tree represent dependent frames, typically the result of repeated module flushes. Any complete frame at the root of a tree can be sent to the module's input ports.

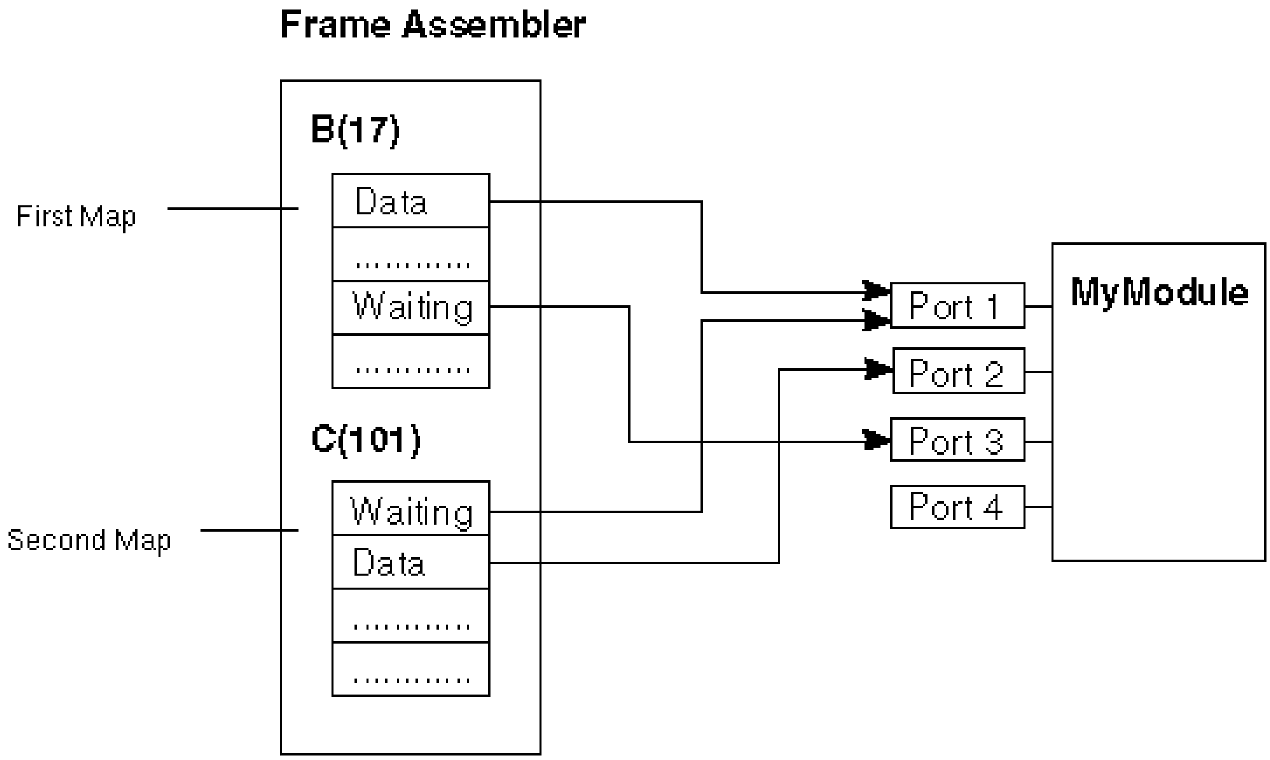


Figure B−5 Frame Assembly

Setting Frame Dependencies

The dependency arcs in the frame graph are constructed from the predecessor information and from the previous frame table. This table lists, for each upstream module, the latest frame originated by that module that is still pending in the frame graph. When a new frame arrives from that same module, the previous frame information is used to create a dependency arc and the table is updated.

The actual computation is: when data arrives on a link, for each module in the union of Frame ID and topology influence table, make this frame depend on previous frame from that module. When the last partial frame from an upstream module is filled and enqueued for processing by the ports, the relevant entry in the previous frame table is emptied, for there are now no partial frame dependencies on that upstream module.

