Prism 5.0 User's Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300 fax 650 969-9131

Part No.: 805-1552-10
Revision A, November 1997

Contents

Preface

The *Prism User's Guide* explains how to use the Prism programming environment to develop, execute, debug, and visualize data in serial and parallel programs.

These instructions are intended for application programmers developing serial or parallel programs that are to run on a Sun™ HPC System. We assume you know the basics of developing and debugging programs, as well as the basics of the system on which you will be using Prism. Some familiarity with the UNIX® debugger dbx is helpful but not required. Prism is based on the X and OSF/Motif standards. Familiarity with these standards is also helpful but not required.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- AnswerBook™ online documentation for the Solaris™ 2.x software environment
- Other software documentation that you received with your system

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output.	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output.	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Command-line variable; replace with a real name or value.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be `root` to do this. To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
All	*Sun HPC Software 2.0 System Administrator's Guide*	805-1554-10
All	*Sun HPC Software 2.0 Release Notes*	805-2191-10
Sun MPI Programming	*Sun MPI 3.0 Guide*	805-1556-10
Prism	*Prism 5.0 User's Guide*	805-1552-10
Prism	*Prism 5.0 Reference Manual*	805-1553-10
Sun HPF Programming	*Sun HPF 1.0 Guide*	805-1558-10
S3L	*S3L 2.0 Guide*	805-1557-10

Ordering Sun Documents

SunDocs[SM] is a distribution program for Sun Microsystems technical documentation. Contact SunExpress for easy ordering and quick delivery. You can find a listing of available Sun documentation on the World Wide Web.

TABLE P-4 SunExpress Contact Information

Country	Telephone	Fax
Belgium	02-720-09-09	02-725-88-50
Canada	1-800-873-7869	1-800-944-0661
France	0800-90-61-57	0800-90-61-58
Germany	01-30-81-61-91	01-30-81-61-92
Holland	06-022-34-45	06-022-34-46
Japan	0120-33-9096	0120-33-9097
Luxembourg	32-2-720-09-09	32-2-725-88-50
Sweden	020-79-57-26	020-79-57-27

TABLE P-4 SunExpress Contact Information

Switzerland	0800-55-19-26	0800-55-19-27
United Kingdom	0800-89-88-88	0800-89-88-87
United States	1-800-873-7869	1-800-944-0661
World Wide Web: `http://www.sun.com/sunexpress/`		

Sun Documentation on the Web

The `docs.sun.com` web site enables you to access Sun technical documentation on the World Wide Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email or fax your comments to us. Please include the part number of your document in the subject line of your email or fax message.

- Email: `smcc-docs@sun.com`
- Fax: SMCC Document Feedback
  1-650-786-6443

LSF Technical Support

LSF 3.0, a product of Platform Computing Corporation, is part of the Sun HPC Software 2.0 Foundation Package. As such, it is supported by Sun as part of Sun HPC Software 2.0.

Sun HPC Software includes LSF Base and LSF Batch. However, LSF JobScheduler and LSF MultiCluster are not included and, therefore, not supported by Sun.

Information Sources for PVM and PETSc

TABLE P-5 lists organizations and resources for information about the publicly available libraries PVM and PETSc. This information is subject to change.

TABLE P-5 Information Sources for PVM and PETSc

Product	Contact
PVM	Copyright holders: University of Tennessee, Oak Ridge National Laboratory, Emory University Electronic mail: `pvm@msr.epm.ornl.gov` Newsgroup: `comp.parallel.pvm` Web site: `http://www.epm.ornl.gov/pvm/pvm_home.html`
PETSc	Developed and supported by the Mathematics and Computer Science Division of the Argonne National Laboratory.

CHAPTER 1

Introduction

Prism is an integrated graphical environment within which users can develop, execute, and debug programs. It provides an easy-to-use, flexible, and comprehensive set of tools for performing all aspects of serial, data parallel, and message-passing programming. Prism operates on terminals or workstations running the Solaris™ operating environment under either OpenWindows™ environment or Sun's Common Desktop Environment (CDE). In addition, a commands-only option allows you to operate on any terminal, but without the graphical interface.

This chapter introduces Prism. Subsequent chapters discuss specific aspects of it in more detail.

1.1 Overview

You can either load an executable program into Prism, or start from scratch by calling up an editor and a UNIX® shell within Prism and using them to write and compile the program.

Once an executable program is loaded into Prism, you can (among other things):

- Execute the program
- Debug the program
- Visualize data from the program
- Analyze the performance of the program (data parallel programs only)

Prism supports message-passing programs, described in Chapter 10, "MP Prism." Prism also supports data parallel programs, although the view presented to the user is that of a single serial program.

1.2 The Look and Feel of Prism

FIGURE 1-1 shows the main window of Prism with a program loaded. It is within this window that you debug and analyze your program. You can operate with a mouse, use keyboard equivalents of mouse actions, or issue keyboard commands.

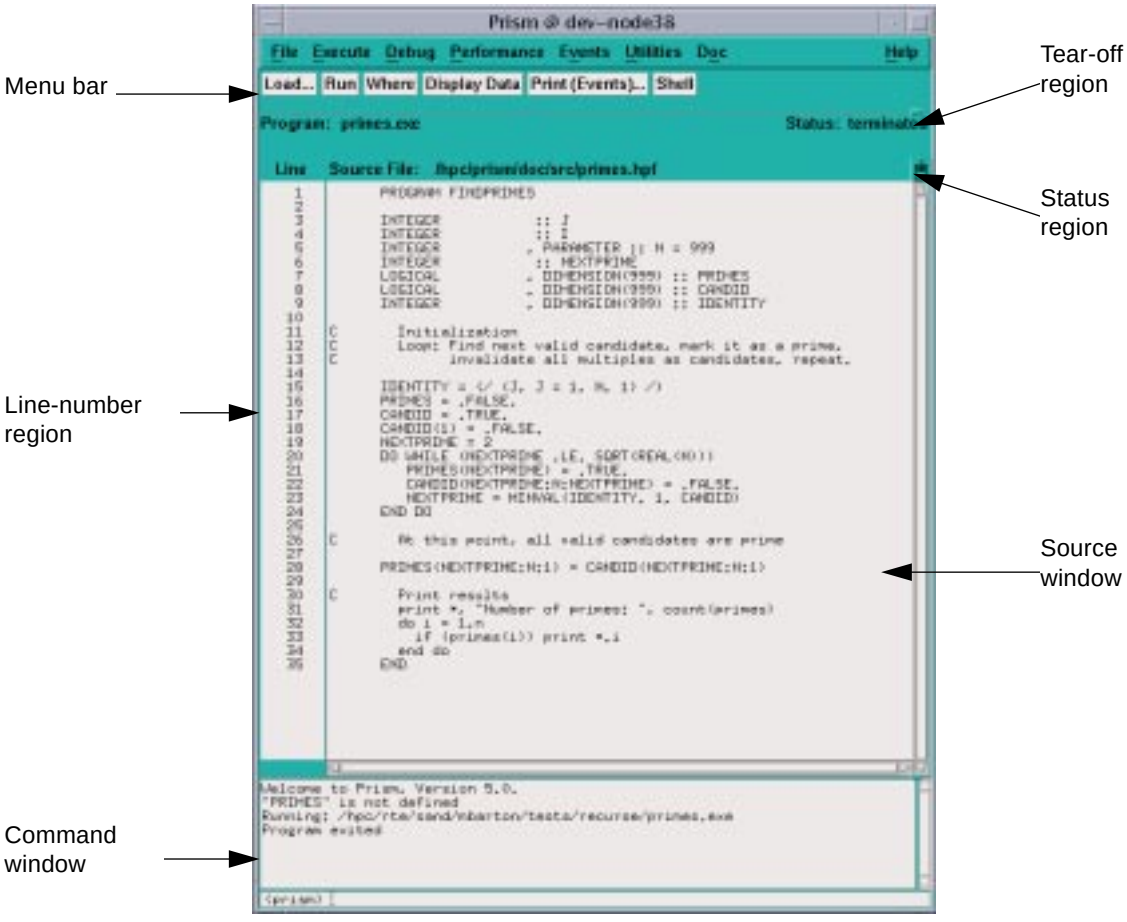

FIGURE 1-1 Prism's Main Window

Clicking on items in the *menu bar* displays pulldown menus that provide access to most of Prism's functionality.

You can add frequently used menu items and commands to the *tear-off region*, below the menu bar, to make them more accessible.

The *status region* displays the program's name and messages about the program's status.

The *source window* displays the source code for the executable program. You can scroll through this source code and display any of the source files used to compile the program. When a program stops execution, the source window updates to show the code currently being executed. You can select variables or expressions in the source code and print their values or obtain other information about them.

The *line-number region* is associated with the source window. You can click to the right of a line number in this region to set a breakpoint at that line. In FIGURE 1-1, a breakpoint is set at line 25.

The *command window* at the bottom of the main Prism window displays messages and output from Prism. You can also type commands in the command window rather than use the graphical interface.

General aspects of using these areas are discussed in Chapter 2.

1.3 Loading and Executing Programs

You can load an executable program into Prism when you start it up, or any time afterward. Once the program is loaded, you can run the program or step through it. You can also interrupt execution at any time.

You can also attach to a running program or associate a core file with a program.

Chapter 3 discusses these topics in more detail. See Section 10.4, "Executing a Program in MP Prism" for aspects of these topics that are unique to MP Prism.

1.4 Debugging

Prism allows you to perform standard debugging operations such as setting breakpoints and traces, and displaying and moving through the call stack. Chapter 4 discusses these topics. See Chapter 10 for a discussion of debugging in MP Prism.

1.5 Visualizing Data

It is often important to obtain a visual representation of the data elements that make up an array or parallel object. In Prism, you can create *visualizers* that provide standard representations of variables or expressions. For example,

- In the *text* representation, the data is shown as numbers or characters.
- In the *colormap* representation, each data element is mapped to a color, based on a range of values and a color map that you specify. (This representation is available only on color workstations.)
- In the *threshold* representation, each data element is mapped to either black or white, based on a cutoff value that you can specify.

A *data navigator* lets you manipulate the display window relative to the data being visualized. Options are available that let you update a visualizer or save a snapshot of it.

See Chapter 5 for a discussion of visualizing data. Section 10.7 covers aspects of visualization unique to MP Prism.

1.6 Analyzing Program Performance

Prism provides performance data that is essential for effectively analyzing and tuning Sun HPF data parallel programs. The data includes:

- User and system CPU time
- Time spent in various forms of communication
- Time spent performing I/O

The performance data is displayed as histograms and percentages (or elapsed times) for each computing resource. You can also obtain data on usage for each procedure and source line in the program. You can save the performance data in a file and redisplay it at a later time.

See Chapter 6, “Obtaining Performance Data,” for a discussion of performance analysis.

1.7 Editing and Compiling

You can call up the editor of your choice within Prism to edit source code (or any other text files). If you change your source code and want to recompile, Prism also provides an interface to the UNIX `make` utility. See Chapter 7.

1.8 Obtaining On-Line Help and Documentation

Prism features a comprehensive on-line help system. Help is available for each menu, window, and dialog box in Prism.

In addition to help on Prism itself, Prism on-line documentation is provided on the Sun AnswerBook™.

On-line help and documentation are described in more detail in Chapter 8.

1.9 Customizing Prism

You can change aspects of the way Prism operates. They are discussed in Chapter 9.

CHAPTER 2

Using Prism

This chapter describes general aspects of using Prism. Succeeding chapters describe how to perform specific functions within Prism.

See the following sections to learn:

- What to do before entering the Prism programming environment — *Section* 2.1.
- How to enter Prism — *Section* 2.2.
- How to perform actions within Prism — *Section* 2.3.
- How to use the menu bar — *Section* 2.4.
- How to use windows, dialog boxes, and lists — *2.5*.
- How to use the source window and line-number region — *Section* 2.5.
- How to use the command window — *Section* 2.7.
- How to write expressions in Prism — *Section* 2.8.
- How to work with Sun HPF generic procedures in Prism — *Section* 2.9.
- How to issue Solaris commands — *Section* 2.10.
- How to leave Prism — *Section* 2.11.

The best way to learn how to use Prism is to try it out for yourself as you read this chapter.

2.1 Before Entering Prism

2.1.1 Supported Languages and Compilers

You can work on Sun HPF, Fortran, and C programs within Prism. Specifically, Prism supports these compilers in Sun™ HPC Software 2.0:

- Sun HPF Release 1.0
- SPARCompiler™ Fortran 77 4.0 and 4.2
- SPARCompiler C 4.0 and 4.2
- `gcc`

Note – Prism does not support `gcc`'s extensions to Standard C. If you make use of these extensions, or if you use another compiler, you may receive error messages about bad symbol table entries.

2.1.2 Compiling and Linking Your Program

To use Prism's debugging features, compile and link each program module with the `-g` compiler option to produce the necessary debugging information.

Note – If you use the C or Fortran SPARCompiler, you must also use the `-xs` option when compiling and linking; this causes the debugging information to be placed in the executable program rather than in the object files.

If you are going to be collecting performance data for a Sun HPF program, compile and link with the `-tmprofile` option. Also, avoid using any optimization flags.

Note – If you use calls to tm_timer in your HPF program, the program (when run within Prism) will occasionally report failure to open a file of the form `/proc/<`*pid*`>`. Since Prism also uses files of the form `/proc/<`*pid*`>`, to avoid conflicts you must comment out calls to tm_timer in your HPF program if you wish to run the program within Prism.

2.1.3 Setting Up Your Environment

To enter the Prism programming environment, you must be logged in to a terminal or workstation running the X Window System (unless you want to run Prism in commands-only mode; see below).

Prism works under these X servers:

- OpenWindows
- CDE

Make sure that your `DISPLAY` environment variable is set for the terminal or workstation from which you are running X. For example, if your workstation is named valhalla, you can issue this command (if you are running the C shell):

```
% setenv DISPLAY valhalla:0
```

2.2 Entering Prism

Note – This section applies only to entering Prism to work with a serial or data parallel program. See Section 10.2 for more information on entering MP Prism to work with a message-passing program. For additional information on using MP Prism with Sun MPI programs, see Section 10.10.

2.2.1 Invoking Prism

Issue the `prism` command just as you would any program. For example, issuing the `prism` command at your Solaris prompt,

```
% prism
```

starts Prism on your login node in a Sun HPC System.

You can also start Prism as part of a tmrun or tmsub command, such as

```
% tmrun –p Thor prism
```

starting Prism on a node on partition Thor.

You can specify default option settings for `tmsub` or `tmrun` via the environment variable `TMRUN_FLAGS`.

For complete information on `tmrun`, `tmsub`, and `TMRUN_FLAGS`, see the *Sun HPC Software User's Guide.*

When Prism starts, you see the main window shown in FIGURE 1-1 in Chapter 1.

2.2.2 Command-Line Options

2.2.2.1 Loading a Serial Program

If you specify the name of an executable program on the command line, that program is automatically loaded into Prism. For example,

```
% prism primes.x
```

When you execute the program, it will execute on the node on which Prism is running.

See 3.1 for more information about loading a program.

2.2.2.2 Loading a Multiprocess Program

If you are loading a data parallel or message-passing program, you typically want to specify the number of processes it is to run. Use the `–np` option to do this. For example,

```
% prism –np 4 primes.x
```

You can also use the `-p` option to specify the partition in which the program is to run, and the `-s` option to specify the Sun HPC System (if you want it to run on a System other than the one to which you are logged in).

Finally, you can use the `-tmrun` option to specify any other `tmrun` options that you want to use to control the execution of the program. Enclose the options in quotation marks. (If the option uses quotation marks itself, precede each of them with a backslash.) For example,

```
% prism –np 4 –tmrun "–S" mprog.x
```

Once you have entered Prism, you can issue the `tmrunargs` command to specify any `tmrun` or `tmsub` options that you want to apply to your message-passing program. Prism stores these options, then applies them when you start up a multiprocess program. These options override any settings made via the `prism` command line, or via the `TMRUN_FLAGS` environment variable.

See 10.2 for more information about using these options and the `tmrunargs` command.

2.2.2.3 Attaching to a Process

You can also attach to a process that is currently running. However, Prism must run on the same node on which the process is running.

To attach to a process, add the process's process ID (pid) after the name of the program.

You can obtain the process's pid and the node on which it is running by issuing the Solaris `ps` command on a Sun HPC System. You can then issue a command like the following:

```
% prism primes1.x 2256
```

See Section 3.3 for more information about attaching to and detaching from a running process.

Note – In MP Prism, you can obtain the task ID (tid) by using the Sun HPC command `tmps`. See Section 10.2.4, "Attaching" for information about attaching to and detaching from processes and tasks using MP Prism.

2.2.2.4 Working With a Core File

You can associate a core file with a program. Add the name of the core file after the name of the executable program.

See Section 3.2 for more information about core files.

2.2.2.5 Specifying Commands-Only Prism

Use the `-C` option to bring up Prism in commands-only mode. This allows you to run Prism on a terminal with no graphics capability.

Use the `-CX` option to bring up a commands-only Prism that lets you redirect the output of certain Prism commands to X windows.

See Appendix A for information about commands-only Prism.

2.2.2.6 Specifying X Toolkit Options

You can include most standard X toolkit command-line options when you issue the `prism` command; for example, you can use the `-geometry` option to change the size of the main Prism window. See your X documentation for information on these options. Also, note these limitations:

- The `–font`, `–title`, and `–rv` options have no effect.
- The `-bg` option is overridden in part by the setting of the `Prism.textBgColor` resource, which specifies the background color for text in Prism; see Section 9.4.5.

X toolkit options are ignored, if you use `-C` to run Prism in commands-only mode.

2.2.2.7 Specifying Input and Output Files

You can use the form

`% prism <` *input–file*

to specify a file from which Prism is to read and execute commands upon startup. Similarly, use the form

`% prism >` *log–file*

to specify a file to which Prism commands and their output are to be logged.

If you have created a `.prisminit` initialization file, Prism automatically executes the commands in the file when it starts up. See 9.5 for information on `.prisminit`.

2.3 Within Prism

Within Prism, you can perform most actions in one of three ways:

- By using a mouse; see Section 2.3.1
- By using keyboard alternatives to the mouse; see Section 2.3.2
- By issuing commands from the keyboard; see Section 2.3.3

2.3.1 Using the Mouse

You can point and click with a mouse in Prism to choose menu items and to perform actions within windows and dialog boxes. Prism assumes that you have a standard three-button mouse.

In any window where you see this mouse icon:

you can left-click on the icon to obtain information about using the mouse in the window.

2.3.2 Using Keyboard Alternatives to the Mouse

You can use the keyboard to perform many of the same functions you can perform with a mouse. This section lists these keyboard alternatives.

In general, to use a keyboard alternative, the *focus* must be in the screen region where you want the action to take place. The focus is generally indicated by the *location cursor*, which is a heavy line around the region.

General keyboard alternatives are listed below.

Key Name	Description
Tab	Use the Tab key to move the location cursor from field to field within a window or dialog box. The buttons in a window or box constitute one field. The location cursor highlights one of the buttons when you tab to this field.
Shift-Tab	Use the Shift-Tab key to perform the same function as Tab, but move through the fields in the opposite direction.
Return	Use the Return key to choose a highlighted choice in a menu, or to perform the action associated with a highlighted button in a window or dialog box.
Arrow keys	Use the up, down, left, and right arrow keys to move within a field. For example, when the location cursor highlights a list, you can use the up and down arrow keys to move through the choices in the list. In some windows that contain text, pressing the Control key along with an up or down arrow key scrolls the text one-half page.
F1	Use the F1 key instead of the Help button to obtain help about a window or dialog box.
F10	Use the F10 key to move the location cursor to the menu bar.
Meta	Use the Meta key along with the underlined character in the desired menu item to display a menu or dialog box (equivalent to clicking on the item with the mouse). The Meta key has different names on different keyboards; on some it is the Left or Right key.
Ctrl-c	Use the Ctrl-c key combination to interrupt command execution.
Esc	Use the Esc key instead of the Close or Cancel button to close the window or dialog box in which the mouse pointer is currently located.

TABLE 2-1 General Keyboard Alternatives

The following keys and key combinations work on the command line and in *text-entry boxes*—that is, fields in a dialog box or window where you can enter or edit text:

Key Name	Description
`Back Space`	Deletes the character to the left of the I-beam cursor.
`Delete`	Same as Back Space.
`Ctrl-a`	Moves to the beginning of the line.
`Ctrl-b`	Moves back one character.
`Ctrl-d`	Deletes the character to the right of the I-beam cursor.
`Ctrl-e`	Moves to the end of the line.
`Ctrl-f`	Moves forward one character.
`Ctrl-k`	Deletes to the end of the line.
`Ctrl-u`	Deletes to the beginning of the line.

TABLE 2-2 Text-entry Keyboard Alternatives

In addition, you can use *keyboard accelerators* to perform actions from the menu bar; see Section 2.4.

2.3.3 Issuing Commands

You can issue commands in Prism from the command line in the command window. Most commands duplicate functions you can perform from the menu bar; it's up to you whether you use the command or the corresponding menu selection. Some functions are only available via commands. See the *Prism Reference Manual* for complete information about Prism commands. Section 2.7 describes how to use the command window.

Many commands have the same syntax and perform the same action in both Prism and the Solaris debugger `dbx`. There are differences, however; you should check the reference description of a command before using it.

2.4 Using the Menu Bar

The menu bar is the line of titles across the top of the main window of Prism.

Each title is associated with a pulldown menu, from which you can perform actions within Prism.

2.4.0.1 Keyboard Accelerators

A keyboard accelerator is a shortcut that lets you choose a frequently used menu item without displaying its pulldown menu. Keyboard accelerators consist of the `Control` key plus a function key; you press both at the same time to perform the action. The keyboard accelerator for a menu selection is displayed next to the name of the selection; if nothing is displayed, there is no accelerator for the selection.

The keyboard accelerators (on a Sun keyboard) are

Accelerator	Function
Ctrl-F1	Run
Ctrl-F2	Continue
Ctrl-F3	Interrupt
Ctrl-F4	Step
Ctrl-F5	Next
Ctrl-F6	Where
Ctrl-F7	Up
Ctrl-F8	Down

TABLE 2-3 Sun Keyboard Accelerators

2.5 Using the Source Window

2.5.1 The Source Window

The source window displays the source code for the executable program loaded into Prism. (Chapter 3 describes how to load a program into Prism, and how to display the different source files that make up the program.) When you execute the program, and execution then stops for any reason, the source window updates to show the code being executed at the stopping place. The `Source File:` field at the top of the source window lists the file name of the file displayed in the window.

The source window is a separate pane within the main Prism window. You can resize it by dragging the small resize box at the lower right of the window. If you change its size, the new size is saved when you leave Prism.

You cannot edit the source code displayed in the source window. To edit source code within Prism, you must call up an editor; see Chapter 7.

2.5.1.1 Moving through the Source Code

As mentioned above, you can move through a source file displayed in the source window by using the scroll bar on the right side of the window. You can also use the up and down arrow keys to scroll a line at a time, or press the Control key along with the arrow key to move half a page at a time. To return to the current execution point, type Ctrl-x in the source window.

To search for a text string in the current source file, issue the `/string` or `?string` command in the command window. The `/string` command searches forward in the file for the string that you specify and repositions the file at the first occurrence it finds. The `?string` command searches backward in the file for the string that you specify.

You can display different files by choosing the File or Func selection from the File menu; see Section 3.5. You can also move *between* files. Prism keeps a list of the files you have displayed. With the mouse pointer in the source window, do this to move through the list:

- To display the previous file in the list, click the middle mouse button while pressing the left button. You are returned to the location at which you left the file.
- To display the next file in the list, click the right mouse button while pressing the left button.