You can store all dependency arcs in the upstream partial frame, which simplifies scheduling of frames. When data arrives on a frame with no dependencies, that frame is enqueued. When the frame is processed by the ports, all partial frames that depend on it have their dependency counts decremented. This has two advantages: it does the correct breadth−first scheduling of completed frames, and it uses the ports queue to help out, rather than implementing a second queue for scheduling. As a result, each frame also has a dependency count of the number of previous partial frames it depends on.

B.1.5. Sending Data Downstream

Several API routines let you send multiple ports at once during a simulation. These include:

cxOutputDataFlushAll

It flushes all ports, sending new data on ports with a recent *cxOutputDataSet*, and null data tokens elsewhere.
This allows a simulation writer to load several output ports and flush them all in a single frame.

This allows a simulation writer to load several output ports and flush them all in a single frame. The net effect is that *cxOutputDataSet* loads data into a frame, while *cxOutputDataFlush* and *cxOutputDataFlushAll* copy a (partial) frame to the output ports and send it downstream.

cxOutputDataSendNow

It advances the current output frame's data on a single connection to the output port and thence to the downstream modules, marking that port as sent. No other *cxOutputDataSet* or *cxOutputDataSendNow* commands may take effect on this port until a *cxOutputDataFlushAll* is called to flush the rest of the frame. During a *cxOutputDataFlushAll* call, nothing is transmitted for sent ports, since their data have already been sent.

The purpose of *cxOutputDataSendNow* is to support asynchronous parallel execution of downstream sub−maps, by sending data as soon as they are ready, rather than erecting a wait barrier on completion of the entire output frame.

For instance, the map in **Figure B−6** shows how many downstream modules can fire before the reader has sent output on all ports. The independent flows of data are labelled 1, 2, 3, and 4 on the output arcs. Modules are labelled with corresponding numbers to show which dataset sending causes them to fire. One module can fire when the first dataset is emitted from A, two on the second dataset, one on the third, and three on the fourth.