2.5.1.2 Selecting Text

You can select text in the source window by dragging over it with the mouse; the text is then highlighted. Or double-click with the mouse pointer pointing to a word to select just that word. Left-click anywhere in the source window to “deselect” selected text.

Right-click in the source window to display a menu that includes actions to perform on the selected text, see FIGURE 2-1. For example, select Print to display a visualizer containing the value(s) of the selected variable or expression at the current point of execution. (See Chapter 5 for a discussion of visualizers and printing.) To close the popup menu, right-click anywhere else in the main Prism window

FIGURE 2-1 Popup Menu in Source Window

You can display the definition of a function by pressing the Shift key while selecting the name of the function in the source window. This is equivalent to choosing the Func selection from the File menu and selecting the name of the function from the list; see Chapter 3. Do not include the arguments to the function, just the function name.

▼ Splitting the Source Window

You can split the source window to simultaneously display the source code and assembly code of the loaded program. Follow these steps to split the source window:

1. **First load a program, as described in Chapter 3.**
2. **Right-click in the source window to display the popup menu.**
3. **Click on Show source pane in the popup menu.**

   This displays another menu.

4. **Choose Show .s source from it.**

   This causes the assembly code for your program to be displayed in the bottom pane of the window, as shown in FIGURE 2-2.

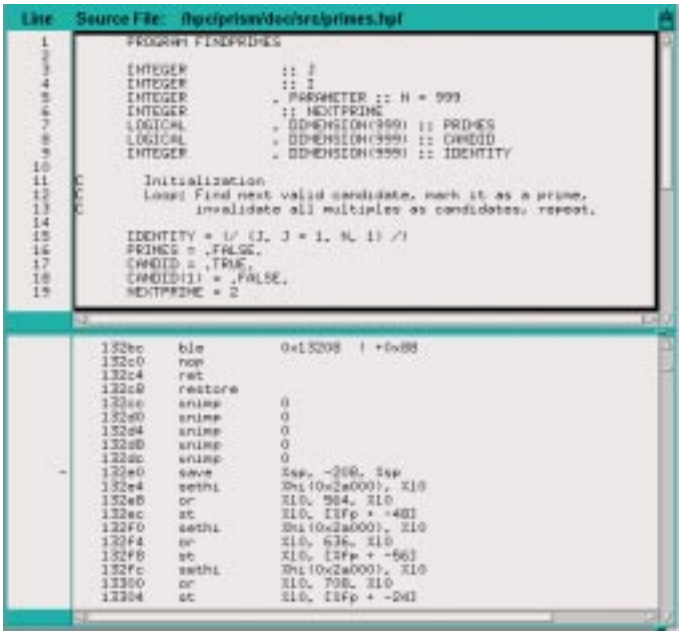

FIGURE 2-2 Split Source Window

When you split the source window, the top pane is highlighted; it is the *master pane*. Left-click in the slave pane to make it the master. If you scroll through the master, the slave pane scrolls to the corresponding place as well. Scrolling through the slave does not cause the master to scroll.

To return to a single source window, right-click in the pane you want to get rid of, and choose Hide this source pane from the popup menu.

2.6 Using the Line-Number Region

The line-number region shows the line numbers associated with the source code displayed in the source window. FIGURE 2-3 shows a portion of a line-number region, with a breakpoint set.

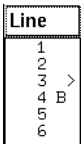

FIGURE 2-3 Line Number Region

The > symbol in the line-number region in FIGURE 2-3 is the *execution pointer*. When the program is being executed, the execution pointer points to the next line to be executed. If you move elsewhere in the source code, typing Ctrl-x returns to the current execution point.

A B appears in the line-number region next to every line at which execution is to stop. You can set simple breakpoints directly in the line-number region; all methods for setting breakpoints are described in Section 4.3.

A T appears in the line-number region next to a line for which Prism is tracing execution. See Section 4.4 to learn how to trace program execution.

Shift-click on B or T in the line-number region to display the *event* associated with the breakpoint or tracepoint. See Section 4.1 for a discussion of events.

The display of breakpoints and tracepoints in the line-number region is slightly more complicated in MP Prism; see Section 10.6.1.

There are two other symbols you will see in the line-number region:

- The - symbol is the *scope pointer*; it indicates the current source position (that is, the scope). Prism uses the current source position to interpret names of variables. When you scroll through source code, the scope pointer moves to the middle line of the code that is displayed. Various Prism commands also change the position of the scope pointer.
- The * symbol is used when the current source position is the same as the current execution point; this happens whenever execution stops.

If you right-click in the line-number window, you display the source-window popup menu discussed in the previous section. Right-click anywhere in the main Prism window to close this menu.

2.7 Using the Command Window

The command window is the area at the bottom of the main Prism window in which you type commands and receive Prism output.

The command window consists of two boxes: the command line, at the bottom, and the history region, above it. FIGURE 2-4 shows a command window, with a command on the command line and messages in the history region.

FIGURE 2-4 Command Window With History Region

The command window is a separate pane within the main Prism window. You can resize this window (using the resize box at the top right of the window) and scroll through it. If you don't intend to issue commands in the command window, you may want to make this window smaller, so that you can display more code in the source window. If you use the command window frequently, you may want to make it bigger. If you change the size of the window, the new size is saved when you leave Prism.

Use the `set $history` command, as described below, to specify the maximum number of lines that Prism is to retain in the history region; the default is 10,000. For example,

```
set $history = 2000
```

reduces the number of lines to 2000.

Prism uses up memory in maintaining a large history region. A smaller history region, therefore, may improve performance and prevent Prism from running out of memory.

2.7.1 Using the Command Line

You type commands on the command line at the bottom of the command window. You can type in this box whenever it is highlighted and an I-shaped cursor, called an *I-beam*, appears in it. See Section 2.3.2 for a list of keystrokes you can use in editing the command line. Press `Return` to issue the command. Type `Ctrl-c` to interrupt execution of a command (or choose the `Interrupt` selection from the `Execute` menu).

You can issue multiple commands on the Prism command line; separate them with a semicolon (;). One exception: If a command takes a file name as an argument, you cannot follow it with a semicolon, because Prism can't tell if the semicolon is part of the file name.

Prism keeps the commands that you issue in a buffer. Type Ctrl-p to display the previous command in this buffer. Type Ctrl-n to display the next command in the buffer. You can then edit the command and issue it in the usual way.

During long-running commands (for example, when you have issued the `run` command to start a program executing), you may still be able to execute other commands. If you issue a command that requires that the current command complete execution, you receive a warning message and Prism waits for the command to complete.

2.7.2 Using the History Region

Commands that you issue on the command line are echoed in the history region, above the command line. Prism's response appears beneath the echoed command. Prism also displays other messages in this area, as well as command output that you specify to go to the command window. Use the scroll bar at the right of this box to move through the display.

You can select text in the history region, using one of these methods:

- Double-click to select the word to which the mouse pointer is pointing.
- Triple-click to select the line on which the mouse pointer is located.
- Press the left mouse button and drag the mouse over the text to select it.

You can then paste the selected text into other text areas within Prism by clicking the middle mouse button.

To re-execute a command, triple-click on a line in the history region to select it, then click the middle mouse button with the mouse pointer still in the history region. If you middle-click with the mouse pointer on the command line, the selected text appears on the command line but is not executed. This gives you a way to edit the text before executing it.

2.7.3 Redirecting Output

You can redirect the output of most Prism commands to a file by including an "at" sign (@) followed by the name of the file on the command line. For example,

```
where @ where.output
```

puts the output of a `where` command (a stack trace) into the file `where.output`, in your current working directory within Prism.

You can also redirect output of a command to a window by using the syntax `on` *window*, where *window* can be:

- `command` (abbreviated `com`). This sends output to the command window; this is the default.
- `dedicated` (abbreviated `ded`). This sends output to a window dedicated to output for this command. If you subsequently issue the same command (no matter what its arguments are) and specify that output is to be sent to the dedicated window, this window will be updated. For example,

  ```
  list on ded
  ```

  displays the output of the `list` command in a dedicated window. (Some commands that have equivalent menu selections display their output in the standard window for the menu selection.)
- `snapshot` (abbreviated `sna`). This creates a window that provides a snapshot of the output. If you subsequently issue the same command and specify that output is to be sent to the snapshot window, Prism creates a separate window for the new output. The time each window was created is shown in its title. Snapshot windows let you save and compare outputs.

You can also make up your own name for the window; the name appears in the title of the window. This is useful if you want a particular label for a window. For example, if you were doing a stack trace at line 22, you could issue this command:

```
where on line22
```

to label the window with the location of the stack trace.

The commands whose output you cannot redirect are `run`, `edit`, `make`, and `sh`.

Note – Although the `run` command cannot be redirected using `on` or `@`, run can be redirected using > and other shell redirections.

2.7.4 Logging Commands and Output

As mentioned in Section 2.2.2, you can specify on the Prism command line the name of a file to which commands and output are to be logged. You can also do this from within Prism, by issuing the `log` command.

Use the `log` command to log Prism commands and output to a file. The log file will be located in the current directory. This can be helpful in saving a record of a Prism session. For example,

```
log @ prism.log
```

logs output to the file `prism.log`. Use `@@` instead of `@` to append the log to an already existing file. Issue the command

```
log off
```

to turn off logging.

You can use the `log` command along with the `source` command to replay a session in Prism; see the next section. If you want to do this, you must edit the log file to remove Prism output.

2.7.5 Executing Commands From a File

As mentioned in Section 2.2.2, you can specify on the Prism command line the name of a file from which commands are to be read in and executed. You can also do this from within Prism by issuing the `source` command.

Using the `source` command lets you rerun a session you saved via the `log` command. You might also use `source` if, for example, your program has a long argument list that you don't want to retype constantly.

For example,

```
source prism.cmds
```

reads in the commands in the file `prism.cmds`. They are executed as if you had actually typed them in the command window. When reading the file, Prism interprets lines beginning with a pound sign (#) as comments.

The `.prisminit` file is a special file of commands; if it exists, Prism executes this file automatically when it starts up. See Section 9.5 for more information.

2.8 Writing Expressions in Prism

While working in Prism, there are circumstances in which you may want to write expressions that Prism will evaluate. For example, you can print or display expressions, and you can specify an expression as a condition under which an action is to take place. You can write these expressions in the language of the program you are working on. This section discusses additional aspects of writing expressions.

2.8.1 How Prism Chooses the Correct Variable or Procedure

Multiple variables and procedures can have the same name in a program. This can be a problem when you specify a variable or procedure in an expression. To determine which variable or procedure you mean, Prism tries to *resolve its name* by using these rules:

- It first tries to resolve the name using the scope of the current function. For example, if you use the name `x` and there is a variable named `x` in the current function or the current file, Prism uses that `x`. The current function is ordinarily the function at the program's current stopping point, but you can change this. See Section 3.5.
- If this fails to resolve the name, Prism goes up the call stack and tries to find the name in the caller of the current function, then its caller, and so on.
- If the name is not found in the call stack, Prism arbitrarily chooses one of the variables or procedures with the name in the source code. When Prism prints out the information, it adds a message of the form "[using qualified name]". Qualified names are discussed below.

Issue the `which` command to find out which variable or procedure Prism would choose; the command displays the fully qualified name, as described below.

2.8.1.1 Using Qualified Names

You can override Prism's procedure for resolving names by *qualifying* the name.

A fully qualified name starts with a back-quotation mark (`). The symbol farthest to the left in the name is the file, followed optionally by the procedure, followed by the variable name. Each is preceded by a backquote (`). Thus,

```
`foo`a
```

specifies the variable *a* in file `foo`. (Note that you drop the extension in the filename.) And

```
`foo`foo`a
```

specifies the *a* in the procedure *foo* in the file `foo`.

Partially qualified names do not begin with `‘`, but have a `‘` in them. For example,

```
foo`a
```

In this case, Prism looks up the name farthest to the left first and picks the innermost symbol with that name that is visible from your current location.

Use the `whereis` command to display a list of all the fully qualified names that match the identifier you specify.

Prism assigns its own names (for example, `$b1`) to local blocks of C code. This disambiguates variable names, in case you reuse a variable name in more than one of these local blocks.

Prism attempts to be case-insensitive in interpreting names, but will use case to resolve ambiguities.

2.8.2 Using Fortran Intrinsic Functions in Expressions

Prism supports the use of a subset of Fortran and HPF intrinsic functions in writing expressions; the intrinsics work for all languages that Prism supports, except as noted below.

The intrinsics, along with the supported arguments, are

- **ALL(***logical array***)** – Determines whether all elements are true in a logical array. Works for Fortran only.
- **ANY(***logical array***)** – Determines whether any elements are true in a logical array. Works for Fortran only.
- **CMPLX(***numeric-arg, numeric-arg***)** – Converts the arguments to a complex number. If the intrinsic is applied to Fortran variables, the second argument must not be of type complex or double-precision complex.
- **COUNT(***logical array***)** – Counts the number of true elements in a logical array. Works for Fortran only.
- **DSIZE(***array***)** – Counts the total number of elements in the array.
- **ILEN(***I***)** – Returns one less than the length, in bits, of the two's-complement representation of an integer. If *I* is nonnegative, **ILEN(***I***)** has the value log2(*I* + 1); if *I* is negative, **ILEN(***I***)** has the value log2(-*I*).
- **IMAG(***complex number***)** – Returns the imaginary part of a complex number. Works for Fortran only.
- **MAXVAL(***array***)** – Computes the maximum value of all elements of a numeric array.
- **MINVAL(***array***)** – Computes the minimum value of all elements of a numeric array.
- **PRESENT(***arg***)** – Determines if the specified argument exists in the context of the current procedure call. Works for Sun HPF only.
- **PRODUCT(***array***)** – Computes the product of all elements of a numeric array.
- **RANK(***scalar or array***)** – Returns the rank of the array or scalar.

- **REAL(***numeric argument***)** – Converts an argument to real type. Works for Fortran only.
- **SUM(***array***)** – Computes the sum of all elements of a numeric array.

The intrinsics can be either upper- or lowercase.

2.8.3 Using C Arrays in Expressions

Prism handles arrays slightly differently from the way C handles them.

In a C program, if you have the declaration

```
int a[10];
```

and you use *a* in an expression, the type of *a* converts from "array of ints" to "pointer to int". Following the rules of C, therefore, a Prism command like

```
print a + 2
```

should print a hexadecimal pointer value. Instead, it prints two more than each element of *a* (that is, `a[0] + 2`, `a[1] + 2`, etc.). This allows you to do array operations and use visualizers on C arrays in Prism. (The `print` command and visualizers are discussed in Chapter 5.)

To get the C behavior, issue the command as follows:

```
print &a + 2
```

2.8.4 Using Array-Section Syntax

2.8.4.1 In C Arrays

You can use Fortran 90 array-section syntax when specifying C arrays. This syntax is useful, for example, if you want to print the values of only a subset of the elements of an array. The syntax is:

(*lower-bound: upper-bound: stride*)

where

- **lower-bound** – The lowest-numbered element you choose along a dimension; it defaults to 0.
- **upper-bound** – The highest-numbered element you choose along the dimension; it defaults to the highest-numbered element for the dimension.
- **stride** – The increment by which elements are chosen between the lower bound and upper bound; it defaults to 1.

You must enclose the values in parentheses (rather than brackets), as in Fortran. If your array is multidimensional, you must separate the dimension specifications with commas within the parentheses, once again as in Fortran.

For example, if you have this array:

```
int a[10][20];
```

then you can issue this command in Prism to print the values of elements 2-4 of the first dimension and 2-10 of the second dimension:

```
print a(2:4,2:10)
```

2.8.5 Hints for Detecting NaNs and Infinities

Prism provides expressions that you can use to detect NaNs (values that are "not a number") and infinities in your data. These expressions derive from the way NaNs and infinities are defined in the IEEE standard for floating-point arithmetic.

To find out if `x` is a NaN, use the expression:

```
(x .ne. x)
```

For example, if `x` is an array, issue the command

```
where (x .ne. x) print x
```

to print only the elements of `x` that are NaNs. (The `print` command is discussed in Chapter 5.)

Also, note that if there are NaNs in an array, the mean of the values in the array will be a NaN. (The mean is available via the `Statistics` selection in the `Options` menu of a visualizer—see Chapter 5.)

To find out if `x` is an infinity, use the expression:

```
(x * 0.0 .ne. 0.0)
```

2.9 Using Sun HPF Generic Procedures

You can use Sun HPF generic procedures in any Prism command or dialog box that asks for a procedure. If you do so, Prism will prompt you for the name(s) of the specific procedure(s) you want to use.

For example, you use the syntax `stop in` *procedure* to set a breakpoint in a procedure. If you use this syntax for a generic procedure, in graphical Prism a dialog box like the one shown in FIGURE 2-5 would be displayed.

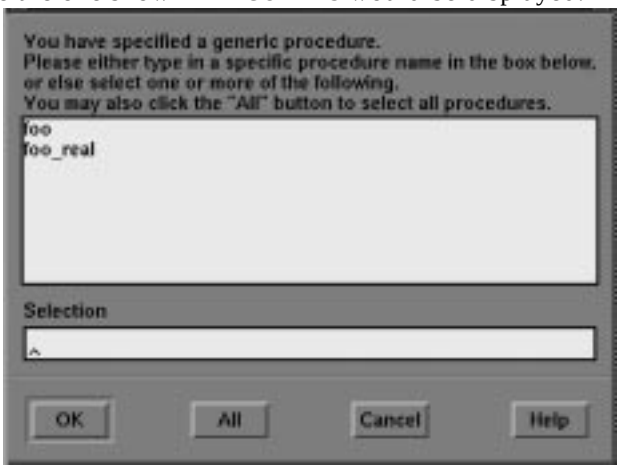

FIGURE 2-5 Generic Procedure Dialog Box

Commands-only Prism would prompt you as in this example:

```
(prism) stop in foo
The procedure ”foo” is a generic procedure.
Please either specify a specific procedure name, or type
”Return” to be prompted by a procedure menu that will contain
the 4 legal procedures in this context.
>
```

If you press the Return key, you would see a menu like this:

```
Please select one or more of the following:
 0) Cancel
 1) All
 2) foo(glorf.f:64)
 3) foo_real(glorf.f:121)
 4) foo_bar(glorf.f:189)
 5) foo_glorf(glorf.f:244)
>
```

If you choose 0 or press Return, the command is cancelled. If you choose other numbers, Prism sets the breakpoint(s) in the specified procedure(s). For example,

```
> 3 4
(1) stop in foo_bar
(2) stop in foo_glorf
(prism)
```

2.10 Issuing Solaris Commands

You can issue Solaris commands from within Prism.

- **From the menu bar** – Choose the Shell selection from the Utilities menu. Prism creates a Solaris shell. The shell is independent of Prism; you can issue Solaris commands from it just as you would from any Solaris shell. The type of shell that is created depends on the setting of your `SHELL` environment variable.
- **From the command window** – Issue the `sh` command on the command line. With no arguments, it creates a Solaris shell. If you include a Solaris command line as an argument, the command is executed, and the results are displayed in the history region.

Some Solaris commands have Prism equivalents, as described below.

2.10.1 Changing the Current Working Directory

By default your current working directory within Prism is the directory from which you started Prism. To change this working directory, use the `cd` command, just as you would in the Solaris environment. For example,

```
cd /sistare/bin
```

changes your working directory to `/sistare/bin`.

```
cd ..
```

changes your working directory to the parent of the current working directory. Issue `cd` with no arguments to change the current working directory to your login directory.

Prism interprets all relative file names with respect to the current working directory. Prism also uses the current working directory to determine which files to show in file-selection dialog boxes.

To find out what your current working directory is, issue the `pwd` command, just as you would in the Solaris environment.

2.10.2 Setting and Displaying Environment Variables

You can set, unset, and display the settings of environment variables from within Prism, just as you do in the Solaris system.

Use the `setenv` command to set an environment variable. For example,

`setenv EDITOR emacs`

sets your `EDITOR` environment variable to `emacs`.

Use the `unsetenv` command to remove the setting of an environment variable. For example,

`unsetenv EDITOR`

removes the setting of the `EDITOR` environment variable.

Use the `printenv` command to print the setting of an individual environment variable. For example,

`printenv EDITOR`

prints the current setting of the `EDITOR` environment variable. Or, issue `printenv` or `setenv` with no arguments to print the settings of all your environment variables.

2.11 Leaving Prism

To leave Prism:

- **From the menu bar** – Choose the Quit selection from the File menu. You are asked if you are sure you want to quit. Click on OK if you're sure; otherwise, click on Cancel or press the Esc key to stay in Prism.
- **From the command window** – Issue the `quit` command on the command line. (You aren't asked if you're sure you want to quit.)

If you have created subprocesses while in Prism (for example, a Solaris shell), Prism displays this message before exiting:

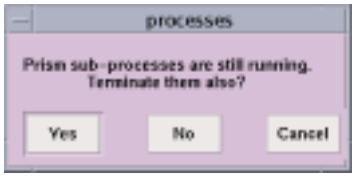

FIGURE 2-6 Sub-process Warning

Choose Yes (the default) to leave Prism and terminate the subprocesses. Choose No to leave Prism without terminating the subprocesses. Choose Cancel to stay in Prism.

CHAPTER 3

Loading and Executing a Program

This chapter describes how to load and run programs within Prism.

See the following sections to learn

- How to load a program into Prism — *Section* 3.1.
- How to associate a core file with a loaded program — *Section* 3.2.
- How to attach to and detach from a running process — *Section* 3.3.
- How to execute a program — *Section* 3.4.
- How to change the current file and the current function — *Section* 3.5.
- How to specify the directories to be searched for source files — *Section* 3.6.

For this chapter, you should already have an executable program that you want to run within Prism. You can also develop a new program by calling up an editor within Prism; see Chapter 7.

3.1 Loading a Program

Before you can execute or debug a program in Prism, you must first load the program into Prism. Only one program can be loaded at a time.

As described in Chapter 2, you can load a program into Prism by specifying its name as an argument to the `prism` command. If you don't use this method, you can load a program once you are in Prism by using one of the methods discussed next.

3.1.1 From the Menu Bar

Choose the Load selection from the File menu. (It is also by default in the tear-off region.) A dialog box appears, as shown in FIGURE 3-1.

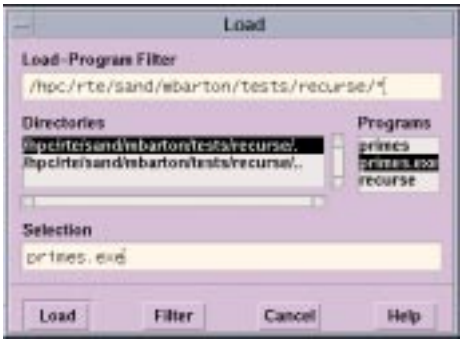

FIGURE 3-1 Load Program Filter

To load a program, you can simply double-click on its name, if the name appears in the Programs scrollable list. Or, you can put its path name in the Selection box, then click on Load. To put the file's path name in the Selection box, you can either type it directly in the box or click on its name in the Programs list. The Programs list contains the executable programs in your current working directory; see Section 2.10.1.

Use the Load-Program Filter box to control the display of file names in the Programs list; the box uses standard Solaris filters. For example, you can click on a directory in the Directories list if you want to change to that directory. But the Programs list does not update automatically to show the programs in the new directory. Instead, the filter changes to *directory-name*/*, indicating that all files in *directory-name* are to be displayed. Click on Filter to display the file names of the programs. Or simply double-click on the directory name in the Directories list to display the programs in the directory.

If you want to use a different filter, you can edit the Load-Program Filter box directly. For example, change it to *directory-name*/*prog** to display only programs beginning with *prog*.

Click on Cancel or press the Esc key if you decide not to load a program.

3.1.2 From the Command Window

Issue the `load` command on the command line, with the name of the executable program as its argument. For example,

```
load myprogram
```

The program you specify is loaded.

3.1.3 What Happens When You Load a Program

Once a program is successfully loaded:

- The program's name appears in the Program field in the main window.
- The source file containing the program's main function appears in the source window.
- The Load dialog box disappears (if you loaded the program using this box).
- The status region displays the message `not started`.

You can now issue commands to execute and debug this program.

If Prism can't find the source file, it displays a warning message in the command window. Choose the Use selection from the File menu to specify other directories in which Prism is to search; see Section 3.6.

3.1.4 Loading Subsequent Programs

Only one program can be loaded at a time. If you have a program loaded and you want to switch to a new program, simply load the new program; the previously loaded program is automatically unloaded. If you want to start fresh with the current program, issue the `reload` command with no arguments; the currently loaded program is reloaded into Prism.

3.2 Associating a Core File With a Loaded Program

As mentioned in Chapter 2, you can have Prism associate a core file with a program by specifying its name after the name of the program on the `prism` command line.

You can also do this by loading the program and then issuing the `core` command, specifying the name of the corresponding core file as its argument.

In either case, Prism reports the error that caused the core dump and loads the program with a *stopped* status at the location where the error occurred. You can then work with the program within Prism. You can, for example, examine the stack and print the values of variables. You cannot, however, continue execution from the current location.

3.3 Attaching To and Detaching From a Running Process

See Section 10.2.4 for information on attaching to a running message-passing process.

As described in Section 2.2.2, you can load a running process into Prism by specifying the name of the executable program and the process ID of the corresponding running process on the Prism command line.

You can also attach to a running process from within Prism; note that, as with the procedure described above, *the process must be running on the same node as Prism* (unless you are using MP Prism).

▼ To attach from within Prism,

1. **Find out the process's process ID by issuing the Solaris command `ps` (or, if you are using MP Prism, find the process's task ID by issuing the Sun HPC command `tmps`).**
2. **Load the executable program for the process into Prism.**
3. **Issue the `attach` command on the Prism command line, using the process's process `ID` (or task `ID` in MP Prism) as the argument.**

   With either method of attaching to the process, the process is interrupted; a message is displayed in the command window giving its current location, and its status is stopped. You can then work with the program in Prism as you normally would. The only difference in behavior is that it does not do its I/O in a special Xterm window; see Section 3.4.2.

To detach from a running process, issue the command `detach` from the Prism command line. The process continues to run in the background from the point at which it was stopped in Prism; it is no longer under the control of Prism. Note that you can detach any process in Prism via the `detach` command, not just processes that you have explicitly attached.

Note – Use the `kill` command to terminate the process or task (rather than releasing it to run in the background) currently running within Prism.

3.4 Executing a Program

To execute a program, you must first load it, as described in Section 3.1. Once you start the program running, you can step through it, and interrupt and continue execution.

See Section 10.4 for information on executing a program in MP Prism.

3.4.1 Running a Program

To run a program:

- **From the menu bar** – If you have no command-line arguments you want to specify, choose the Run selection from the Execute menu; execution starts immediately. (The Run selection by default is in the tear-off region.)

  If you have command-line arguments, choose the Run (args) selection from the Execute menu. A dialog box is displayed, in which you can specify any command-line arguments for the program; see FIGURE 3-2. If you have more arguments than fit in the input box, they scroll to the left. Click on the Run button to start execution.

FIGURE 3-2 Run (args) Dialog Box

- **From the command window –** Issue the `run` command, including any arguments to the program on the command line. You can abbreviate the command to `r`. If you have already run the program, you can issue the `rerun` command to run it

again, using the same argument list you previously passed to the program. In both cases, you can redirect input or output using **<** or **>** in the standard Solaris manner.

When the program starts executing, the status region displays the message `running`.

You can continue to interact with Prism while a program is running, but many features will be unavailable. Unavailable selections are grayed out in menus. If you issue a command that cannot be executed while the program is running, it is queued until the program stops.

3.4.2 Program I/O

Prism by default creates a new window for a program's I/O. This window persists across multiple executions and program loads, giving you a complete history of your program's input and output. If you prefer, you can display I/O in the Xterm from which you invoked Prism; see Section 9.3.

3.4.3 Stepping Through a Program

You must begin execution by choosing Run or Run (args) (or issuing `run` from the command line). If execution stops before the program finishes (for example, because you have set a breakpoint), you can then step through the program, as described in this section. To step through the entire program, set a breakpoint at the first executable line, and then run to it. (See Section 4.3 for information on setting breakpoints.)

From the menu bar:

- Choose the Step selection from the Execute menu to execute the next line of the program. (It is by default in the tear-off region.) Step steps into any functions called on that line.
- Choose the Next selection from the Execute menu to execute the next statement of the program. (It is also by default in the tear-off region.) Next steps over any function called in the line, considering the function to be a single statement.
- Choose the Stepout selection from the Execute menu to execute the current function, then return to its caller.

The execution pointer moves to indicate the next line to be executed.

From the command window: Issue the `step`, `next`, or `stepout` command from the command line to perform the same action as the equivalent menu-bar selection; `return` is a synonym for `stepout`. In addition, you can specify the number of lines

to be executed as an argument to `step` and `next`, and you can specify as an argument to `stepout` the number of levels of the call stack that you want to step out.

The `stepi` and `nexti` commands are also available for stepping by machine instruction. The address and instruction are displayed in the command window.

If execution takes considerable time—for example, because Next calls a long-running function—the status changes to running. You can use Prism, but many commands will be unavailable. Unavailable selections are grayed out in menus.

3.4.4 Interrupting and Continuing Execution

To interrupt execution, choose `Interrupt` from the `Execute` menu or type `Ctrl-c`. The status changes to interrupted, and the source window updates to show the point at which execution stopped.

To continue execution after a program has been interrupted, choose `Continue` from the `Execute` menu, or issue the `cont` command from the command line. (Or you can step through the program, as described above.)

Continue and Interrupt are available by default in the tear-off region.

3.4.5 Status Messages

Prism displays the status messages listed in TABLE 3-1 before, during, and after the execution of a program.

TABLE 3-1 Status Messages

Messag	Meaning
`error`	Prism has encountered an internal error.
`connected`	Prism has connected to other nodes to work on a data parallel or message-passing program.
`connecting`	Prism is connecting to other nodes in order to work on a data parallel or message-passing program.
`initial`	Prism is starting up without a program loaded.
`interrupted`	The program has been interrupted.
`loading`	Prism is loading a program.
`not started`	The program is loaded but not yet started.

TABLE 3-1 Status Messages

Messag	Meaning
`running`	The program is running.
`stopped`	The program has stopped at a breakpoint or signal.
`terminated`	The program has run to completion and the process has gone away.

3.5 Choosing the Current File and Function

Prism uses the concepts of *current file* and *current function.*

The current file is the source file currently displayed in the source window. The current function is the function or procedure displayed in the source window. You might change the current file or function if, for example, you want to set a breakpoint in a file that is not currently displayed in the source window, and you don't know the line number at which to set the breakpoint.

In addition, changing the current file and current function changes the scope used by Prism for commands that refer to line numbers without specifying a file, as well as the scope used by Prism in identifying variables; see Section 2.8.1 for a discussion of how Prism identifies variables. The scope pointer (-) in the line-number region moves to the current file or current function to indicate the beginning of the new scope.

To change the current file:

- **From the menu bar** – Choose the File selection from the File menu. A window is displayed, listing in alphabetical order the source files that make up the loaded program. Click on one, and it appears in the Selection box; click on OK, and the source window updates to display the file. Or simply double-click, rapidly, on the source file. You can also edit the file name in the Selection box.

Note – The File window displays only files compiled with the `-g` switch.

FIGURE 3-3 File Window

- **From the command window** – Issue the `file` command, with the name of a file as its argument. The source window updates to display the file.

To change the current function or procedure:

- **From the menu bar** – Choose the Func selection from the File menu. A window is displayed, listing the functions in the program in alphabetical order. (Fortran procedure names are converted to all lowercase.) Click on one, and it appears in the Selection box; click on OK, and the source window updates to display the function. Or simply double-click on the function name in the list. You can also edit the function name in the Selection box.

  By default, the Func window displays only functions in files compiled with the `-g` switch. To display all functions in the program, click on the Select All Functions button. The button then changes to Show -g Functions; click on it to return to displaying only the `-g` functions.
- **From the command window** – Issue the `func` command with the name of a function or subroutine as its argument. The source window updates to display the function.
- **From the source window** – Select the name of the function in the source window by dragging the mouse over it while pressing the Shift key. When you let go of the mouse button, the source window is updated to display the definition of this function.

Note – Do not include the arguments with the function, just its name.

Note that if the function you choose is in a different source file from the current file, changing to this function also has the effect of changing the current file.

3.6 Creating a Directory List for Source Files

If you have moved a source file, or if for some other reason Prism can't find it, you can explicitly add its directory to Prism's search path.

- **From the menu bar –** Choose the Use selection from the File menu. This displays a dialog box, as shown in FIGURE 3-4. To add a directory, type its path name in the Directory box, then click on Add. To remove a directory, click on it in the directory list; its path name appears in the Directory box; then click on Remove.

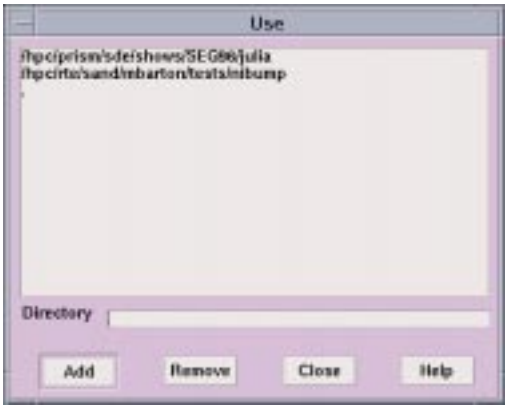

FIGURE 3-4 Use Dialog Box

- **From the command window –** Issue the `use` command on the command line. Specify a directory as an argument; the directory is added to the front of the search path. Issue `use` with no arguments to display the list of directories to be searched.

Note – No matter what the contents of your directory list are, Prism searches for the source file first in the directory in which the program was compiled.

CHAPTER 4

Debugging a Program

This chapter discusses how to debug programs in Prism. It also describes how to use *events* to control the execution of a program.

See the following sections to learn

- What events are — *Section* 4.1.
- How to use the event table — *Section* 4.2.
- How to set breakpoints — *Section* 4.3.
- How to trace program execution — *Section* 4.4.
- How to display and move through the call stack — *Section* 4.5.
- How to examine the contents of memory and registers — *Section* 4.6.

See Chapter 10 for additional information on debugging a program in MP Prism.

4.1 Overview of Events

A typical approach to debugging is to stop the execution of a program at different points so that you can perform various actions—for example, check the values of variables. You stop execution by setting a *breakpoint*. If you perform a *trace*, execution stops, then automatically continues.

Breakpoints and traces are *events*. You can specify before the execution of a program begins what events are to take place during execution. When an event occurs:

- The execution pointer moves to the current execution point.
- A message is printed in the command window.
- If you specified that an action was to accompany the event (for example, the printing of a variable's value), it is performed.

- If the event is a trace, execution then continues. If it is a breakpoint, execution does not resume until you explicitly order it to (for example, by choosing Continue from the Execute menu).

Prism provides various ways of creating these events—for example, by issuing commands, or by using the mouse in the source window. Section 4.3 describes how to create breakpoint events; Section 4.4 describes how to create trace events. Section 4.2 describes the *event table*, which provides a unified method for listing, creating, editing, and deleting events.

See Section 10.6.1 for a discussion of events in MP Prism.

You can define events so that they occur:

- *When the program reaches a certain point in its execution*—For example, at a specified line or function.
- *When the value of a variable changes*—For example, you can define an event that tells Prism to stop the program when *x* changes value. This kind of event is sometimes referred to as a *watchpoint*. It slows execution considerably, since Prism has to check the value of the variable after each statement is executed.
- *At every line or assembly-language instruction.*
- *Whenever a program is stopped*—For example, you can define an event that tells Prism to print the value of *x* whenever the program stops.

Such events are referred to as *triggering conditions*.

In addition, you can qualify an event as follows:

- *So that it occurs only if a specified condition is met*—For example, you can tell Prism to stop at line 25 if *x* is not equal to 1. Like watchpoints, this kind of event slows execution.
- *So that it occurs only after its triggering condition has been met a specified number of times*—For example, you can tell Prism to stop the tenth time that the program reaches the function `foo`.

You can include one or more Prism commands as actions that are to take place as part of the event. For example, using Prism commands, you can define an event that tells Prism to stop at line 25, print the value of `x`, and do a stack trace.

4.2 Using the Event Table

The event table provides a unified method for controlling the execution of a program. Creating an event in any of the ways discussed later in this chapter adds an event to the list in this table. You can also display the event table and use it to:

- Add new events
- Delete existing events
- Edit existing events

You display the event table by choosing the Event Table selection from the Events menu.

This section describes the general process of using the event table.

4.2.1 Description of the Event Table

FIGURE 4-1 shows the event table.

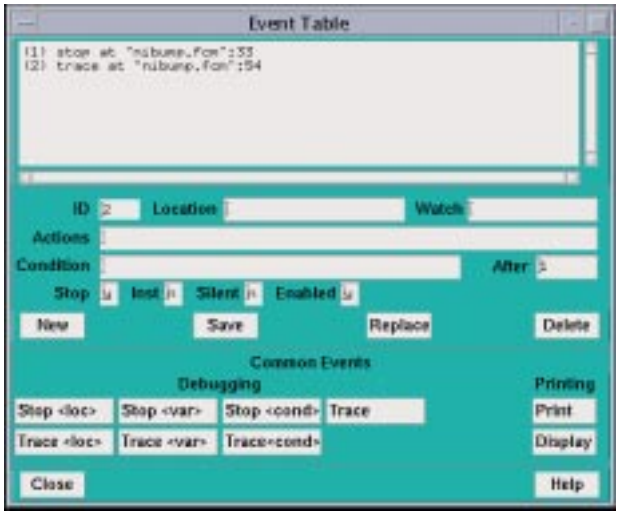

FIGURE 4-1 Event Table

The top area of the event table is the *event list*—a scrollable region in which events are listed. When you execute the program, Prism uses the events in this list to control execution. Each event is listed in a format in which you could type it as a command in the command window. It is prefaced by an ID number assigned by Prism. For example, in FIGURE 4-1, the events have been assigned the IDs 1 and 2.

The middle area of the event table is a series of fields that you fill in when editing or adding an event; only a subset of the fields is relevant to any one event. The fields are:

- **ID –** This is an identification number associated with the event. You cannot edit this field.
- **Location –** Use this field to specify the location in the program at which the event is to take place. Use the syntax *"filename":line-number* to identify the source file and the line within this file. If you just specify the line number, Prism uses the current file. There are also three keywords you can use in this field:
  - Use `eachline` to specify that the event is to take place at each line of the program; this is the default.
  - Use `eachinst` to specify that the event is to take place at each assembly-language instruction.
  - Use `stopped` to specify that the event is to take place whenever the program stops execution.
- **Watch –** Use this field to specify a variable or expression whose value(s) are to be watched; the event takes place if the value of the variable or expression changes. (If the variable is an array or a parallel variable, the event takes place if the value of any element changes.) This slows execution considerably.
- **Actions –** Use this field to specify the action(s) associated with the event. The actions can be most Prism commands; separate multiple commands with semicolons. (The commands that you can't include in the Actions field are `attach`, `core`, `detach`, `load`, `return`, `run`, and `step`.)
- **Condition –** Use this field to specify a logical condition that must be met if the event is to take place. The logical condition can be any language expression that evaluates to true or false. See Section 2.8 for more information about writing expressions in Prism. Specifying a condition slows execution considerably, unless you also specify a location at which the condition is to be checked.
- **After –** Use this field to specify how many times a triggering condition is to be met (for example, how often a program location is reached) before the event is to take place. The event table updates during execution to show the current count (that is, how many times are left for the triggering condition to be met before the event is to take place). Once the event takes place, the count is reset to the original value. The default setting is 1, and the event takes place each time the condition is met. See Section 4.1 for a discussion of triggering conditions.
- **Stop –** Use this field to specify whether or not the event is to halt execution of the program. Putting a `y` in this field creates a breakpoint event; putting an `n` in this field creates a trace event.
- **Inst –** Use this field to specify whether to display a disassembled assembly-language instruction when the event occurs.
- **Silent –** Use this field to specify whether or not the event is to cause a message to appear in the command window when it occurs.

- **Enabled –** Use this field to specify whether the event is enabled. Putting an n in this field disables the event; it still exists, but it does not affect program execution.

The buttons beneath these fields are for use in creating and deleting events, and are described below.

The area headed Common Events contains buttons that provide shortcuts for creating certain standard events.

Click on Close or press the Esc key to cancel the Event Table window.

4.2.2 Adding an Event

You can either add an event, editing field by field, or you can use the Common Events buttons to fill in some of the fields for you. You would add an event from scratch if it weren't similar to any of the categories covered by the Common Events buttons.

▼ To Add an Event

1. **Click on the New button; all values currently in the fields are cleared.**
2. **Fill in the relevant fields to create the event.**
3. **Click on the Save button to save the new event; it appears in the event list.**

▼ To Use the Common Events Buttons to Add an Event

1. **Click on the button for the event you want to add—for example, Print.**

   This fills in certain fields (for example, it puts print on dedicated in the Actions field) and highlights the field or fields that you need to fill in (for example, it highlights the Location field when you click on Print, because you have to specify a program location).

2. **Fill in the highlighted field(s). You can also edit other fields, if you like.**
3. **Click on Save to add the event to the event list.**

   Most of these Common Events buttons are also available as separate selections in the Events menu. This lets you add one of these events without having to display the entire event table. The menu selections, however, prompt you only for the field(s) you must fill in. You cannot edit other fields.

   Individual Common Events buttons are discussed throughout the remainder of this guide.

   You can also create a new event by editing an existing event; see Section 4.2.4.

4.2.3 Deleting an Existing Event

▼ To Delete an Existing Event, Using the Event Table

1. **Click on the line representing the event in the event list, or move to it with the up and down arrow keys.**

   This causes the components of the event to be displayed in the appropriate fields beneath the list.

2. **Click on the Delete button.**

   You can also choose the Delete selection from the Events menu to display the event table. You can then follow the procedure described above.

   Deleting a breakpoint at a program location also deletes the `B` in the line-number region at that location.

4.2.4 Editing an Existing Event

You can edit an existing event to change it, or to create a new event similar to it.

▼ To Edit an Existing Event

1. **Click on the line representing the event in the event list, or move to it with the up and down arrow keys.**

   This causes the components of the event to be displayed in the appropriate fields beneath the list.

2. **Edit these fields.**

   For example, you can change the Location field to specify a different location in the program.

3. **Click on Replace to save the newly edited event *in place of* the original version of the event.**

   Click on the Save button to save the new event *in addition to* the original version of the event; it is given a new ID and is added to the end of the event list. Clicking on Save is a quick way of creating a new event similar to an event you have already created.

4.2.5 Enabling and Disabling Events

You can disable and enable events. When you disable an event, Prism keeps it in the event list, but it no longer affects execution. You can subsequently enable it when you once again want it to affect execution. This can be more convenient than deleting events and then redefining them.

- **From the event table** – The event table has an `Enabled` field. By default, there is a *y* in this field, meaning that the event being defined or edited is enabled. Click on the field and change the *y* to an *n* to disable the event. The event remains in the event list, but is labeled `(disabled)`. You can then edit the event as described in Section 4.2.4 and change the field back to a *y* to enable the event once again.
- **From the command line** – Issue the `disable` command to disable an event. Use the event's ID as the argument. You can obtain this ID from the event list in the event table, or by issuing the `show events` command.

For example, this sequence of commands displays the event list, then disables an event, then redisplays the event list:

```
(prism) show events
(1) trace
(2) when stopped { print board }
(prism) disable 1
event 1 disabled
(prism) show events
(1) trace (disabled)
(2) when stopped { print board }
```

Issue the `enable` command to enable an event that has been disabled. Specify the ID of the disabled event as the argument.

4.2.6 Saving Events

Events that you create for a program are automatically maintained when you reload the same program during a Prism session. This saves you the effort of redefining these events each time you reload a program.

Note these points:

- Prism prints a warning message if it can't maintain an event—for example, because the event is supposed to occur at a source line that no longer exists. Obviously, changing the program can also change the meaning of events; a breakpoint set at line 32, for example, may still be a valid event, but it may not be the event you want if you have deleted lines earlier in the program.
- Disabled events become enabled when a program is reloaded.
- Events are deleted when you leave Prism.

▼ Executing Prism Commands From a File

To use Prism commands to save your events to a file, and then execute them from the file rather than individually

1. **Issue the** `show events` **command, which displays the event list.**

   Redirect the output to a file. For example,

   ```
   show events @ primes.events
   ```

   (See Section 2.7.3 for information on redirecting output.)

2. **Edit this file to remove the ID number at the beginning of each event.**

   This leaves you with a list of Prism commands.

3. **Issue the** `source` **command when you want to read in and execute the commands from the file.**

   For example,

   ```
   source primes.events
   ```

4.3 Setting Breakpoints

A *breakpoint* stops execution of a program when a specific location is reached, if a variable or expression changes its value, or if a certain condition is met. Breakpoints are events that Prism uses to control execution of a program. This section describes the methods available in Prism for setting a breakpoint.

You can set a breakpoint

- By using the line-number region
- By using the event table and the Events menu
- From the command window, by issuing the command `stop` or `when`

You'll probably find it most convenient to use the line-number region for setting simple breakpoints; however, the other two methods give you greater flexibility—for example, in setting up a condition under which the breakpoint is to take place.

In all cases, an event is added to the list in the event table. If you delete the breakpoint using any of the methods described in this section, the corresponding event is deleted from the event list. If you set a breakpoint at a program location, a `B` appears next to the line number in the line-number region.

4.3.1 Using the Line-Number Region

To use the line-number region to set a breakpoint, the line at which you want to stop execution must appear in the source window. If it doesn't, you can scroll through the source window (if the line is in the current file), or use the File or Func selection from the File menu to display the source file you are interested in.

▼ To Set a Breakpoint in the Line-Number Region

1. **Position the mouse pointer to the right of the line numbers; the pointer turns into a** B**.**
2. **Move the pointer next to the line at which you want to stop execution.**
3. **Left-click the mouse.**
4. **A** B **is displayed, indicating that a breakpoint has been set for that line.**

   A message appears in the command window confirming the breakpoint, and an event is added to the event list.

   The source line you choose must contain executable code; if it does not, you receive a warning in the command window, and no B appears where you clicked.

   Shift-click on the letter in the line-number region to display the complete event (or events) associated with it.

   See Section 2.6 for more information on the line-number region.

   See Section 10.6.1 for a discussion of the line-number region in MP Prism.

4.3.1.1 Deleting Breakpoints via the Line-Number Region

To delete the breakpoint, left-click on the B that represents the breakpoint you want to delete. The B disappears; a message appears in the command window, confirming the deletion.

4.3.1.2 What Happens in a Split Source Window

As described in Section 2.5.1, you can split the source window to display source code and the corresponding assembly code.

You can set a breakpoint in either pane of the split source window. The B appears in the line-number region of both panes, unless you set the breakpoint at an assembly code line for which there is no corresponding source line.

Deleting a breakpoint from one pane of the split source window deletes it from the other pane as well.

4.3.2 Using the Event Table and the Events Menu

To set a breakpoint, choose the Stop <loc> or Stop <var> selection from the Events menu. These choices are also available as Common Events buttons within the event table itself; see Section 4.2.2.