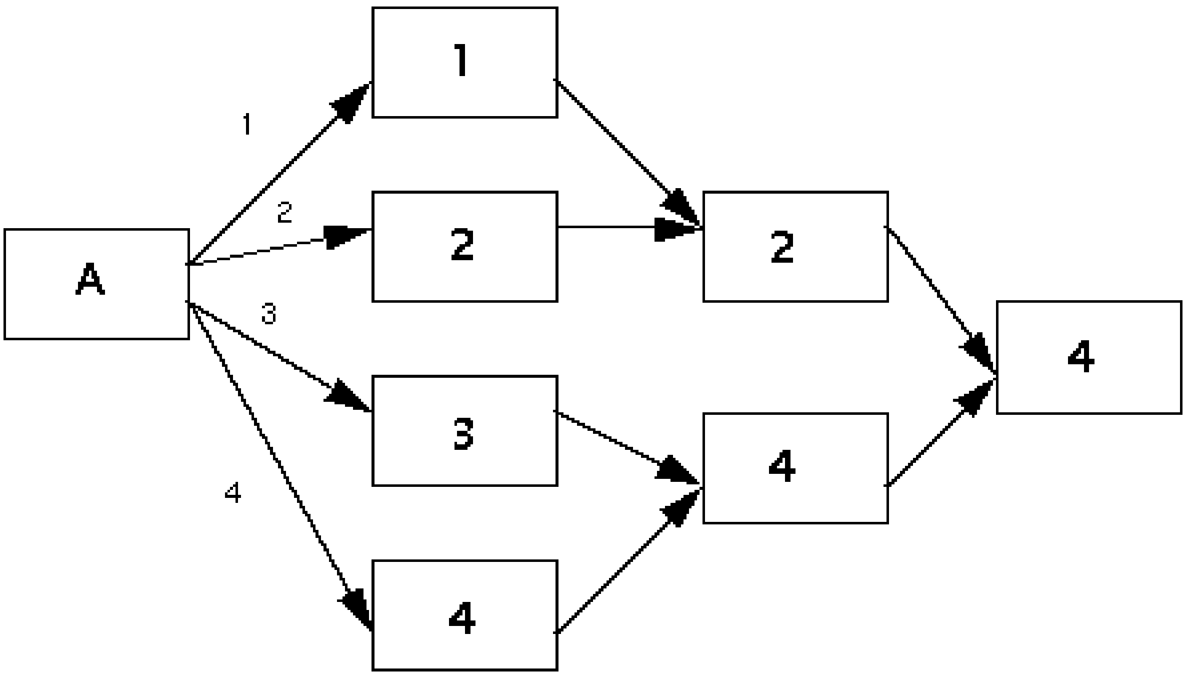


Figure B−6 Parallel Data Flows

B.1.6. Handling Fan−in Connections

To handle fan−in of input data, data are gathered per connection, rather than per port. This allows multiple datasets to be gathered into a single frame with no loss of data and no additional firings.

The frame is not considered complete until all of its connections, be they single or fanned in, present data from the upstream frame. An entire, completed frame of data is them delivered to the module's ports simultaneously. This means that a fan−in port will receive multiple datasets together.

You have access to these datasets through *cxInputDataGetV*, which returns a pointer to a static vector of dataset pointers.

cxInputDataGet returns a single dataset pointer even when the port contains a vector of datasets. The routine returns the new dataset from the most recently wired connection or, if there is no new dataset, the previously returned dataset.

cxInputDataChangedV returns a static Boolean (int) vector of dataset state.

The routines *cxInputDataConnIDGet* and *cxInputDataConnIDGetV* return the connection identifier(s) for a given port, in wiring order. You are referred to the *IRIS Explorer Reference Pages* for a full description of their functionality.

Note: Only one dataset can flow on a single wire at a given time.

B.1.7. Merging Frames on Input Ports

When a new frame of data has been filled and finally made it through the frame queue, it arrives at the input port, where its datasets are merged with the previous port data by overlaying new data. Old data on a link with new data are replaced, while all links not carrying new data in the frame are marked as old on the port. The link with the newest data, either the most recently wired link with new data, or the most recently wired with old data, becomes the focus for the single dataset API. If the newest link is deleted in the arriving frame, then that frame's newest link takes on that role.

This sequence of events is shown in **Figure B–7**.