- Stop <loc> prompts for a location at which to stop the program. You can also specify a function or procedure; the program stops at the first line of the function or procedure.

FIGURE 4-2 Stop <loc> Dialog Box

- Stop <var> prompts for a variable name. The program stops when the variable's value changes. The variable can be an array, in which case execution stops any time any element of the array changes. This slows execution considerably.

In addition, Stop <cond> is available as a Common Events button. It prompts for a condition, which can be any expression that evaluates to true or false; see Section 2.8 for more information on expressions. The program stops when the condition is met. This slows execution considerably.

You can also use the event table to create combinations of these breakpoints; for example, you can create a breakpoint that stops at a location if a condition is met. In addition, you can use the Actions field of the event table to specify the Prism commands that are to be executed when execution stops.

4.3.2.1 Deleting Breakpoints via the Event Table

To delete a breakpoint, choose the Delete selection from the Events menu, or use the Delete button in the event table itself. See Section 4.2.3.

4.3.3 Using Commands

Issue the command `stop` (or `when`, which is an alias for `stop`) from the command line to set a breakpoint. The syntax of the `stop` command is also used by the `stopi`, `trace`, and `tracei` commands, which are discussed below. The general syntax for all the commands is:

command [*variable* | `at` *line* | `in` *func*] [`if` *expr*] [{*cmd*[; *cmd*...]}] [`after` *n*]

where

- *command* – As mentioned above, can be `stop`, `stopi`, `when`, `trace`, or `tracei`.
- *variable* – Is the name of a variable. The command is executed (in other words, the event takes place) if the value of the variable changes. If the variable is an array, an array section, or a parallel variable, the command is executed if the value of any element changes. This form of the command slows execution considerably. You cannot specify both a variable and a program location.
- *line* – Specifies the line number where the stop or trace is to be executed. If the line is not in the current file, use the format:

  `at "`*filename*`":`*line-number*

- *func* – Is the name of the function or procedure in which the stop or trace is to be executed.
- *expr* – Is any language expression that evaluates to true or false. This argument specifies the logical condition, if any, under which the stop or trace is to be executed. For example,

  ```
  if a .GT. 1
  ```

  This form of the command slows execution considerably, unless you combine it with the `at` *line* syntax. See Section 2.8 for more information on writing expressions in Prism.
- *cmd* – Is any Prism command (except `attach`, `core`, `detach`, `load`, `return`, `run`, or `step`). This argument specifies the actions, if any, that are to accompany the execution of the stop or trace. For example, `{print a}` prints the value of `a`. If you include multiple commands, separate them with semicolons.
- *n* – Is an integer that specifies how many times a triggering condition is to be reached before the stop or trace is executed; see Section 4.1 for a discussion of triggering conditions. This is referred to as an after count. The default is 1. Once the stop or trace is executed, the count is reset to its original value. Note that if there is both a condition and an after count, the condition is checked first.

The first option listed (specifying the location or the name of the variable) must come first on the command line; the other options, if you include them, can be in any order.

For the `when` command, you can use the keyword `stopped` to specify that the actions are to occur whenever the program stops execution.

When you issue the command, an event is added to the event list. If the command sets a breakpoint at a program location, a `B` appears in the line-number region next to the location.

4.3.3.1 Examples

To stop execution the tenth time in function `foo` and print `a`:

```
stop in foo {print a} after 10
```

To stop at line 17 of file `bar` if `a` is equal to 0:

```
stop at "bar":17 if a == 0
```

To stop whenever `a` changes:

```
stop a
```

To stop the third time `a` equals 5:

```
stop if a .eq. 5 after 3
```

To print `a` and do a stack trace every time the program stops execution:

```
when stopped {print a; where}
```

4.3.3.2 For Machine Instructions

To set a breakpoint at a machine instruction, issue the `stopi` command, using the syntax described above, and specifying a machine address. For example,

```
stopi at 0x1000
```

stops execution at address 1000 (hex).

The history region displays the address and the machine instruction. The source pointer moves to the source line being executed.

4.3.3.3 Deleting Breakpoints via the Command Window

To delete a breakpoint via the command window, first issue the `show events` command. This prints out the event list. Each event has an ID number associated with it.

To delete one or more of these events, issue the `delete` command, listing the ID numbers of the events you want to delete; separate multiple IDs with one or more blank spaces. For example,

```
delete 1 3
```

deletes the events with IDs 1 and 3. Use the argument `all` to delete all existing events.

4.4 Tracing Program Execution

You can trace program execution by using the event table or Events menu, or by issuing commands. All methods add an event to the event table. If you trace a source line, Prism displays a `T` next to the line in the line-number region.

As described earlier, tracing is essentially the same as setting a breakpoint, except that execution continues automatically after the breakpoint is reached. When tracing source lines, Prism steps into procedures if they were compiled with the `-g` option; otherwise it steps over them as if it had issued a `next` command.

4.4.1 Using the Event Table and the Events Menu

To trace program execution, choose the Trace**,** Trace <loc>**,** or Trace <var> selection from the `Events` menu. These choices are also available as Common Events buttons within the event table itself.

- Trace displays source lines in the command window before they are executed.
- Trace <loc> prompts for a source line. Prism displays a message immediately prior to the execution of this source line.
- Trace <var> prompts for a variable name. A message is printed when the variable's value changes. The variable can be an array, an array section, or a parallel variable, in which case a message is printed any time any element changes. This slows execution considerably.

In addition, Trace <cond> is available as a Common Events button. It prompts for a condition, which can be any expression that evaluates to true or false; see Section 2.8 for more information on writing expressions. The program displays a message when the condition is met. This also slows execution considerably.

For variations of these traces, you can create your own event in the event table. You can also use the Actions field to specify Prism commands that are to be executed along with the trace.

4.4.1.1 Deleting Traces via the Event Table

To delete a trace, choose the Delete selection from the Events menu, or use the Delete button in the event table itself. See Section 4.2.3.

4.4.2 Using Commands

Issue the `trace` command from the command line to trace program execution. Issuing `trace` with no arguments causes each source line in the program to be displayed in the command window before it is executed.

The `trace` command uses the same syntax as the `stop` command; see Section 4.3.3. For example:

- **trace {print a} –** traces and prints `a` on every source line.
- **trace at 17 if a .GT. 10 –** traces line 17 if `a` is greater than 10.

In addition, Prism interprets

- **trace** *line-number* – as being the same as – **trace at** *line-number*

4.4.2.1 For Machine Instructions

To trace machine instructions, use the `tracei` command, specifying a machine address. When tracing machine instructions, Prism follows all procedure calls down. The `tracei` command has the same syntax as the `stop` command; see Section 4.3.3.

The history region displays the address and the machine instruction. The execution pointer moves to the next source line to be executed.

4.4.2.2 Deleting Traces via the Command Window

To delete a trace, use the `show events` command to obtain the ID associated with the trace, then issue the `delete` command with the ID as its argument. See Section 4.3.3.

4.5 Displaying and Moving Through the Call Stack

The *call stack* is the list of procedures and functions currently active in a program. Prism provides you with methods for examining the contents of the call stack.

See Section 10.6.2 for a discussion of displaying the call stack in MP Prism.

4.5.1 Displaying the Call Stack

- **From the menu bar** – Choose the Where selection from the Debug menu. The Where window is displayed; see FIGURE 4-3. The window contains the call stack; it is updated automatically when execution stops or when you issue commands that change the stack.

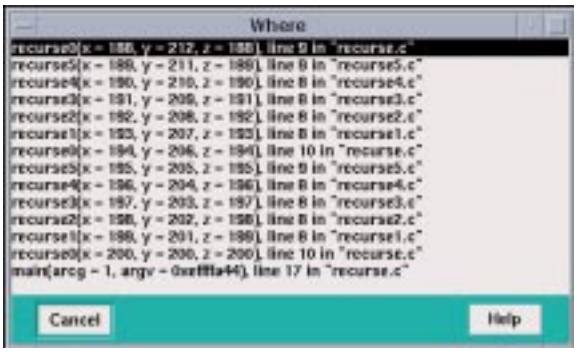

FIGURE 4-3 Where Window

- **From the command window** – Issue the `where` command on the command line. If you include a number, it specifies how many active procedures are to be displayed; otherwise, all active procedures are displayed in the history region.

Values of arguments in displayed procedures are shown in the default radix, which is decimal unless you change it via the `set $radix` command; see Section 5.1.3.

4.5.2 Moving Through the Call Stack

Moving *up* through the call stack means heading toward the main procedure. Moving *down* through the call stack means heading toward the current stopping point in the program.

Moving through the call stack changes the current function and repositions the source window at this function. It also affects the scope that Prism uses for interpreting the names of variables you specify in expressions and commands.

Prism provides these methods for moving through the call stack:

From the menu bar – Choose Up or Down from the Debug menu. Up moves up one level in the call stack; Down moves down one level. These selections are available by default in the tear-off region.

From the command window – Issue the `up` command on the command line to move up one level. If you specify an integer as an argument, you move up that number of levels. Issue the `down` command to move down one level; specifying an integer moves down that number of levels.

From the Where window – If the Where window is displayed, clicking on a function in it changes the stack level to make that function current.

4.6 Examining the Contents of Memory and Registers

You can issue commands in the command window to display the contents of memory addresses and registers.

4.6.1 Displaying Memory

To display the contents of an address, specify the address on the command line, followed by a slash (/). For example,

```
0x10000/
```

If you specify the address as a period, Prism displays the contents of the memory address following the one printed most recently.

Specify a symbolic address by preceding the name with an *&*. For example,

```
&x/
```

prints the contents of memory for variable *x*. The Prism output, for example, might be

```
0x000237f8:  0x3f800000
```

The address you specify can be an expression made up of other addresses and the operators **+**, **-**, and indirection (unary *****). For example,

```
0x1000+100/
```

prints the contents of the location 100 addresses above address 0x1000.

After the slash you can specify how memory is to be displayed. These formats are supported:

d	Print a short word in decimal
D	Print a long word in decimal
o	Print a short word in octal
O	Print a long word in octal
x	Print a short word in hexadecimal
X	Print a long word in hexadecimal
b	Print a byte in octal
c	Print a byte as a character
s	Print a string of characters terminated by a null byte
f	Print a single-precision real number
F	Print a double-precision real number
i	Print the machine instruction

The initial format is *X*. If you omit the format in your command, you get either *X* (if you haven't previously specified a format) or the format you specified previously.

You can print the contents of multiple addresses by specifying a number after the slash (and before the format). For example,

```
0x1000/8X
```

displays the contents of eight memory locations starting at address 0x1000. Contents are displayed as hexadecimal long words.

4.6.2 Displaying the Contents of Registers

You can examine the contents of registers in the same way that you examine the contents of memory. Specify a register by preceding its name with a dollar sign. For example,

```
$f0/
```

prints the contents of the *X* register.

Specify a number after the slash to print the contents of multiple registers. For example,

```
$f0/3
```

prints the contents of registers `f0`, `f1`, and `f2`. The order in which the registers are displayed is that shown in TABLE 4-1.

You can also specify a format, as described above. The format specifier controls the display of the output; it doesn't affect how much of the register contents is displayed. Thus,

```
% $f0/3X
```

displays three registers; the output is displayed as hexadecimal longwords.

Name	Register
`$g0-$g7`	Global registers (64 bits)
`$o0-$o7`	Output registers (64 bits)
`$l0-$l7`	Local registers
`$i0-$i7`	Input registers
`$psr`	Processor state register
`$pc`	Program counter
`$npc`	Next program counter
`$y`	Y register
`$wim`	Window invalid mask
`$tbr`	Trap base register
`$f0-$f31`	Floating-point registers
`$fsr`	Floating status register (64 bits)
`$f0f1-$f62f63`	Floating-point registers
`$xg0-$xg7`	Upper 32 bits of `$g0-$g7`
`$xo0-$xo7`	Upper 32 bits of `$o0-$o7`
`$xfsr`	Upper 32 bits of `$fsr`
`$fprs`	Floating-point registers state
`$tstate`	Trap state register
`$fp`	Frame pointer (synonym for `$i6`)
`$sp`	Stack pointer (synonym for `$o6`)

TABLE 4-1 UltraSPARC Registers

CHAPTER 5

Visualizing Data

This chapter describes how to examine the values of variables and expressions in your program. This is referred to as *visualizing* data. In addition, it describes how to find out the type of a variable and change its values.

See the following sections to learn

- How to choose the variable or expression whose values are to be visualized — *Section* 5.2.
- How to work with graphical visualizers — *Section* 5.3.
- How to save, restore, and compare visualizers — *Section* 5.4.
- How to visualize structures and pointers — *Section* 5.6.
- How to print the type of a variable — *Section* 5.7.
- How to change the values of a variable — *Section* 5.8.
- How to change the radix of data — *Section* 5.9.

See Section 10.7 for a discussion of visualizing data in MP Prism.

5.1 Overview

You can visualize either variables (including arrays, structures, pointers, etc.) or expressions; see Section 2.9 for information on writing expressions in Prism. In addition, you can provide a *context*, so that Prism handles the values of data elements differently, depending on whether they meet the condition you specify.

5.1.1 Printing and Displaying

Prism provides two general methods for visualizing data: printing and displaying.

- Printing data shows the value(s) of the data at a specified point during program execution.
- Displaying data causes its value(s) to be updated every time the program stops execution.

Printing or displaying to the history region of the command window prints out the numeric or character values of the data in standard fashion.

Printing or displaying to a graphical window creates a *visualizer*, which provides you with various options as to how to represent the data.

5.1.2 Visualization Methods

Prism provides these methods for choosing what to print or display:

- By choosing the Print or Display selection from the Debug menu in the menu bar (see Section 5.2.1)
- By selecting text within the source window (see Section 5.2.2)
- By adding events to the event table (see Section 5.2.3)
- By issuing commands from the command window (see Section 5.2.5)

In all cases, choosing Display adds an event to the event list, since displaying data requires an action to update the values each time the program is stopped. Note that, since Display updates automatically, the only way to keep an unwanted display window from reappearing is to delete the corresponding display event.

You create print events only via the event table and the Events menu.

5.1.3 Changing the Default Radix

By default, Prism prints and displays values as decimal numbers. You can change this default by issuing the `set $radix` command, specifying as a setting 2 (binary), 8 (octal), or 16 (hexadecimal). For example,

```
set $radix = 16
```

changes the default representation to hexadecimal. To reset the default to decimal, issue the command

```
set $radix = 10
```

You can override the default for an individual print or display operation. See Section 5.2.5 and Section 5.3.4.

The default setting also affects the display of argument values in procedures in the call stack; see Section 4.5.1.

5.1.4 Data Visualization Limits

Note these points in visualizing data:

- You cannot print or display any variables after a program finishes execution.
- Visualizers do not deal correctly with Fortran adjustable arrays. The size is determined when you create a visualizer for such an array. Subsequent updates to the visualizer will continue to use this same information, even though the size of the array may have changed since the last update. This will result in incorrect values in the visualizer. Printing or displaying values of an adjustable array in the command window or to a new window will work, however.

5.2 Choosing the Data to Visualize

This section describes the methods Prism provides for printing and displaying data.

5.2.1 Printing and Displaying From the Debug Menu

To print a variable or expression at the current program location, choose Print from the Debug menu. It is also by default in the tear-off region.

To display a variable or expression every time execution stops, starting at the current program location, choose Display from the Debug menu.

When you choose Print or Display, a dialog box appears; FIGURE 5-1 shows an example of the Print dialog box.

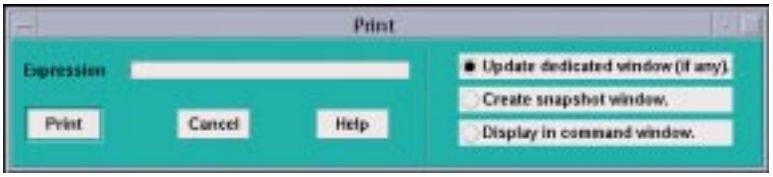

FIGURE 5-1 Print Dialog Box

In the Expression box, enter the variable or expression whose value(s) you want printed. Text selected in the source window appears as the default; you can edit this text.

The dialog boxes also offer choices as to the window in which the values are to appear:

- You can specify that the values are to be printed or displayed in a standard window dedicated to the specified expression. The first time you print or display the data, Prism creates this window. If you print data, and subsequently print it again, this standard window is updated. This is the default choice for both Print and Display.
- You can create a separate *snapshot* window for printing or displaying values. This is useful if you want to compare values between windows.
- You can print out the values in the command window.

Click on Print or Display to print the values of the specified expression at the current program location.

Click on Cancel or press the Esc key to close the window without printing or displaying.

5.2.2 Printing and Displaying from the Source Window

▼ To Print and Display From the Source Window

1. **Select the variable or expression by dragging over it with the mouse or double-clicking on it.**
2. **Right-click the mouse to display a pop-up menu.**
3. **Click on Print in this menu to display a snapshot visualizer containing the value(s) of the selected variable or expression at that point in the program's execution.**

   Click on Display to display a visualizer that is automatically updated whenever execution stops.

   To print without bothering to display the menu, press the Shift key while selecting the variable or expression.

Note – Prism prints the correct variable when you choose it in this way, even if the scope pointer sets a scope that contains another variable of the same name.

5.2.3 Printing and Displaying From the Events Menu

You can use the Events menu to define a print or display event that is to take place at a specified location in the program.

The Print dialog box (see FIGURE 5-2) prompts for the variable or expression whose value(s) are to be printed, the program location at which the printing is to take place, and the name of the window in which the value(s) are to be displayed.

FIGURE 5-2 Print Dialog Box

Window names are dedicated, snapshot, and command; you can also make up your own name. The default is dedicated. See Section 2.7.3 for a discussion of these names.

When you have filled in the fields, click on OK; the event is added to the event table. When the location is reached in the program, the value(s) of the expression or variable are printed.

The Display dialog box is similar, but it does not prompt for a location; the display visualizer will update every time the program stops execution.

5.2.4 Printing and Displaying From the Event Table

You can use the event table to define a print or display event that is to take place at a specified location in the program.

Click on Print or Display in the Common Events buttons to create an event that will print or display data.

If you click on Print, the Location and Action fields are highlighted. Put a program location in the Location field. Complete the print event in the Actions field, specifying the variable or expression, and the window in which it is to be printed. For example,

```
print d2 on dedicated
```

If you click on Display, the Location field displays `stopped`, and the Actions field displays `print on dedicated`. Complete the description of the print event, as described above. The variable or expression you specify is then displayed whenever the program stops execution.

5.2.5 Printing and Displaying from the Command Window

Use the `print` command to print the value(s) of a variable or expression from the command window. Use the `display` command to display the value(s). The `display` command prints the value(s) of the variable or expression immediately, and creates a display event so that the values are updated automatically whenever the program stops.

The commands have this format:

[**where (***expression***)**] *command variable*[**,** *variable* **...**]

The optional `where` **(***expression***)** syntax sets the context for printing the variable or expression; see below.

In the syntax, *command* is either `print` or `display`, and *variable* is the variable or expression to be displayed or printed.

Redirection of output to a window via the *on window* syntax works slightly differently for `display` and `print` from the way it works for other commands; see Section 2.7.3 for a discussion of redirection. Separate windows are created for each variable or expression that you print or display. Thus, the commands

```
display x on dedicated
display y/4 on dedicated
display [0:128:2]z on dedicated
```

create three windows, each of which is updated separately.

To print or display the contents of a register, precede the register's name with a dollar sign. For example,

```
print $pc
```

prints the program counter register. See Section 4.6.2 for a list of register names supported by Prism.

5.2.5.1 Setting the Context

You can precede the `print` or `display` command with a `where` statement that can make elements of a variable or array *inactive*. Inactive elements are not printed in the command window; Section 5.3.4 describes how they are treated in visualizers. Making elements inactive is referred to as *setting the context*.

To set the context, follow the `where` keyword with an expression in parentheses. The expression must evaluate to true or false for every element of the variable or array being printed.

For example,

```
where (i .gt. 0) print i
```

prints (in the command window) only the values of `i` that are greater than 0.

You can use certain Fortran intrinsics in the `where` statement. For example,

```
where (a .eq. maxval(a)) print a
```

prints the element of `a` that has the largest value. (This is equivalent to the `MAXLOC` intrinsic function.) See Section 2.8 for more information on writing expressions in Prism.

Note that setting the context affects only the printing or displaying of the variable. It does not affect the actual context of the program as it executes.

5.2.5.2 Specifying the Radix

You can specify the radix to be used in printing or displaying values by adding a suffix of the form `/radix` to the `print` or `display` command. `radix` can be `b` (binary), `d` (decimal), `x` (hexadecimal), or `o` (octal). For example,

```
print/b pvar1
```

prints the binary representation of `pvar1` in the command window.

```
display/x pvar2 on dedicated
```

displays the hexadecimal values of `pvar2` in a dedicated window.

The default radix is decimal, unless you have used the `set $radix` command to change it; see Section 5.1.3.

5.3 Working with Visualizers

The window that contains the data being printed or displayed is called a *visualizer.* FIGURE 5-3 shows a visualizer for a 3-dimensional array.

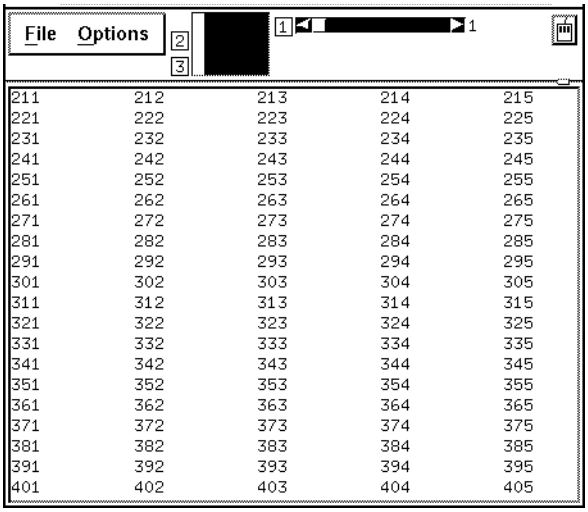

FIGURE 5-3 Visualizer for a 3-Dimensional Array.

The visualizer consists of two parts: the *data navigator* and the *display window*. There are also File and Options pulldown menus.

The *data navigator* shows which portion of the data is being displayed, and provides a quick method for moving through the data. The appearance of the data navigator depends on the number of dimensions in the data. It is described in more detail in Section 5.3.1.

The *display window* is the main part of the visualizer. It shows the data, using a representation that you can choose from the Options menu. The default is `text`: that is, the data is displayed as numbers or characters. FIGURE 5-3 is a text visualizer. The display window is described in more detail in Section 5.3.2.

The File menu lets you save, update, or cancel the visualizer; see Section 5.3.3 for more information. The Options menu, among other things, lets you change the way values are represented; see Section 5.3.4.

5.3.1 Using the Data Navigator in a Visualizer

The data navigator helps you move through the data being visualized. It has different appearances, depending on the number of dimensions in your data. If your data is a single scalar value, there is no data navigator.

For 1-dimensional arrays and parallel variables, the data navigator is the scroll bar to the right of the data. The number to the right of the buttons for the File and Options menus indicates the coordinate of the first element that is displayed. The elevator in the scroll bar indicates the position of the displayed data relative to the entire data set.

For 2-dimensional data, the data navigator is a rectangle in the shape of the data, with the axes numbered. The white box inside the rectangle indicates the position of the displayed data relative to the entire data set. You can either drag the box or click at a spot in the rectangle. The box moves to that spot, and the data displayed in the display window changes.

For 3-dimensional data, the data navigator consists of a rectangle and a slider, each of which you can operate independently. The value to the right of the slider indicates the coordinate of the third dimension. Changing the position of the bar along the slider changes which 2-dimensional plane is displayed out of the 3-dimensional data.

For data with more than three dimensions, the data navigator adds a slider for each additional dimension.

5.3.1.1 Changing the Axes

You can change the way the visualizer lays out your data by changing the numbers that label the axes. Click in the box surrounding the number; it is highlighted, and an I-beam appears. You can then type in the new number of the axis; you don't have to delete the old number. The other axis number automatically changes; for example, if you change axis 1 to 2, axis 2 automatically changes to become axis 1.

5.3.2 Using the Display Window in a Visualizer

The display window shows the data being visualized.

In addition to using the data navigator to move through the data, you can drag the data itself relative to the display window by holding down the left mouse button; this provides finer control over the display of the data.

To find out the coordinates and value of a specific data element, click on it while pressing the Shift key. Its coordinates are displayed in parentheses, and its value is displayed beneath them. If you have set a context for the visualizer, you also see whether the element is active or inactive (see Section 5.3.4). Drag the mouse with the Shift key pressed, and you see the coordinates, value, and context of each data element over which the mouse pointer passes.

You can resize the visualizer to display more (or less) data either horizontally or vertically.

5.3.3 Using the File Menu

Click on File to pull down the File menu.

Choose Update from this menu to update the display window for this variable, using the value(s) at the current program location. See Section 5.3.5 for more information on updating a visualizer.

Choose Save or Save As to save the visualizer's values to a file. See Section 5.4.1 for more information.

Choose Diff or Diff With to compare the visualizer's values with values stored in a file. See Section 5.4.3 for more information.

Choose Snapshot to create a copy of the visualizer, which you can use to compare with later updates.

Choose Close to cancel the visualizer.

5.3.4 Using the Options Menu

Click on Options to pull down the Options menu. See FIGURE 5-4.

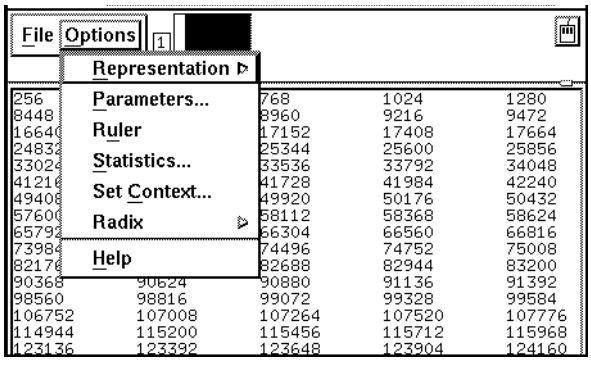

FIGURE 5-4 Options Menu in a Visualizer

5.3.4.1 Choosing the Representation

Choose Representation from the Options menu to display another menu that gives the choices for how the values are represented in the display window. The choices are described below. You can control aspects of the way these visualizers appear by changing their parameters, as described later in this section.

- Choose Text to display the values as numbers or letters. This is the default.
- Choose Histogram to display the values of an array or parallel variable in a histogram. See FIGURE 5-5 for an example.

The vertical axis displays the number of data points; the horizontal axis displays the range of values. Prism divides up this range evenly in creating the histogram bars. It prints summary data above the histogram.

Shift-click on a histogram bar to display the range and number of data points it represents.

Note that the histogram represents all the values of the variable, not just those shown in the 2-dimensional slice of data that happens to be displayed in other representations.

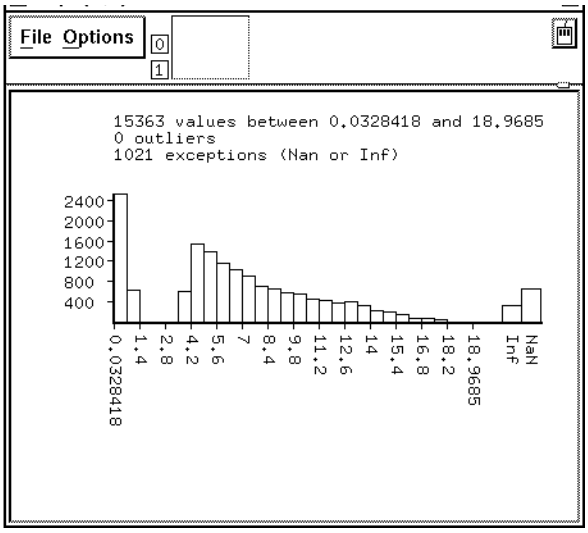

FIGURE 5-5 Histogram Visualizer

- Choose Dither to display the values as a shading from black to white. Groups of values in a low range are assigned more black pixels; groups of values in a high range are assigned more white pixels. This has the effect of displaying the data in

various shades of gray. FIGURE 5-6 shows a 2-dimensional dither visualizer. The lighter area indicates values that are higher than values in the surrounding areas; the darker area indicates values that are lower than surrounding values.

For complex numbers, Prism uses the modulus.

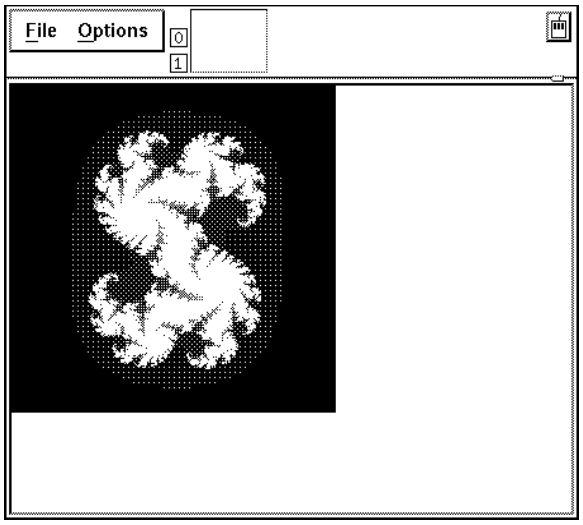

FIGURE 5-6 Dither Visualizer

- Choose Threshold to display the values as black or white. By default, Prism uses the mean of the values as the threshold; values less than or equal to the mean are black, and values greater than the mean are white. FIGURE 5-7 shows a threshold representation of a 3-dimensional array.

For complex numbers, Prism uses the modulus.

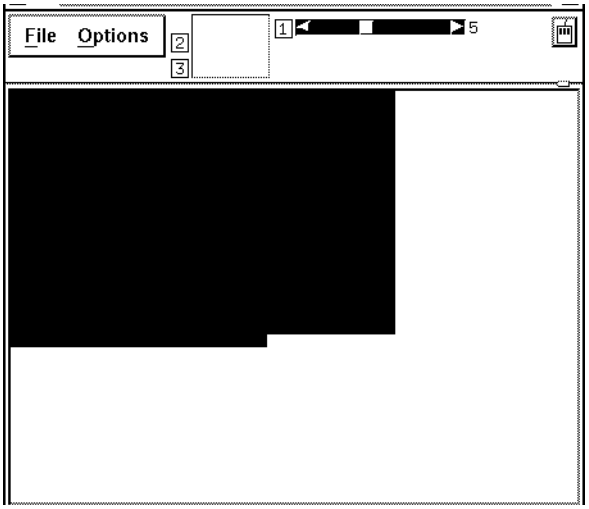

FIGURE 5-7 Threshold Visualizer

- Choose Colormap (if you are using a color workstation) to display the values as a range of colors. By default, Prism displays the values as a continuous spectrum from blue (for the minimum value) to red (for the maximum value). You can change the colors that Prism uses; see Section 9.3.2.

For complex numbers, Prism uses the modulus.

- Choose Graph to display values as a graph, with the index of each array element plotted on the horizontal axis and its value on the vertical axis. A line connects the points plotted on the graph. This representation is particularly useful for 1-dimensional data, but can be used for higher-dimensional data as well; for example, in a 2-dimensional array, graphs are shown for each separate 1-dimensional slice of the 2-dimensional plane.

FIGURE 5-8 shows a graph visualizer for a 1-dimensional array.

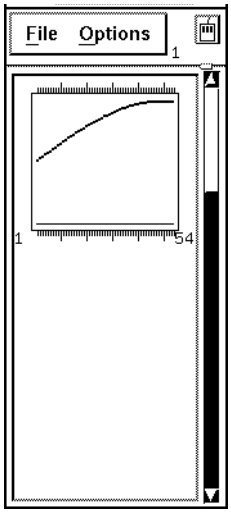

FIGURE 5-8 i-Dimensional Graph Visualizer

- Choose Surface (if your data has more than one dimension) to render the 3-dimensional contours of a 2-dimensional slice of data. In the representation, the 2-dimensional slice of data is tilted 45 degrees away from the viewer, with the top edge further from the viewer than the bottom edge. The data values rise out of this slice. FIGURE 5-9 is an example.

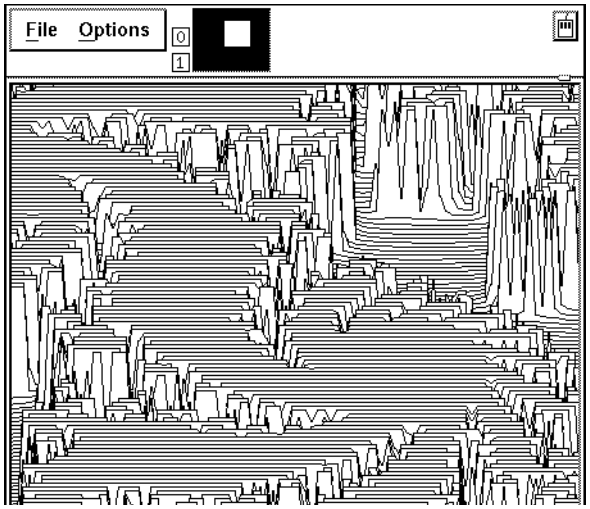

FIGURE 5-9 Surface Visualizer

Note – If there are large values in the top rows of the data, they may be drawn off the top of the screen. To see these values, flip the axes as described earlier in this section, so that the top row appears in the left column.

- Choose Vector to display data as vectors. The data must be a Fortran complex or double complex number, or a pair of variables to which the `CMPLX` intrinsic function has been applied (see Section 2.8.2). The complex number is drawn showing both magnitude and direction. The length of the vector increases with magnitude. There is a minimum vector length of five pixels, because direction is difficult to see for smaller vectors. By default, the lengths of all vectors scale linearly with magnitude, varying between the minimum and maximum vector lengths. FIGURE 5-10 shows a vector visualizer.

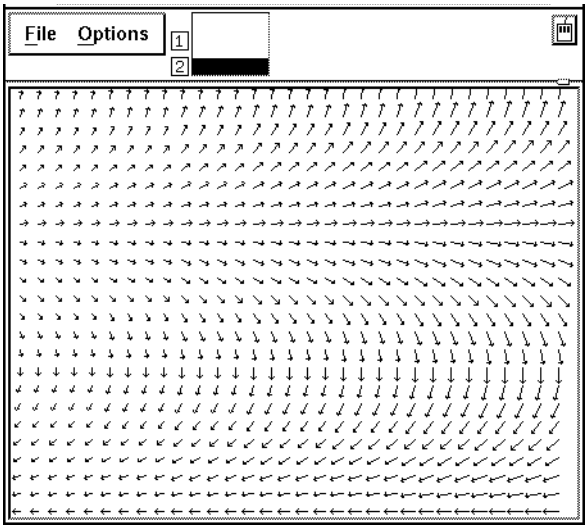

FIGURE 5-10 Vector Visualizer

5.3.4.2 Setting Parameters

Choose Parameters from the Options menu to display a dialog box in which you can change various defaults that Prism uses in setting up the display window; see FIGURE 5-11. If a parameter is grayed out or missing, it does not apply to the current representation.

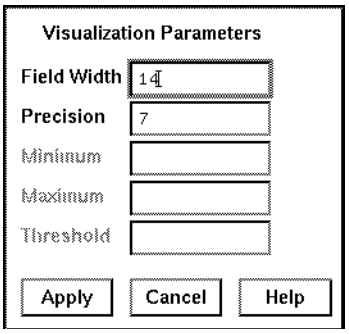

FIGURE 5-11 Visualization Parameters Dialog Box

The parameters (for all representations except the histogram representation) are:

- **Field Width** – Type a value in this box to change the width of the field that Prism allocates to every data element.

For the text representation, the field width specifies the number of characters in each column. If a number is too large for the field width you specify, dots are printed instead of the number.

For dither, threshold, colormap, and vector representations, the field width specifies how wide (in pixels) the representation of each data element is to be. By default, dither, threshold, and colormap visualizers are scaled to fit the display window. Note, however, that for dither visualizers, the gray shading may be more noticeable with a smaller field width.

For the graph representation, the field width specifies the horizontal spacing between elements.

For the surface representation, it specifies the spacing of elements along both directions of the plane.

- **Field Height** – For graph and surface representations, changing this value affects the maximum height (in pixels) to which Prism scales every data value.
- **Precision** – Type a value in this box to change the precision with which Prism displays real numbers in a text visualizer. The precision must be less than the field width. By default, Prism prints doubles with 16 significant digits, and floating-point values with 7 significant digits. You can change this default by issuing the `set` command with the `$d_precision` variable (for doubles) or `$f_precision` variable (for floating-point values). For example,

  ```
  set $d_precision = 11
  ```

  sets the default precision for doubles to 11 significant digits.
- **Minimum** and **Maximum** – For colormap representations, use these variables to specify the minimum and maximum values that Prism is to use in assigning color values to the data elements. Data elements that have values below the minimum and above the maximum are assigned default colors.

  For graph, surface, and vector representations, these parameters represent the bottom and top of the range that is to be represented. Values below the minimum are shown as the minimum; values above the maximum are shown as the maximum.

  By default Prism uses the entire range of values for all these representations.
- **Threshold** – For threshold representations, use this variable to specify the value at which Prism is to change the display from black to white. Data elements whose values are at or below the threshold are displayed as black; data elements whose values are above the threshold are displayed as white. By default, Prism uses the mean of the data as the threshold.

The parameters for the histogram representation are:

- **Bar Width** – Specifies the width in pixels of each histogram bar (except for the bars representing infinities and NaNs, which must be wide enough to fit the `Inf` or `NaN` label underneath). The default is 10 pixels.
- **Bar Height** – Specifies the height in pixels of the largest histogram bar. The default is 100 pixels.
- **Minimum** – Specifies the minimum value to be included in the histogram. By default the actual minimum value is used.
- **Maximum**– Specifies the maximum value to be included in the histogram. By default the actual maximum value is used.

  If you specify a different minimum or maximum, values below the minimum or above the maximum are not displayed in the histogram, but are counted as outliers instead; the number of outliers is displayed above the histogram.
- **Max Buckets** – Specifies the number of "buckets" into which values are to be poured—in other words, the number of histogram bars to be used. The default is 30. (Prism may use fewer to make the horizontal labels come out evenly.)

5.3.4.3 Displaying a Ruler

Choose Ruler from the Options menu to toggle the display of a ruler around the data in the display window. The ruler is helpful in showing which elements are being displayed. FIGURE 5-12 shows a 3-dimensional threshold visualizer with the ruler displayed.

In the surface representation, the ruler cannot indicate the coordinates of elements in the vertical axis, since they change depending on the height of each element. However, you can press the Shift key and left-click as described above to display the coordinates and value of an element.

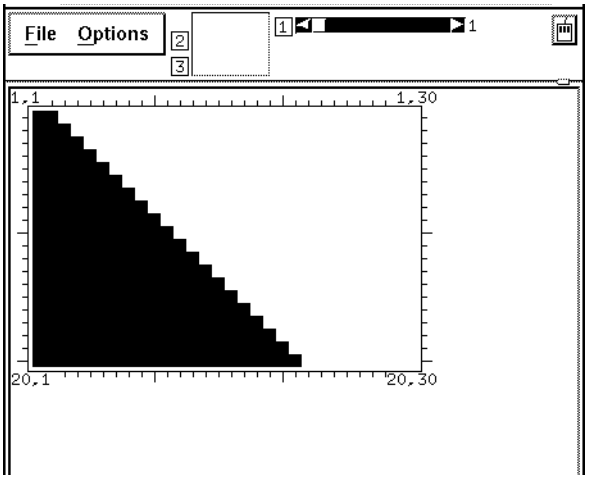

FIGURE 5-12 Threshold Visualizer with a Ruler

5.3.4.4 Displaying Statistics

Choose Statistics from the Options menu to display a window containing statistics and other information about the variable being visualized. The window contains:

- The name of the variable
- Its type and number of dimensions
- The total number of elements the variable contains, and the total number of active elements, based on the context you set within Prism (see the next section for a discussion of setting the context)
- The variable's minimum, maximum, and mean; these statistics reflect the context you set for the visualizer

FIGURE 5-13 gives an example of the Statistics window.

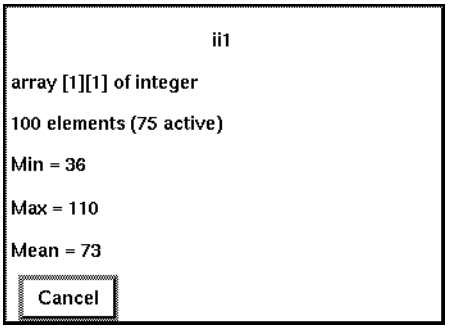

FIGURE 5-13 Statistics for a Visualizer

For complex numbers, Prism uses the modulus.

5.3.4.5 Using the Set Context Dialog Box

Choose Set Context from the Options menu to display a dialog box in which you can specify which elements of the variable are to be considered active and which are to be considered inactive. Active and inactive elements are treated differently in visualizers:

- In text, graph, surface, and vector visualizers, inactive elements are grayed out.
- In colormap visualizers, inactive elements by default are displayed as gray. You can change this default; see Section 9.3.2.
- Context has no effect on dither and threshold visualizers.

FIGURE 5-14 shows the Set Context dialog box.

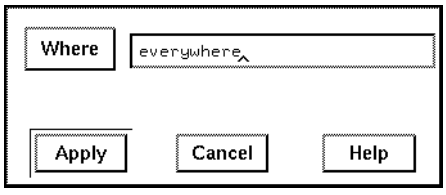

FIGURE 5-14 Set Context Dialog Box

By default, all elements of the variable are active; this is the meaning of the everywhere keyword in the text-entry box. To change this default, you can either edit the text in the text-entry box directly or click on the Where button to display a menu. The choices in the menu are everywhere and other:

- Choose everywhere, as mentioned above, to make all elements active.

- Choose other to erase the current contents of the text-entry box. You can then enter an expression into the text-entry box.

In the text-entry box, you can enter any valid expression that will evaluate to true or false for each element of the variable.

The context you specify for printing does not affect the program's context; it just affects the way the elements of the variable are displayed in the visualizer.

See "Setting the Context" above for more information on context. See Section 2.8 for more information on writing expressions in Prism.

Click on Apply to set the context you specified. Click on Cancel or press the Esc key to close the dialog box without setting the context.

5.3.4.6 Changing the Radix

Choose Radix from the Options menu to change the radix used in the text representation of a value.

Choosing Radix pulls down a submenu with four selections: Decimal, Hex, Octal, and Binary. Choosing one of these changes the value to the specified radix. Prism continues to use this radix if the visualizer is updated.

By default, Prism displays values in decimal. You can change this default via the `set $radix` command; see Section 5.1.3. You can also override it for a specific `print` or `display` command; see Section 5.2.5.

5.3.5 Updating and Closing the Visualizer

If you created a visualizer by issuing a `display` command, it automatically updates every time the program stops execution.

If you created the visualizer by issuing a `print` command, its display window is grayed out when the program resumes execution and the values in the window are outdated. To update the values, choose Update from the visualizer's File menu.

To close the visualizer, choose Close from the File menu, or press the Esc key.

5.4 Saving, Restoring, and Comparing Visualizers

You can save the values of a variable or expression to a file. You can subsequently visualize these values and compare them with the values in another visualizer—for example, the same variable later in the run, or during a totally separate execution of the program. This provides a convenient way of spotting changes in the values of a variable.

5.4.1 Saving the Values of a Variable

You can save the values of a variable or expression to a file for later use.

5.4.1.1 From the Command Line

Use the command `varsave` to save the values of a variable or expression to a file. Its syntax is

`varsave "`*`filename`***`"`** *expression*

where *filename* is the name of the file to which the data is to be saved, and *expression* is the variable or expression whose values are to be saved. For example,

```
varsave "alpha.data" alpha
```

saves the values of the variable `alpha` in the file `alpha.data` (in your current working directory within Prism).

```
varsave "/u/kathy/alpha2.data" alpha*2
```

saves the results of the expression `alpha*2` in the file with the pathname `/u/kathy/alpha2.data`.

5.4.1.2 From a Visualizer

Use the Save or Save As selection from a visualizer's File menu to save the visualizer's values to a file.

If you choose Save As, a dialog box appears in which you can specify the name of the file to which the values are to be saved:

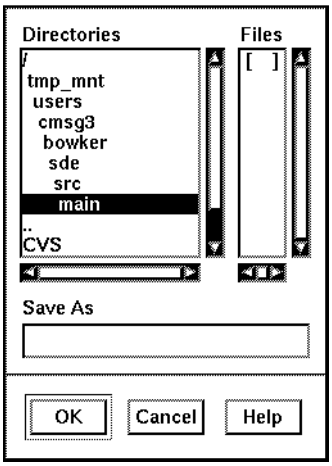

FIGURE 5-15 Saving Visualizer's Data to a File

The highlighted directory is the current working directory. If you want to put the file there, simply type its name in the Save As box and click on OK.

If you want to put the file in another directory, click on the directory. (The parent directories of the current working directory are shown above it in the Directories list; its subdirectories are listed beneath it.) This will display the subdirectories of the directory you clicked on. You can traverse the directory structure in this manner until you find the directory in which you want to put the file, or, you can simply type the entire path name in the Save As box.

Choose the `Save` selection to save the values in the file you most recently specified. If you haven't specified a file, the values are saved in a file called `noname.var` in your current working directory in Prism.

5.4.2 Restoring the Data

Use the intrinsic `varfile` to bring values you have saved to a file back into Prism. Its syntax is

`varfile("`*`filename`***`")`**

where *filename* is the name of the file that contains the values you want to restore.

Note – The `varfile` intrinsic is not available in MP Prism.

You can use the **varfile** intrinsic anywhere you could have used the original variable or expression that you saved to a file. For example, if you saved *x*:

```
varsave "x.var" x
```

then the command

```
print varfile("x.var")
```

is equivalent to

```
print x
```

Note that this allows you to save a variable's values, then print them during a later Prism session, without having a program loaded or running.

5.4.3 Comparing the Data

You can compare a variable or expression whose values have been saved in a file with another version of the variable or expression. This comparison could take place later in the same run of the program, during a subsequent run, or even during a second, simultaneous Prism session.

You can also compare the values with those of another variable, as long as both variables have the same base type (that is, you can't compare integers with floating-point numbers).

5.4.3.1 From the Command Line

You can use the `print` or `display` command with the difference operator and the `varfile` intrinsic to perform a comparison between two versions of a variable or expression.

For example, if you saved *x* in the file `x.var`:

```
varsave "x.var" x
```

then the command

```
print x - varfile("x.var")
```

prints the difference between the current and saved values of *x*.

If an element is printed as 0, it is the same in both versions. If it is nonzero, its value is different in the two versions.

5.4.3.2 From a Visualizer

Use the Diff or Diff With selection from a visualizer's File menu to compare the visualizer's values with values stored in a file.

Choose Diff With to choose the file containing the values. It displays a dialog box like the one shown below.