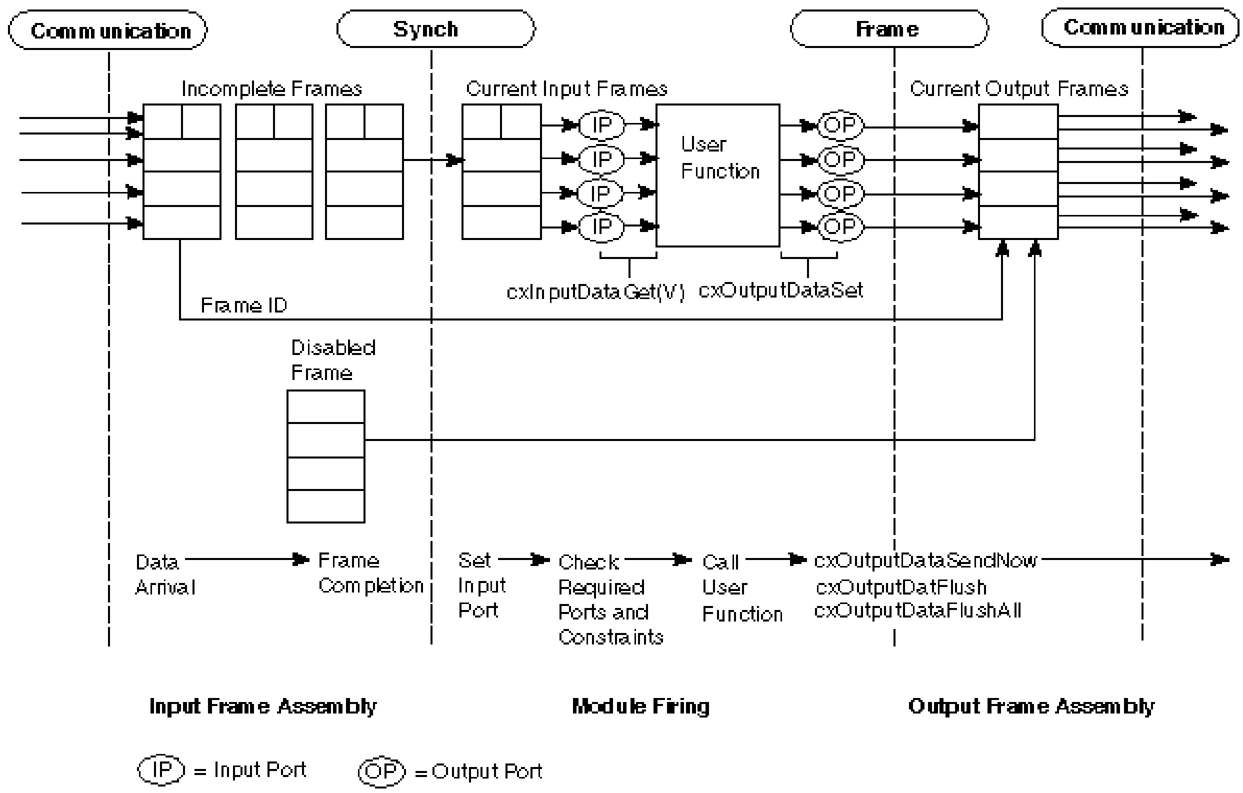


Figure B–7 Data Handling in a Module

After the data are merged into the ports, each new dataset is checked for constraint correctness. If a dataset fails, it is released from the port's link (making that link empty), a `modAlert` is issued, and the user function is not fired.

The user function might be legally fireable, for instance if the port with bad data were optional, so that an empty port would be acceptable. However, since the new dataset on this frame was not legal, firing the module in the absence of that dataset would not give the correct or expected results. If multiple datasets are in error, they can be flagged and rejected all together.

A fan−in port with five links and only four datasets (i.e., one link is empty) will not fire if the port is marked *Required*. Thus a port's links must each meet the *Required* test for data before the port is considered satisfied.

Disabled Modules

A disabled module handles new frames by merging them onto the input ports, then sending a NULL (synchronization) frame downstream; the user function is not called. When the module is enabled, if there is an unused partial frame merged into the input ports, then the user function is called.

B.1.8. Incorporating Parameters

Parameters both with widgets and without widgets now live in modules. They all obey the firing algorithm, which eliminates unwanted multiple firings. This leaves the GUI to perform two parameter−related actions:

- sending parameter updates as a result of widget manipulations
- displaying widget values as a result of incoming parameters.

Parameter functions (P−Funcs) are dealt with in the same way: their ownership is pushed down to the module.

Setting Widgets

In order to change several widget values simultaneously, there must be a single firing frame for the multiple widget values. You can thus make several changes to disabled modules, then re−enable the modules together, and the resulting parameters are sent as a vector of widget settings. This is the widget analogue of a frame of data comprising outputs on multiple ports.

The event that causes a vector of widget settings is the enabling of a module or modules. This can be done with the Enable menu item on a module control panel, with the Enable All menu item in the Map Editor, or by launching a module or map. In each case, the subsequent set of module enables is considered to be a single, atomic event.

This event is tagged by including the UI module identifiers of all relevant modules in the frame of data propagated from the UI modules into the map. Thus instead of having a single initiator, the frame has several UI module initiators. The fact that we can represent them all in a single frame means that we can synchronize the firing due to the multiple enables.

There are four times when a widget value must be set:

- when the widget is changed in the GUI
- when a *cxInWdgtXXXSet* call issues from the module
- when a new parameter value arrives from an upstream module
- when a Pfunc creates a new value

However, only the latter three require a message from the module to the GUI, since the GUI creates the new parameter value in the first case. When a new parameter arrives from an upstream module, the widget is not set until the input data frame is delivered to the input ports.

When there are both widgets and Pfuncs on a fan−in port, the user sees either the fan−in connections or the Pfuncs, but not both.