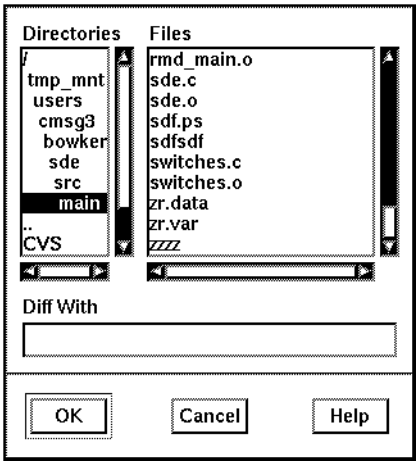

FIGURE 5-16 Diff With Dialog Box

The dialog box has the same format as the Save As dialog box described in Section 5.4.1. It lists the files found in your current working directory in Prism. Click on a file name, then click on OK to choose the file. Or type a file name in the Diff With text-entry box and click on OK.

Choose Diff to compare the visualizers values to those in the most recently specified file; if no file has been specified, values are compared to those in the file `noname.var` in your current working directory in Prism.

Once you have specified a file via Diff or Diff With, Prism creates a new visualizer that displays the difference in values between the visualizer and the file. If an element's value in the new visualizer is 0, the value is the same in both versions. If it is nonzero, it is different in the two versions.

You can work with this visualizer as you would any visualizer. For example, you can change the representation and display summary statistics.

5.5 Visualizing Layouts of Parallel Objects

Prism provides a `layout` intrinsic that returns the numbers of the nodes on which the data elements of a parallel object are located. Its syntax is

`layout(`*`pvar`***`)`**

where *pvar* is a parallel object. For Sun HPF, it can be a parallel array. You can use the Fortran 90 array-section syntax described in Section2.8.4 to specify a range of elements within a parallel object.

You can print or display the results of applying the `layout` intrinsic to a parallel object. For example,

```
print layout(p1) on dedicated
```

creates a visualizer that is the same size and shape as the parallel object `p1`. The visualizer displays the rank of the process that is holding each value.

Note that you can use other visualizer representations—for example, dither or colormap—to display the layout graphically.

5.6 Visualizing Structures

If you print a pointer or a structure (or a structure-valued expression) in a window, a *structure visualizer* appears.

FIGURE 5-17 shows an example of a structure visualizer.

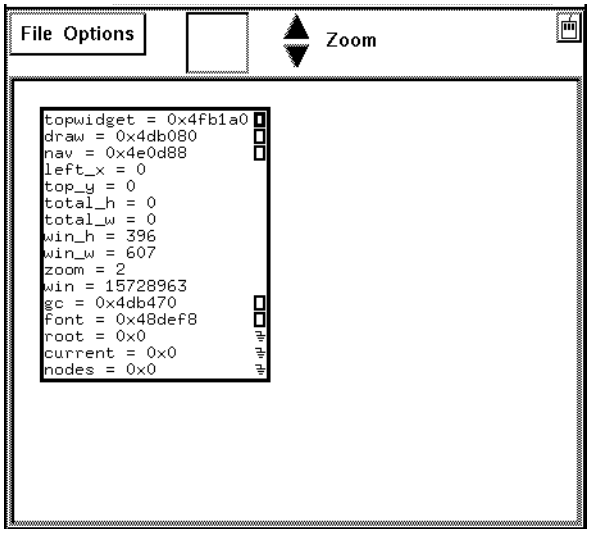

FIGURE 5-17 Structure Visualizer

The structure you specified appears inside a box; this is referred to as a *node*. The node shows the fields in the structure and their values. If the structure contains pointers, small boxes appear next to them; they are referred to as *buttons*. Left-click on a node to select it. Use the up and down arrow keys to move between buttons of a selected node.

You can perform various actions within a structure visualizer, as described below.

5.6.1 Expanding Pointers

You can expand scalar pointers in a structure to generate new nodes. (You cannot expand a pointer to a parallel variable.)

To expand a single pointer:

- **With a mouse** – Left-click on a button to expand the pointer. For example, clicking on the button next to the nav field in FIGURE 5-17 changes the visualizer as shown in FIGURE 5-18.
- **From the keyboard** – Use the right arrow key to expand and visit the node pointed to by the current button. If the node is already expanded, pressing the right arrow key simply visits the node. Use the left arrow key to visit the parent of a selected node.

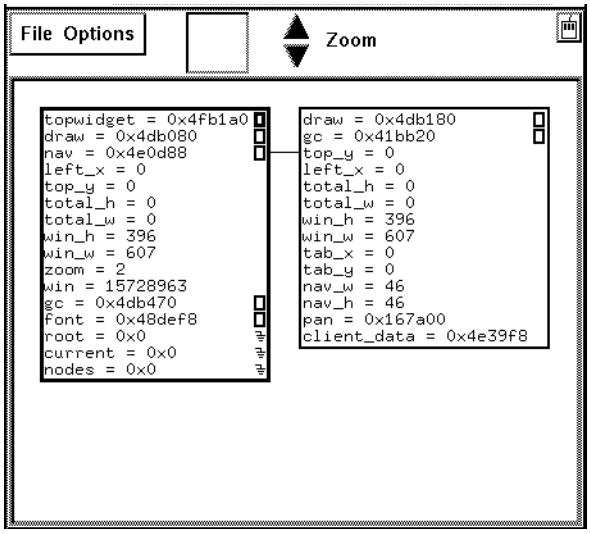

FIGURE 5-18 Structure Visualizer, With One Pointer Expanded

To expand all pointers in a node:

- **With the mouse** – Double-click or Shift-left-click on the node.
- **From the keyboard** – Press the Shift key along with the right arrow key.
- **From the Options menu** – Click on Expand. The cursor turns into a target; move the cursor to the node you are interested in and left-click.

To recursively expand all pointers from the selected node on down:

- **With the mouse** – Triple-click or Control-left-click on the node.
- **From the keyboard** – Press the Control key and the right arrow key.
- **From the Options menu** – Click on Expand All. The cursor turns into a target; move the cursor to the node you are interested in and left-click.

5.6.2 Panning and Zooming

You can left-click and drag through the data navigator or the display window to pan through the data, just as you can with visualizers; see Sections 5.3.1 and 5.3.2.

You can also “zoom” in and out on the data by left-clicking on the Zoom arrows. Click on the down arrow to zoom out and see a bird’s-eye view of the structure; click on the up arrow to get a closeup. FIGURE 5-19 shows part of a complicated structure visualizer in which we have zoomed out.

Left-click on a node in a zoomed-out structure visualizer to pop up a window showing the full contents of the node.

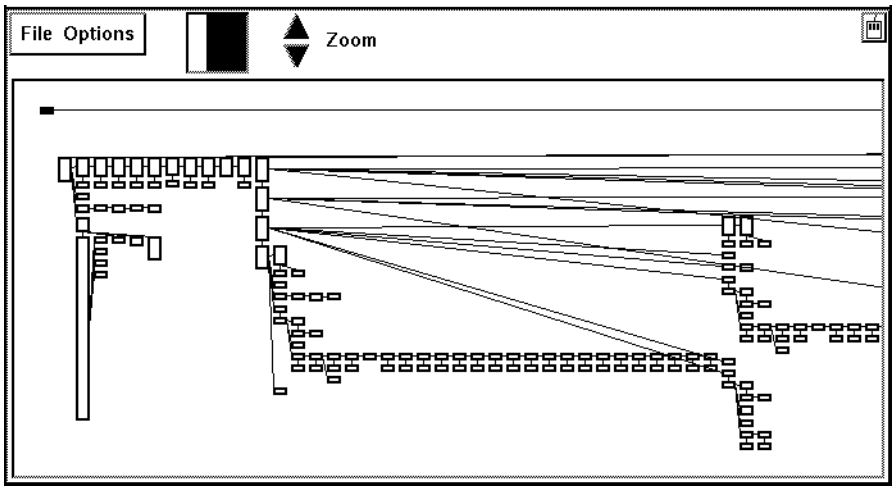

FIGURE 5-19 Zooming Out in a Structure Visualizer

The selected node is centered in the display window whenever you zoom in or out.

5.6.3 Deleting Nodes

To delete a node (except the root node),

- **With the mouse** – Middle-click on a node (except the root node).
- **From the Options menu** – Click on Delete. The cursor turns into a target; move the cursor to the node you want to delete and left-click.

Deleting a node also deletes its children (if any).

5.6.4 More about Pointers in Structures

Note the following about pointers in structure visualizers:

- Null pointers—for example, root in FIGURE 5-18—have “ground” symbols next to them.
- If a pointer has previously been expanded, it has an arrow next to its button; you can’t expand the pointer again. (This prevents infinite loops on circular data structures.)

- A pointer containing a bad address has an **X** drawn over its button.

5.6.5 Updating and Closing a Structure Visualizer

Left-click on Update in the File menu to update a structure visualizer. When you do this, the root node is re-read; Prism attempts to expand the same nodes that are currently expanded. (The same thing happens if you re-print an existing structure visualizer.)

Left-click on Close in the File menu to close the structure visualizer.

5.7 Printing the Type of a Variable

Prism provides several methods for finding out the type of a variable.

From the menu bar – Choose the Whatis selection from the Debug menu. The Whatis dialog box appears; it prompts for the name of a variable. Click on Whatis to display the information about the variable in the command window.

From the source window – Select a variable by double-clicking on it or by dragging over it while pressing the left mouse button. Then hold down the right mouse button; a pop-up menu appears. Choose Whatis from this menu. Information about the variable appears in the command window.

From the command window – Issue the `whatis` command from the command line, specifying the name of the variable as its argument.

5.7.1 What Is Displayed

Prism displays the information about the variable in the command window. For example,

```
whatis primes
logical primes(1:999)
```

5.8 Modifying Data

You can use the `assign` command to assign new values to a variable or an array. For example,

```
assign x = 0
```

assigns the value 0 to the variable `x`. You can put anything on the left-hand side of the statement that can go on the left-hand side in the language you are using—for example, a variable or a Fortran array section.

If the right-hand side does not have the same type as the left-hand side, Prism performs the proper type coercion.

5.9 Changing the Radix of Data

Use the command *value* = *base* to change the radix of a value in Prism. The value can be a decimal, hexadecimal, or octal number. Precede hexadecimal numbers with `0x`; precede octal numbers with `0` (zero). The base can be `D` (decimal), `X` (hexadecimal), or `O` (octal). Prism prints the converted value in the command window.

For example, to convert 100 (hex) to decimal, issue this command:

```
0x100=D
```

Prism responds:

```
256
```

5.10 Printing the Names and Values of Local Variables

Use the `dump` command, followed by the name of a function or procedure, to print the names and values of all local variables in that function or procedure. If you omit the function name, `dump` uses the current function. If you specify a period, `dump` prints the names and values of all local variables in the functions in the stack.

CHAPTER 6

Obtaining Performance Data

Prism lets you collect performance data on your Sun HPF program. Collecting and analyzing performance data can help you uncover and correct bottlenecks that slow down a program.

Section 6.1 is an overview See the following sections to learn:

- How to write and compile your program to obtain performance data — *Section* 6.2.
- How to obtain the most accurate performance data — *Section* 6.3.
- How to collect performance data — *Section* 6.4.
- How to display performance data — *Section* 6.5.
- How to interpret performance data — *Section* 6.6.
- How to save a file of performance data and reload it into Prism — *Section* 6.7.

6.1 Overview

Prism helps you determine where your Sun HPF program is spending its time, and why.

To determine where your program is spending its time, Prism provides data at the level of the entire program, individual procedures within the program (with both call-graph and flat displays), and individual source lines within procedures. This allows you to zero in on the lines that have the greatest impact on a program's performance.

To determine why a procedure or a source line is a bottleneck in your program, Prism provides data on a program's use of several different computing resources, not just CPU time. For example, the code may be doing a lot of scatter/gather communication or I/O. Providing data on the code's use of these resources makes it easier to determine how, or if, the code's performance can be improved.

In addition to displaying the data, Prism provides tips on how to interpret the data. See Section 6.6 for more information.

6.2 Writing and Compiling Your Program

To collect performance data on Sun HPF programs, you must compile (and link) your program with the `-tmprofile` option. If your program calls routines that were not compiled with the `-tmprofile` option, such as a routine from a library like S3L, Prism still provides summary information under the `Code not profiled` category.

Note – Prism reports summary information on routines compiled without the `-tmprofile` option, but only if such routines are called (directly or indirectly) from other routines that have been compiled (and linked) with the `-tmprofile` option.

6.3 Obtaining the Most Accurate Performance Data

This section gives some hints on how to obtain the most accurate performance data in Prism.

Note these general points:

- Collecting performance data slows execution of the program. The exact degree to which this occurs depends on the program. In general, programs whose processes work on small amounts of data are affected more by this performance-collection overhead. Prism corrects for this effect when it presents its data. If you are using your own timers to measure performance during a run in which Prism is also collecting data, however, you need to be aware that this effect will inflate the values in these timers.
- Interrupting your program while collecting performance data (for example, by stopping at a breakpoint and printing values) will distort the data.

6.4 Collecting Performance Data

To collect performance data, you must turn collection on before running the program. Collection remains on until you explicitly turn it off.

- **From the menu bar** – Choose Collection from the Performance menu. (This selection is also available by default in the tear-off region.) Collection toggles the collection of performance data. Performance collection is off when the toggle box to the left of the menu selection is not filled in; this is the default. Choosing Collection turns it on, and the toggle box is filled in. To turn it off, choose Collection when the toggle box is filled in.
- **From the command window** – Issue the `collection on` command to turn collection on; issue `collection off` to turn it off. Issuing the `collection` command also affects the state of the toggle box in the Collection menu selection.

6.4.1 Collecting Performance Data Outside of Prism

You can also collect performance data by setting environment variables, without entering Prism. This is convenient if you can't enter Prism for some reason (for example, because the Sun HPC partition is only accepting batch jobs).

To turn on collection of performance data, set the environment variable `TMPROF` to `t`. For example (in the C shell),

```
% setenv TMPROF t
```

To turn collection off, you can either set `TMPROF` to `f`, or unset it. For example,

```
% unsetenv TMPROF
```

Note – Setting the environment variable `TMPROF` outside of Prism has no effect on the Collection command within Prism. In fact, choosing Collection from the Performance menu (or issuing `collection on` from the command window) within Prism causes `TMPROF` to be set to t.

To specify the program on which data is to be collected, set the environment variable `TMPROF_EXEC` to the path name of the executable program. For example,

```
% setenv TMPROF_EXEC a.out
```

To specify the file to which the performance data is to be sent, set the environment variable `TMPROF_DATAFILE` to the path name of the file. For example,

```
% setenv TMPROF_DATAFILE perf.data
```

You can load this file into Prism for examination at a later time; Section 6.7 explains how.

6.5 Displaying Performance Data

To display performance data, the program must have finished execution. Choose `Display Data` from the `Performance` menu. A window appears, containing the data. FIGURE 6-1 shows an example.

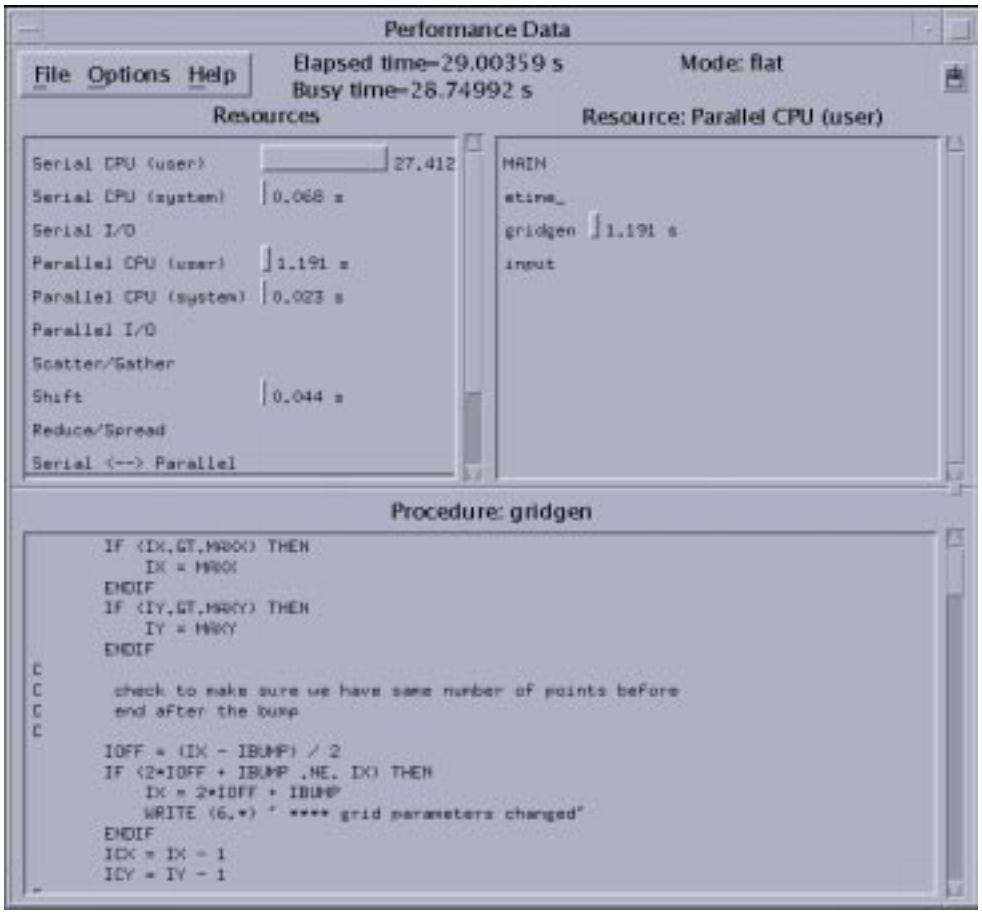

FIGURE 6-1 Performance Data Window

At the top of the window are two values that represent the program's execution time:

- **Elapsed time** represents the actual "wall clock" time from when the program started to when it finished; thus, it is sensitive to effects of timesharing and load.
- **Busy time** represents the time actually spent running the program (specifically, the longest amount of time spent on any one node); this measurement takes into account timesharing, and should be reasonably consistent no matter what the load is on the nodes.

The Performance Data window contains three levels of performance data:

- Performance statistics for the resources that Prism measures, along with totals for each of the two subsystems.
- Per-procedure performance statistics for a specified resource or subsystem. You can choose either flat or call-graph display of these statistics.
- Per-source-line performance statistics for a specified resource and procedure.

All statistics are displayed as histograms in panes within the Performance Data window, along with the amount of time or the percentage of busy time that each histogram bar represents. If the program didn't use the resource, the histogram bar does not appear. (Occasionally, however, a resource will show a utilization of 0% because of rounding.)

By default, the window displays the number of seconds used by the resource next to each histogram bar. Choose Units from the Options menu to change this. You have these choices:

- Choose Seconds (the default) to display the actual time, in seconds, that the histogram bar represents.
- Choose Microseconds to display the actual time in microseconds.
- Choose Utilization to display the percentage of the total busy time that the histogram bar represents.

In all cases, the data represents the longest amount of time spent on the resource by any process.

Once collected, performance data is retained until you load another program (whether or not you leave collection on) or until you re-execute the currently loaded program with collection on.

Choose Close from the File menu to close the Performance Data window.

6.5.1 The Resources Pane

The Resources pane within the Performance Data window displays histogram bars showing a program's use of the measured resources. For each resource, Prism displays the maximum utilization of the resource across all processes that are part of the program.

You can use the Sort By selection from the Options menu to determine the order in which the resources are displayed.

- Choose Name (the default) to display the resource utilizations by category.
- Choose Time to display the resources in order from the highest utilization (at the top) to the lowest.

The Resources pane provides this data:

- **Serial user CPU** – This is CPU time used by the serial portion of the program.
- **Serial system CPU** – This is CPU time used by the operating system on behalf of the serial portion of the program.
- **Serial I/O** – This is the time spent doing I/O during the serial portion of the program.
- **Parallel user CPU** – This is CPU time used by the parallel (distributed) portion of the program.
- **Parallel system CPU** – This is CPU time used by the operating system on behalf of the parallel (distributed) portion of the program.
- **Parallel I/O** – This is the time spent doing I/O during the parallel (distributed) portion of the program.
- **Scatter/Gather** – This is the time that the program spent in scatter/gather communication; for example, time spent in HPF routines using vector subscripts.
- **Shift** – This is the time that the program spent in shift communication (also referred to as *grid* communication); for example, time spent in calls to HPF functions such as `CSHIFT`, and `OSHIFT`.
- **Reduce/Spread** – This is the time that the program spent doing data reductions; for example, time spent in calls to HPF functions such as `SUM`, `PRODUCT`, `ALL`, `ANY`, and `SPREAD`.
- **Serial<-->Parallel** –This is the time that the serial and parallel portions of the program spent in sending data between serial and parallel arrays in HPF.
- **Code not profiled** – This is the time spent by routines that weren't compiled with the `-tmprofile` option. If the routine had been compiled with `-tmprofile`, this time would be allocated to one of the other resources.

- **Serial Total** and **Parallel Total** – The totals of these resources. (These may provide a different overall total from the total busy time because the Resources pane lists the maximum usage of each resource by any one process, whereas busy time reflects the largest amount of time spent in the entire program by any one process.)

6.5.2 The Procedures Pane

The pane titled Resource: *name* in the Performance Data window displays histograms showing the utilization of a specific resource or subsystem by each procedure in a program; this is the *Procedures* pane. You choose the resource by left-clicking on it in the Resources pane. By default, the most-used resource appears in the Procedures pane. The name of the resource appears in the title of the pane.

Use the Mode selection from the Options menu to choose how you want to display the procedure data:

- Choose Call Graph to display the dynamic call graph of the procedures.
- Choose Flat (the default) to display a list of all procedures in the program and their use of the resource.

In flat mode, the Procedures pane displays a list of all procedures in the program and each one's total use of the selected resource. This is useful for determining which procedures are consuming most of the time for the resource. The Procedures pane in FIGURE 6-1 shows the data in flat mode.

Note – Data for the `Code not profiled` resource is not available in flat mode.

In call-graph mode, you see which procedures call which other procedures, and the use of the selected resource for each individual call. This gives a more detailed picture of the program's behavior. FIGURE 6-2 shows the call-graph display for the data shown in the Procedures pane in FIGURE 6-1. Note in FIGURE 6-2 that the time allocated to the `MAIN` routine includes the time spent in the routines that it calls.

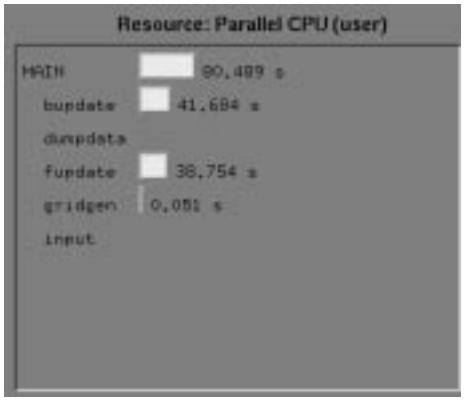

FIGURE 6-2 Call-Graph display

To navigate down through the call graph, click anywhere on the line that lists a procedure (other than the procedure at the top); the display changes to show this procedure at the top, with the procedures it calls below it. Thus, in call-graph mode, the Procedures pane at any one time shows two levels of the call graph.

To move up through the call graph, click on the top procedure in the display; the display changes to show the caller of this procedure at the top, with the procedures it calls beneath it.

As with the Resources pane, you can use the Sort By selection from the Options menu to arrange the procedures in the Procedures pane.

- Choose **Time** (the default) to list procedures according to their use of the resource, from most to least.
- Choose **Name** to arrange the procedures by category.

In call-graph mode, the sorting applies only to the children of the calling procedure; the calling procedure is always at the top of the display.

If a routine is not compiled with the `-tmprofile` option, Prism will display data only for the `Unprofiled` resource.

6.5.3 The Source-Lines Pane

The pane titled Procedure: *name* displays performance data associated with each source line in a procedure; this is the *Source-Lines* pane. Choose the procedure by left-clicking on the line for the procedure in the Procedures pane; by default, Prism displays the source code for the procedure that has the highest utilization of the most-used resource. The resource for which the data is shown is the one displayed in the Procedures pane.

For Sun HPF programs, Prism actually calculates performance data at the level of *basic blocks*. These basic blocks can include one or more lines of source code; the lines are not necessarily contiguous. Prism allocates the amount of time spent in a basic block equally to each line in the block. In general, this will give an accurate picture of each line's contribution to the overall time spent in the basic block. It is possible, however, that the data may be misleading. To get a more accurate picture of per-line data, compile with the `-g` switch in addition to `-tmprofile`. This produces unoptimized code, however, and overall performance will be much worse.

If a routine is not compiled with the `–tmprofile` option, source-line data is not available.

6.5.4 Displaying Performance Data in the Command Window

To display an ASCII version of the performance data, issue the `perf` command from the command window. As with other commands, you can redirect output to a file by using the syntax *@ filename*. This is useful if you are using Prism with the commands-only option, or if you want to study the data at a later time when you don't have a graphical interface available.

By default the `perf` command displays actual times, in seconds, for resources. Use the `util` argument to display utilization percentages.

6.6 Interpreting the Data

Prism's performance data gives you a picture of how your program uses system resources. You will want to use this information to try to improve the program's performance. The key to improving performance is to find the *bottlenecks* in the program—the procedures, and the source lines within the procedures, whose use of a particular resource has the greatest impact on how long the program takes to complete.

To help you in this analysis, Prism provides performance tips. To display this information, choose Tip from the Performance menu, or issue the command `perftip`. You can use these performance tips, or you can analyze the data on your own, to isolate the bottlenecks in your program. Following this procedure provides the best method for interpreting the performance data.

Ask these questions to isolate the bottlenecks in your program:

- Which resource has the highest usage? If Scatter/Gather communication is the most-used resource, then you will obtain the greatest performance gains by reducing the use of this resource.
- Which procedure uses this resource most heavily? This tells you where you will have the biggest payoff when attempting to reduce the use of the most heavily used resource.
- Which source lines within this procedure use this resource most heavily? Finally, going to the source-line level isolates the specific lines of code that have the greatest effect on performance.

When you first display data for a program, by default the Performance `Data` window displays the most-used resource and the procedure that uses this resource the most; this helps you analyze your data quickly.

6.7 Re-using Performance Data Files

You can save performance data you have collected for a program in a file; you can later load this file into Prism and re-display the data. This lets you look at the progression of performance analyses as you work on your program. It is also useful if you do your original data collection outside of Prism or in commands-only Prism, and later want to look at your data in the graphical version.

▼ To Save and Load Performance Data Files

1. **Collect the data as you normally do (that is, turn collection on and run the program to completion).**
2. **Choose Save Data from the Performance menu. (Alternatively, you can choose Display Data from the Performance menu to display the Performance Data window, then choose Save Data from the File menu in this window.)**

   A dialog box appears; in it, specify the name of the file in which you want to save the data. If you don't supply a complete path name, the file name is interpreted relative to the directory from which you started Prism. The data is then saved in this file.

   Alternatively, you can issue the `perfsave` command from the command window, specifying the name of the file in which the data is to be saved.

   When you want to look at the data again, choose Load Data from the Performance menu (or from the File menu in the Performance Data window). A file-selection dialog box is displayed, from which you choose the file in which you saved the data. The data is then reloaded. If no program is loaded at the time, Prism loads the corresponding executable program; if another program is loaded, Prism displays a dialog box and asks if you want to load the program associated with the

performance data. If you don't, the usefulness of the performance data will be limited, since Prism will incorrectly associate the data with the procedures and source lines of the program that is loaded.

Alternatively, you can issue the `perfload` command from the command window, specifying the name of the file in which the data was saved.

Note these points in saving and loading performance data:

- The performance data is associated with a specific version of the program. If you modify the program, Prism will not be able to load the version for which the data was collected. (It prints a warning when it detects that its performance data file is out of date.) Therefore, if you want to use this feature to maintain a historical record of your attempts at improving a program's performance, you should rename the program whenever you change it, and save the earlier versions along with their performance data files.
- You can display only one set of performance data at a time within Prism. Therefore, if you want to compare data from different versions of a program on-screen, you have to run multiple instances of Prism.

CHAPTER 7

Editing and Compiling Programs

You can edit and compile source code by invoking the appropriate utilities from Prism.

See the following sections to learn:

- How to edit source code — *Section* 7.1.
- How to use the Solaris make utility from within Prism to compile and link source code — *Section* 7.2.

7.1 Editing Source Code

Prism provides an interface to the editor of your choice. You can use this editor to edit source code (or anything else).

To call the editor from within Prism:

- **From the menu bar** – From the Utilities menu, choose the Edit selection.
- **From the command window** – On the command line, issue the command `edit`.

You can specify which editor Prism is to call by using the Customize utility to set a Prism resource; see Section 9.3. If this resource has no setting, Prism uses the setting of your `EDITOR` environment variable. Otherwise, Prism uses a default editor, as listed in the Customize window.

The editor is invoked on the current file, as displayed in the source window. If possible, the editor is also positioned at the current execution point, as seen in the source window; this depends on the editor.

If you issue the `edit` command from the command window, you can specify a file name or a function name, and the editor will be invoked on the specified file or function.

After the editor has been created, it runs independently. This means that changes you make in the current file are not reflected in the source window. To update the source window, you must recompile and reload the program. You can do this using the Make selection from the Utilities menu, as described below.

7.2 Using the `make` Utility

Prism provides an interface to the standard Solaris tool `make`. The `make` utility lets you automatically recompile and relink a program that is broken up into different source files. See your Solaris documentation for an explanation of `make` and makefiles.

7.2.1 Creating the Makefile

Create the makefile as you normally would. Within Prism, you can choose the `Edit` selection from the Utilities menu to bring up a text editor in which you can create the file; see Section 7.1.

7.2.2 Using the Makefile

After you have made changes in your program, you can run `make` to update the program.

Prism uses the standard Solaris `make` utility, `/usr/ccs/bin/make`, unless you specify otherwise. You do this by using the `Customize` utility to change the setting of a Prism resource; see Section 9.3.

- **To run `make` from the menu bar** – From the Utilities menu, choose the Make selection. A window appears; FIGURE 7-1 is an example.

FIGURE 7-1 The `make` Window

The window prompts for the names of the makefile, the target file(s), the directory in which the makefile is located, and other arguments to `make`. If a file is loaded, its name is in the Target box, and the directory in which it is located is in the Directory box; you can change these if you like.

If you leave the Makefile or the Target box empty, `make` uses a default. See your Solaris documentation for a discussion of these defaults. If you leave the Directory box empty, `make` looks for the makefile in the directory from which you started Prism.

You can specify any standard `make` arguments in the Other Args box.

The dialog box also asks if you want to reload after the make. Answering Yes (the default) automatically reloads the newly compiled program into Prism if the make is successful. If you answer No, the program is not reloaded.

To cancel the make while it is in progress, click on the Cancel button. If a make is not in progress, clicking on Cancel closes the window.

The output from `make` is displayed in the box at the bottom of the Make window. Subsequent makes use the same window, unless you start a new make while a previous make is still in progress.

From the command window – Issue the `make` command on the command line. You can specify any arguments that are valid in the Solaris version of `make`.

CHAPTER 8

Getting Help

This chapter describes how to obtain information about Prism and other Sun products available at your site.

See the following sections to learn

- How to obtain help about Prism — *Section* 8.1.
- How to obtain other on-line documentation — *Section* 8.2.

8.1 Getting Help

There are several ways in which you can get help in Prism:

- The Help menu in the menu bar provides help on several major topics. It includes the Help Index, which gives in-depth information about all aspects of Prism. See Section 8.1.2.
- The Help selection in menus and the Help button in windows and dialog boxes provide instructions for using these screen areas. Pressing the F1 key in a window or dialog box also displays a help screen.
- Command-line help provides information about commands you can issue from the command window.

8.1.1 Using the Help System

Prism displays its help files via your World Wide Web browser. The default browser is Netscape™, although your system administrator can change this. To specify which browser you want to use for graphical Prism, set the Prism resource `Prism.helpBrowser` to the executable name of the browser; see Section 9.4.11.

If you don't have a browser running, Prism starts one. If you have a browser currently running as you use Prism, by default Prism displays the help information in that browser. You can change this behavior via the `Prism.helpUseExisting` resource; once again, see Section 9.4.11.

8.1.2 Choosing Selections from the Help Menu

The Help menu provides information in a variety of ways:

- Choose Index to display an index of entries. Click on an entry to display the section of the *Prism User's Guide* in which the entry is discussed.
- Choose Using Help to display an overview of the Help system.
- Choose Overview to display an overview of the features of Prism.
- Choose Glossary to display a list of terms used in Prism. You can click on a term to find out more about it.
- Choose Commands Reference to display a list of Prism commands. You can click on a command's link marker to obtain its reference description.
- Choose Tutorial to display a tutorial that will teach you the basics of Prism.

8.1.3 Getting Help on Using the Mouse

Some Prism windows include an icon of a mouse,

Click on this icon to display information about using the mouse in the window.

8.1.4 Obtaining Help from the Command Window

Use the `help` command to obtain help from the command window. Issuing the command

```
help commands
```

displays a list of Prism commands and editing key combinations. Issuing `help` with the name of a command as an argument displays help on that command. Issuing `help` with no arguments displays a brief message about how to use command-line help.

8.2 Obtaining Online Documentation

Prism documentation is available both in print and Sun AnswerBook forms. Prism also comes with a Solaris-style manual page.

8.2.1 Viewing Manual Pages

To obtain a manual page, choose the Man Pages selection from the Doc menu. This brings up `xman`, a standard X program for viewing manual pages; `xman` operates independently of Prism.

Help for `xman` appears in the `xman` window, as shown in FIGURE 8-1. You can use `xman` to view any Solaris manual pages available on your Sun system.

Note – If `xman` is not available on your system, you will not be able to use this feature.

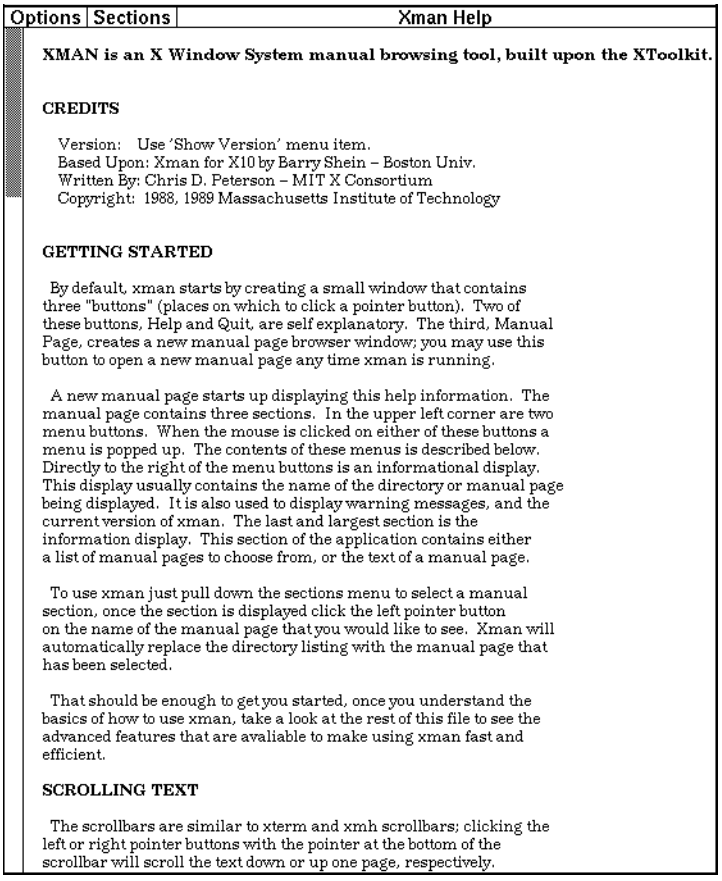

FIGURE 8-1 xman Window

CHAPTER 9

Customizing Prism

This chapter discusses ways in which you can change various aspects of Prism's appearance and the way Prism operates.

See the following sections to learn:

- How to use the tear-off region — *Section* 9.1.
- How to set up alternative names for commands and variables — *Section* 9.2.
- How to change Prism defaults by using the Customize utility — *Section* 9.3.
- How to change Prism defaults in your X resource database — *Section* 9.4.
- How to initialize Prism — *Section* 9.5.

9.1 Using the Tear-Off Region

You can place frequently used menu selections and commands in the tear-off region below the menu bar; in the tear-off region, they become buttons that you can click on to execute functions. FIGURE 9-1 shows the buttons that are there by default.

FIGURE 9-1 The Tear-Off Region.

Putting menu selections and commands in the tear-off region lets you access them without having to pull down a menu or issue a command from the command line.

Changes you make to the tear-off region are saved when you leave Prism; see Section 9.3.3.

9.1.1 Adding Menu Selections to the Tear-Off Region

- **From the menu bar** – To add a menu selection to the tear-off region, first enter tear-off mode by choosing Tear-off from the Utilities menu. A dialog box appears that describes tear-off mode; see FIGURE 9-2.

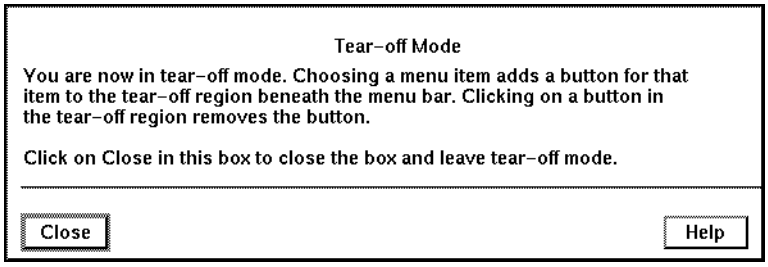

FIGURE 9-2 Tear-Off Region Dialog Box

While the dialog box is on the screen, choosing any selection from a menu adds a button for this selection to the tear-off region. Clicking on a button in the tear-off region removes that button. If you fill up the region, you can resize it to accommodate more buttons. To resize the region, drag the small resize box at the bottom right of the region.

Click on Close or press the Esc key while the mouse pointer is in the dialog box to close the box and leave tear-off mode.

When you are not in tear-off mode, clicking on a button in the tear-off region has the same effect as choosing the equivalent selection from a menu.

- **From the command window** – Use the `tearoff` and `untearoff` commands from the command window to add menu selections to and remove them from the tear-off region. Put the selection name in quotation marks; case doesn't matter, and you can omit spaces and the ellipsis (...) that indicates the selection displays a window or dialog box. If the selection name is ambiguous, put the menu name in parentheses after the selection name. For example,

  tearoff "print (events)"

  adds a button for the Print selection from the Events menu to the tear-off region.

9.1.2 Adding Prism Commands to the Tear-Off Region

To add a Prism command to the tear-off region, issue the `pushbutton` command, specifying the label for the tear-off button and the command it is to execute. The label must be a single word. The command can be any valid Prism command, along with its arguments. For example,

```
pushbutton printa print a on dedicated
```

adds a button labeled `printa` to the tear-off region. Clicking on it executes the command `print a on dedicated`.

To remove a button created via the `pushbutton` command, you can either click on it while in tear-off mode, or issue the `untearoff` command as described above.

9.2 Creating Aliases for Commands and Variables

Prism provides commands that let you create alternative names for commands, variables, and expressions.

Use the `alias` command to set up an alternative name for a Prism command. For example,

```
alias ni nexti
```

makes `ni` an alias for the `nexti` command. Prism provides some default aliases for common commands. Issue `alias` with no arguments to display a list of the current aliases. Issue the `unalias` command to remove an alias. For example,

```
unalias ni
```

removes the alias created above.

Use the `set` command to set up an alternative name for a variable or expression. For example,

```
set alan = annoyingly_long_array_name
```

abbreviates the annoyingly long array name to `alan`. You can use this abbreviation subsequently in your program to refer to this variable. Use the `unset` command to remove a setting. For example,

```
unset alan
```

removes the setting created above.

Changes you make via `alias` and `set` last for your current Prism session. To make them permanent, you can add the appropriate commands to your `.prisminit` file; see Section 9.5.

9.3 Using the Customize Utility

Many aspects of Prism's behavior and appearance—for example, the colors it displays on color workstations, and the fonts it uses for text—are controlled by the settings of *Prism resources*. The default settings for many of these resources appear in the file `Prism` in the X11 `app-defaults` directory for your system. Your system administrator can change these system-wide defaults. You can override these defaults in two ways:

- For many of them, you can use the Customize selection from the Utilities menu to display a window in which you can change the settings. This section describes this method.
- A more general method is to add an entry for a resource to your X resource database, as described in the next section. Using the `Customize` utility is much more convenient, however.

Choosing Customize from the Utilities menu displays the window shown in FIGURE 9-3.

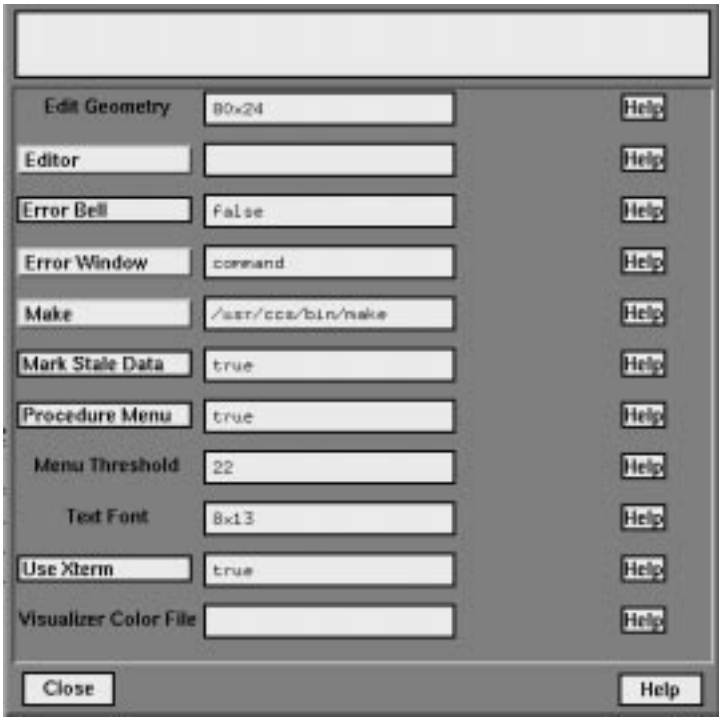

FIGURE 9-3 Customize Window.

9.3.1 How to Change a Setting

On the left of the Customize window are the names of the resources. Next to each resource is a text-entry box that contains the resource's setting (if any). To the right of the fields are Help buttons. Clicking on a Help button or anywhere in the text-entry field displays help about the associated resource in the box at the top of the window.

The way you set a value for a resource differs depending on the resource:

- For Edit Geometry, Menu Threshold, Text Font, and Visualizer Color File, you enter the setting in the resource's text-entry box.
- For Editor, Error Window, and Make, you can left-click on the button labeled with the resource's name. This displays a menu of choices for the resource. Clicking on one of these choices displays it in the resource's text-entry box. For Editor and Make, you can also enter the setting directly in the text-entry box.

- For Error Bell, Procedure Menu, Mark Stale Data, and Use Xterm, there are only two possible settings, true and false; clicking on the button labeled with the resource's name toggles the current setting.

Whenever you make a change in a text-entry box, Apply and Cancel buttons appear to the right of it. Click on Apply to save the new setting; it takes effect immediately. Click on Cancel to cancel it; the setting changes back to its previous value.

Click on Close or press the Esc key to close the Customize window.

9.3.2 Resources

Edit Geometry – Use this resource to specify the X geometry string for the editor created by the Edit and Email selections from the Utilities menu. The geometry string specifies the number of columns and rows, and optionally the left and right offsets from the corner of the screen. The Prism default is 80x24 (that is, 80 rows and 24 columns). See your X documentation for more information on X geometries.

Editor – Use this resource to specify the editor that Prism is to invoke when you choose the Edit or Email selection from the Utilities menu. Click on the Editor box to display a menu of possible choices. If you leave this field blank, Prism uses the setting of your `EDITOR` environment variable to determine which editor to use.

Error Bell – Use this resource to specify how Prism is to signal errors. Choosing true tells Prism to ring the bell of your workstation. Choose false (the Prism default) to have Prism flash the screen instead.

Error Window – Use this resource to tell Prism where to display Prism error messages. Choose command (the Prism default) to display them in the command window. Choose dedicated to send the messages to a dedicated window; the window will be updated each time a new message is received. Choose snapshot to send each message to a separate window.

Make – Use this resource to tell Prism which `make` utility to use when you choose the Make selection from the Utilities menu. The Prism default is the standard Solaris `make` utility, `/usr/ccs/bin/make`. Click on the Make box to display a menu of possible choices.

Mark Stale Data – Use this resource to tell Prism how to treat the data in a visualizer that is out-of-date (because the program has continued execution past the point at which the data was displayed). Choose `true` (the default) to have Prism draw diagonal lines over the data; choose `false` to leave the visualizer's appearance unchanged.

Procedure Menu – Use this resource to specify whether a menu is to be displayed when you set a breakpoint in a Sun HPF generic procedure. If you choose true (the default), a menu of possible procedures is displayed, from which you can choose the procedure(s) in which the breakpoint is to be set. Choose false if you want to set the breakpoint automatically in all the generic procedures.

Menu Threshold – Use this resource to specify the maximum number of procedures that are to be displayed in a menu when you perform an action (for example, setting a breakpoint) on a Sun HPF generic procedure. The default is 22. Enter 0 to indicate that there should be no maximum. If the number of procedures exceeds the specified threshold, you are prompted to either enter the procedure name or display the menu.

Text Font – Use this resource to specify the name of the X font that Prism is to use in displaying the labels of histogram bars and text in visualizers. The default, 8x13, is a 12-point fixed-width font. To list the fonts available on your system, issue the Solaris command `xlsfonts`. Specifying a font much larger than the default can cause display problems, because Prism doesn't resize windows and buttons to accommodate the larger font.

Use Xterm – Use this resource to tell Prism what to do with the I/O of a program. Specify true (the Prism default) to tell Prism to create an Xterm in which to display the I/O. Specify false to send the I/O to the Xterm from which you started Prism.

Visualizer Color File – Use this resource to tell Prism the name of a file that specifies the colors to be used in colormap visualizers. If you leave this field blank, Prism uses gray for elements whose values are not in the context you specify; for elements whose values are in the context, it uses black for values below the minimum, white for values above the maximum, and a smooth spectral map from blue to red for all other values.

The file must be in ASCII format. Each line of the file must contain three integers between 0 and 255 that specify the red, green, and blue components of a color.

The first line of the visualizer color file contains the color that is to be displayed for values that fall below the minimum you specify in creating the visualizer. The next-to-last line contains the color for values that exceed the maximum. The last line contains the color used to display the values of elements that are not in the context specified by the user in a `where` statement. Prism uses the colors in between to display the values falling between the minimum and the maximum. For example,

```
0                   0     0
255                 0     0
255                 255   0
0                   255   0
0                   255   255
0                   0     255
255                 0     255
255                 255   255
100                 100   100
```

Like the default settings, this file specifies black for values below the minimum, white for values above the maximum, and gray for values outside the context. But the file reverses the default spectral map for other values: from lowest to highest, values are mapped red-yellow-green-cyan-blue-magenta.

9.3.3 Where Prism Stores Your Changes

Prism maintains a file called `.prism_defaults` in your home directory. In it, Prism keeps

- Changes you make to Prism via the `Customize` utility
- Changes you make to the tear-off region
- Changes you make to the size of the panes within the main Prism window

Do not attempt to edit this file; make all changes to it through Prism itself. If you remove this file, you get the default configuration the next time you start Prism.

9.4 Changing Prism Defaults

As mentioned in the previous section, you can change the settings of many Prism resources either by using the `Customize` utility or by adding them to your X resource database. This section describes how to add a Prism resource to your X resource database. An entry is of the form

resource-name: value

where *resource-name* is the name of the Prism resource, and *value* is the setting. TABLE 9-1 lists the Prism resources.

TABLE 9-1 Prism Resources

Resource	Use
`Prism.cppPath`	Specifies the path to your C preprocessor.
`Prism.dialogColor`	Specifies the color for dialog boxes.
`Prism.editGeometry`	Specifies the size and placement of the editor window.
`Prism.editor`	Specifies the editor to use.
`Prism.errorBell`	Specifies whether the error bell is to ring.
`Prism.errorwin`	Specifies the window to use for error messages.

TABLE 9-1 Prism Resources

Resource	Use
`Prism*fontList`	Specifies the font for labels, menu selections, etc.
`Prism.helpBrowser`	Specifies the browser to use for displaying help.
`Prism.helpUseExisting`	Specifies whether to use a currently running browser for displaying help.
`Prism.mainColor`	Specifies the main background color for Prism.
`Prism.make`	Specifies the `make` utility to use.
`Prism.markStaleData`	Specifies how Prism is to mark stale data in visualizers.
`Prism.procMenu`	Specifies whether a menu is displayed when setting a breakpoint in a Sun HPF generic procedure.
`Prism.procThresh`	Changes the maximum number of specific procedures automatically shown when performing an action on a Sun HPF generic procedure.
`Prism.spectralMapSize`	Specifies the size of the default spectral color map for color visualizers.
`Prism.textBgColor`	Specifies the background color for widgets containing text.
`Prism.textFont`	Specifies the text font to use for certain labels.
`Prism.textManyFieldTranslations`	Specifies the keyboard translations for dialog boxes that contain several text fields.
`Prism.textMasterColor`	Specifies the color used to highlight the master pane in a split source window.
`Prism.textOneFieldTranslations`	Specifies the keyboard translations for dialog boxes that contain one text field.

TABLE 9-1 Prism Resources

Resource	Use
`Prism.useXterm`	Specifies whether to use a new Xterm for I/O.
`Prism.vizColormap`	Specifies the colors to be used in colormap visualizers.
`Prism*XmText.fontList`	Specifies the text font to use for most running text.

Note that the defaults mentioned in the sections below are the defaults for Prism as shipped; your system administrator can change these in Prism's file in your system's `app-defaults` directory.

Note also that commands-only Prism is not aware of the settings of any Prism resources, unless they are contained in Prism's `app-defaults` file. This matters only for the resources `Prism.cppPath` and `Prism.pathMap`.

9.4.1 Adding Prism Resources to the Resource Database

The X resource database keeps track of default settings for programs running under X. Use the `xrdb` program to add a Prism resource to this database. An easy way to do this is to use the `-merge` option and to specify the resource and its setting from the standard input. For example, the following command specifies a default editor (the resource is described below):

```
% xrdb -merge
  Prism.editor: emacs
  Ctrl-d
```

Type Ctrl-d to signal that there is no more input. Note that you must include the `-merge` option; otherwise, what you type replaces the contents of your database. The new settings take effect the next time you start Prism.

Another way to add your changes is to put them in a file, then merge the file into the database. For example, if your changes are in `prism.defs`, you could issue this command:

```
% xrdb -merge prism.defs
```

Consult your X documentation for more information about `xrdb`.

9.4.2 Specifying the Editor and Its Placement

Use the `Prism.editor` resource to specify the editor that Prism is to invoke when you choose the Edit or Email selection from the Utilities menu (or issue the corresponding commands).

Use the resource `Prism.editGeometry` to specify the X geometry string for the editor created by the Edit selection from the Utilities menu. The geometry string specifies the number of columns and rows, and the left and right offsets from the corner of the screen.

You can also change the settings of these resources via the `Customize` utility; see Section 9.3 for more information.

9.4.3 Specifying the Window for Error Messages

Use the `Prism.errorwin` resource to specify the window to which Prism is to send error messages. Predefined values are command, dedicated, and snapshot. You can also specify your own name for the window.

You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

9.4.4 Changing the Text Fonts

You may need to change the fonts Prism uses if, for example, its fonts aren't available on your system. Use the resources described below to do this. To list the names of the fonts available on your system, issue the Solaris `xlsfonts` command. You should try to substitute a font that is about the same size as the Prism default; substituting a font that is much larger can cause display problems, since Prism does not resize windows and buttons to accommodate the larger font.

Use the `Prism.textFont` resource to specify the font that Prism is to use in displaying the labels of histograms and text in visualizers. By default, Prism uses a 12-point fixed-width font for this text.

You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

Use the `Prism*XmText.fontList` resource to change the font used to display most of the running text in Prism, such as the source code in the source window. By default, Prism uses a 12-point fixed-width font for this text.

Use the `Prism*fontList` resource to change the font used for everything else (for example, menu selections, pushbuttons, and list items). By default, Prism uses a 14-point Helvetica font for this text.

9.4.5 Changing Colors

Prism provides several resources for changing the default colors it uses when it is run on a color workstation.

9.4.5.1 Changing the Colors Used for Colormap Visualizers

Use the `Prism.vizColormap` resource to specify a file that contains the colors to be used in colormap visualizers. You can also change the setting of this resource via the `Customize` utility; see Section 9.3. See Section 9.3.2 for a discussion of how to create a visualizer color file.

Use the resource `Prism.spectralMapSize` to specify how large the default spectral color map is to be for colormap visualizers. The default is 100 entries. You would typically use this resource to specify fewer entries, if this number causes problems on your workstation. To set the default to 50, for example, set the resource in your X resource database as follows:

```
Prism.spectralMapSize: 50
```

9.4.5.2 Changing Prism's Standard Colors

Use the `Prism.dialogColor` resource to change the background color of dialog boxes.

Use the `Prism.textBgColor` resource to change the background color for text in buttons, dialog boxes, etc. Note that this setting overrides the setting of the X toolkit `-bg` option.

Use the `Prism.textMasterColor` resource to change the color used to highlight the master pane when the source window is split.

Use the `Prism.mainColor` resource to change the color used for just about everything else.

The defaults are:

```
Prism.dialogColor: Thistle
Prism.textBgColor: snow2
Prism.textMasterColor: black
Prism.mainColor: light sea green
```

9.4.6 Changing Keyboard Translations

You can change the keys and key combinations that Prism translates into various actions. In general, doing this requires an understanding of X and Motif programming. You may be able to make some changes, however, by reading this section and studying the defaults in Prism's file in your system's `app-defaults` directory.

9.4.6.1 Changing Keyboard Translations in Text Widgets

Use the `Prism.textOneFieldTranslations` resource to change the default keyboard translations for dialog boxes that contain only one text field. Its default definition is:

```
Prism.textOneFieldTranslations: \
<Key>osfDelete: delete-previous-character() \n\
    <Key>osfBackSpace: delete-previous-character() \n\
          Ctrl<Key>u: erase_to_beginning() \n\
          Ctrl<Key>k: erase_to_end() \n\
          Ctrl<Key>d: delete_char_at_cursor_position() \n\
          ctrl<Key>f: move_cursor_to_next_char() \n\
          Ctrl<Key>h: move_cursor_to_prev_char() \n\
          Ctrl<Key>b: move_cursor_to_prev_char() \n\
          Ctrl<Key>a: move_cursor_to_beginning_of_text() \n\
          Ctrl<Key>e: move_cursor_to_end_of_text()
```

(The definitions with `osf` in them are special Motif keyboard symbols.)

Use the `Prism.textManyFieldTranslations` resource to change the default keyboard translations for dialog boxes that contain several text fields. Its default definition is:

```
Prism.textManyFieldTranslations: \
    <Key>osfDelete: delete-previous-character() \n\
    <Key>osfBackSpace: delete-previous-character() \n\
    <Key>Return: next-tab-group() \n\
    <Key>KP_Enter: next-tab-group() \n\
          Ctrl<Key>u: erase_to_beginning() \n\
          Ctrl<Key>k: erase_to_end() \n\
          Ctrl<Key>d: delete_char_at_cursor_position() \n\
          Ctrl<Key>f: move_cursor_to_next_char() \n\
          Ctrl<Key>h: move_cursor_to_prev_char() \n\
          Ctrl<Key>b: move_cursor_to_prev_char() \n\
          Ctrl<Key>a: move_cursor_to_beginning_of_text() \n\
          Ctrl<Key>e: move_cursor_to_end_of_text()
```

If you make a change to any field in one of these resources, you must copy all the definitions.

9.4.6.2 Changing General Motif Keyboard Translations

Prism uses the standard Motif translations that define the general mappings of functions to keys. They are shown below.

```
*defaultVirtualBindings:  \
  osfActivate :      <Key>Return \
  osfAddMode :       Shift <Key>F8 \n
  osfBackSpace :     <Key>BackSpace \n\
  osfBeginLine :     <Key>Home \n\
  osfClear :         <Key>Clear \n\
  osfDelete :        <Key>Delete \n\
  osfDown :          <Key>Down \n\
  osfEndLine :       <Key>End \n\
  osfCancel :        <Key>Escape \n\
  osfHelp :          <Key>F1 \n\
  osfInsert :        <Key>Insert \n\
  osfLeft :          <Key>Left \n\
  osfMenu :          <Key>F4 \n\
  osfMenuBar :       <Key>F10 \n\
  osfPageDown :      <Key>Next \n\
  osfPageUp :        <Key>Prior \n\
  osfRight :         <Key>Right \n\
  osfSelect :        <Key>Select \n\
  osfUndo :          <Key>Undo \n\
  osfUp  :           <Key>Up
```

To change any of these, you must edit its entry in this resource. For example, if your keyboard doesn't have an F10 key, you could edit the osfMenuBar line and substitute another function key.

Note these points in changing this resource:

- All entries in the resource must be included in your resource database if you want to change any of them; otherwise the omitted entries are undefined.
- The entries in this resource apply to all Motif-based applications. If you want your changes to apply only to Prism, change the first line of the resource to `Prism*defaultVirtualBindings`.

9.4.7 Changing the Xterm to Use for I/O

By default, Prism creates a new Xterm for input to and output from a program. Set the `Prism.useXterm` resource to `false` to tell Prism not to do this. Instead, I/O will go to the Xterm from which you invoked Prism. You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

9.4.8 Changing the Way Prism Signals an Error

By default, Prism flashes the command window when there is an error. Set the resource `Prism.errorBell` to `true` to tell Prism to ring the bell of your workstation instead. You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

9.4.9 Changing the `make` Utility to Use

By default, Prism uses the standard Solaris `make` utility, `/bin/make`. Use the resource `Prism.make` to specify the path name of another version of `make` to use. You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

9.4.10 Changing How Prism Treats Stale Data in Visualizers

By default, Prism prints diagonal lines over data in visualizers that has become "stale" because the program has continued execution from the spot where the data was collected. Set the resource `Prism.markStaleData` to `false` to tell Prism not to draw these diagonal lines. You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

9.4.11 Specifying the Browser to Use for Displaying Help

There are several resources you can use to affect the way help is displayed.

By default, graphical Prism uses the Netscape browser to display help information; see Section 8.1.1. Set the `Prism.helpBrowser` resource to the executable name of another browser to start; the name must be on your path. Graphical Prism supports Mosaic and Netscape browsers. You can include in the setting any browser-specific options that you want passed to the browser when Prism starts it up. (Note that these options do not take effect if Prism uses an existing browser; see below.)

If you already have a browser running when you request help from Prism, by default Prism displays the help information in this browser. Set the resource `Prism.helpUseExisting` to `false` if you want Prism to start a new browser. Set it to `true` to return to the default behavior.

9.4.12 Changing the Way Prism Handles Sun HPF Generic Procedures

There are two resources you can use to change the way Prism handles Sun HPF generic procedures.

By default, Prism displays a menu (in commands-only Prism) or a dialog box when you attempt to set a breakpoint in a Sun HPF generic procedure. Set the Prism resource `Prism.procMenu` to `false` to specify that Prism is to set the breakpoint in every one of these procedures, without displaying a menu or dialog box. You can also change the setting of this resource via the `Customize` utility; see Section 9.3.

By default, commands-only Prism displays a maximum of 22 procedures in a menu when you attempt to perform an action (like setting a breakpoint) on a Sun HPF generic procedure. If there are more than this number of specific procedures, Prism asks you whether you want to specify the name of a specific procedure or to view a menu. Use the `Prism.procThresh` resource to specify a different maximum. Set the resource to 0 to specify that there is to be no maximum.

9.5 Initializing Prism

Use the `.prisminit` file to initialize Prism when you start it up. You can put any Prism commands into this file. When Prism starts, it executes these commands, echoing them in the history region of the command window.

When starting up, Prism first looks in the current directory for a file called `.prisminit`. If the file is there, Prism uses it. If the file isn't there, Prism looks for it in your home directory. If the file isn't in either place, Prism starts up without executing a `.prisminit` file.

The `.prisminit` file is useful if there are commands that you always want to execute when starting Prism. For example,

- If you always want to log command output, put a `log` command in the file; see Section 2.7.4.
- If you want to use your own aliases for Prism commands, put the appropriate `alias` commands in the file; see Section 9.2.

Note that you don't need to put `pushbutton` or `tearoff` commands into the `.prisminit` file, because changes you make to the tear-off region are automatically saved when you leave Prism; see Section 9.1.

In the `.prisminit` file, Prism interprets lines beginning with # as comments. If \ is the final character on a line, Prism interprets it as a continuation character.

CHAPTER **10**

MP Prism

MP Prism is the version of Prism used for debugging and visualizing data in message-passing or other multiprocess programs that use the SPMD (single program, multiple data) programming style. This chapter describes how to use MP Prism; it assumes that you are already familiar with Prism when used with scalar programs. (If you aren't familiar with Prism, the chapter points you to the appropriate sections in previous chapters.) It talks only about message-passing programs, but MP Prism works with any multiprocess program, whether or not its processes communicate.

In this chapter, the term *DP Prism* refers to the version of Prism discussed in previous chapters of this manual. DP Prism is used to debug data parallel and serial programs. (When you load a data parallel program into Prism, you view and operate on it as if it were a single program; in fact, these programs are actually running as multiprocess message-passing programs, and MP Prism is operating below the surface. You will generally not be aware of this when interacting with Prism.)

Note – You cannot use MP Prism to debug a message-passing program that is made up of different executables.

See the following sections to learn:

- How to enter MP Prism — see *Section* 10.2.
- How to create and use psets — see *Section* 10.3.
- How to execute programs in MP Prism — see *Section* 10.4.
- About combining data parallel and message-passing code — see *Section* 10.5.
- How to debug in MP Prism — see *Section* 10.6.
- How to visualize data in MP Prism — see *Section* 10.7.
- About customizing MP Prism — see *Section* 10.8.
- About using MP Prism with PVM programs — see *Section* 10.9.
- About using MP Prism with Sun MPI programs — see *Section* 10.10

:

10.1 Overview

In most message-passing programs, each individual processor runs its own copy of the same program. At a given point in the execution of the program, all these processes may be doing the same thing, or (more likely) different subsets of processes may be doing different things.

To debug such a program, you may want to look at the behavior of individual processes or of particular groups of processes. A key concept in MP Prism is the *pset*; a pset (pronounced “pee-set”) is a predefined or user-defined group of processes that you can view and operate on as a single entity. Aggregating processes into psets can help provide you with the correct level of analysis for your program.

In general, MP Prism operates in much the same way as DP Prism. The differences in many cases involve the application of the concept of psets to DP Prism features such as events and visualizers. In addition:

- MP Prism provides a variety of ways of creating and deleting psets. You can use psets to qualify many Prism commands, so that the command applies only to the processes that are members of the pset.
- MP Prism lets you interrupt execution of individual processes or sets of processes, and wait for execution to stop in the processes of a pset.
- In MP Prism you can create events that are tied to specified psets so that, for example, you can set a breakpoint only in certain processes. Prism also provides a *Where* graph that displays a snapshot of the dynamic call graph of the program, for processes that aren’t running.
- MP Prism adds an extra “process” axis to visualizers, so that you can see the values of a variable in each process in a pset.

10.2 Entering MP Prism

See Section 2.2 for basic information about entering Prism.

The major difference between starting Prism to work on a serial program and starting MP Prism is that, with MP Prism, you are actually starting multiple Prism processes, in a client/server model:

- There is one client MP Prism process for each process in the message-passing program. The MP Prism process attaches itself to the message-passing process to collect information about it.
- There is a single server Prism process that communicates with the MP Prism processes and provides the interface to the user. This process is referred to as *Host Prism*.

Typically, Host Prism runs on a node in a shared partition. The multiple MP Prism processes run on the nodes on which the message-passing processes are running, which may be in either a shared or a dedicated partition.

Note that the information in this section also applies to executing data parallel programs under Prism. The major difference is that, when you are executing a data parallel program, the MP Prism user interface is, for the most part, hidden. See Section 10.5 for more information.

10.2.1 Command-Line Options

To enter MP Prism, issue the `prism` command with the `-np` option, specifying the number of client MP Prism processes you want to start. Use the value 0 to specify that you want to run on all available nodes, one process per node.

10.2.2 Methods of Entering

You can specify where you want both Host Prism and the message-passing processes (along with their associated MP Prism processes) to run. You can do this either explicitly or by using defaults.

10.2.2.1 Specifying Where Host Prism Is to Run

If you simply issue the command

```
% prism -np 4
```

Host Prism starts on the node to which you are logged in. If necessary, you can start Prism on a partition other than the one you are logged into, by using tmrun or tmsub.

Using tmrun or tmsub, you can specify where Host Prism is to run the same way you would specify where any Sun HPC Software program is to run, via options to the `tmrun` or `tmsub` command, and by the setting of the `TMRUN_FLAGS` environment variable. For example,

```
% tmrun -p Scylla prism -np 4
```

starts Host Prism on a node in the partition Scylla. If you have set the values of other options via TMRUN_FLAGS, tmrun uses them as well in determining where Host Prism is to run.

See Section 2.2 for more information on starting Prism, and see the *Sun HPC Software User's Guide* for more information on starting Sun HPC programs in general.

10.2.2.2 Specifying Where MP Prism and the Message-Passing Processes Are to Run

There are three ways in which you can specify where message-passing processes, and their associated MP Prism processes are to run, as well as other options associated with program execution:

- Via the setting of the TMRUN_FLAGS environment variable
- Via options to the prism command
- Via options to the tmrunargs command, once you are in Prism

TMRUN_FLAGS – Prism uses the setting of the TMRUN_FLAGS environment variable to determine where the message-passing processes, and their associated MP Prism processes, are to run. Thus, if the default options are

```
-p Charybdis -np 4
```

then four copies of the message-passing program start on nodes of the partition Charybdis.

Note that default settings that are appropriate for Host Prism may be inappropriate for the message-passing processes and their associated MP Prism processes, and vice versa. For example, if the default setting is simply

```
-p Shared
```

then only one copy of the message-passing process starts, on a load-balanced node in the shared partition.

Prism command-line options – If you are in MP Prism, you can also use the Prism options listed below to specify where you want the message-passing processes and their associated MP Prism processes to run.

- Use -p *partition_name* on the prism command line to specify the partition in which the message-passing processes are to run.
- Use the -tmrun option to specify any other tmrun or tmsub options that you want to control the selection of nodes. Enclose the options in quotation marks. For example,

```
% prism -np 4 -tmrun "-W" a.x
```

If the tmrun or tmsub option itself uses quotation marks, refer to the documentation for your shell program for the syntax for handling quotes. For example, using the C shell (csh),

```
% prism -np 2 -tmrun '-R "load1<1"' a.x
```

As with tmrun and tmsub, specifying these options on the prism command line overrides settings of these same options established via the TMRUN_FLAGS environment variable.

Thus, the command

```
% tmrun prism -np 4 mpiprogram
```

starts four message-passing processes, with their associated MP Prism processes. Where it starts them is determined by the TMRUN_FLAGS settings.

The tmrunargs command – Once you have entered Prism, you can issue the tmrunargs command to specify any tmrun or tmsub options that you want to apply to your message-passing program. Prism stores these options, then applies them when you start up a multiprocess program. Specifying the setting of an option via the tmrunargs command overrides the setting of the same option you have established via TMRUN_FLAGS or the prism command line. If it is an option that has otherwise not been specified, it is added to the existing settings.

You can issue tmrunargs in DP Prism. If you use the -np option to specify multiple processes, MP Prism starts up.

To remove any existing tmrun or tmsub options you have specified, issue the command

```
tmrunargs off
```

This removes options you have set via the prism command line and the tmrunargs command, but not via TMRUN_FLAGS.

Issuing the tmrunargs command with no options shows the current tmrun and tmsub options.

10.2.3 Other Options

As with DP Prism, you can specify other options on the prism command line. For example, you can specify the -C option to bring up MP Prism in commands-only mode, or the -CX option (from an Xterm) to bring it up in commands-only mode, but be able to send the output of certain commands to X windows.

If you start the graphical version, MP Prism's main window is virtually the same as DP Prism's. There are two differences:

- There are fields for Current Pset and Current Process in the status region above the source window. These fields are discussed in Sections 10.3.6 and 10.3.7.
- The command-line prompt is different. Once again, see Section 10.3.6.

10.2.4 Attaching

You can attach to running message-passing programs. To do this, specify the *task ID* of the processes (not an individual process ID) on the `prism` command line, after the name of the executable program. For example,

```
% prism -p Dedicated tmmpi t462
```

starts MP Prism on the nodes of the partition Dedicated and attaches to the running processes in task t462. You obtain the task ID by issuing the `tmps` command.

You can also attach to a task from within Prism via the `attach` command; see Section 10.4.1.

If you attach to a program under MP Prism, your task will be automatically detached from MP Prism if you quit or run another program. You can detach from the task by issuing the `detach` command from within Prism.

You can also attach to a single process of a message-passing program by specifying its process ID, just as you do in DP Prism; see Section 2.2.2. If you do this, however, you won't be able to view or debug what is happening in the other processes.

10.3 Using Psets

MP Prism provides you with the capability of viewing your message-passing program at the level of an individual process, a group of processes, or all processes that make up the program. For example, at times it may be useful to look at the status of process 0, because you have reason to believe there is a problem with it. At other times you may want to look at all processes that have encountered an error, or at all processes in which the value of a particular variable exceeds a certain number.

These groups of processes, typically chosen because they have some useful characteristic in common, are referred to as *psets* (pronounced "pee-sets"). MP Prism treats a pset as a unit; for example, you can use the name of a pset as a qualifier for many commands. The command is then executed for each process in the set. For example, you can set a breakpoint that applies only to processes in a specified pset. (See Section 10.3.9 for more information.) In addition, many graphical actions apply only to processes in a pset.

If you don't need to view your program at the level of an individual process or a subset of processes, you can also view its operation on all the processes that make up the message-passing program.

You can view psets in the Psets window, as described in Sections 10.3.1 and 10.3.4.

Note – MP Prism assigns a logical ID number to each process that makes up a message-passing program. For example, in an 8-process message-passing program, the individual processes would be numbered 0-7. These numbers are used to identify processes in psets. Do not confuse these numbers with the Solaris process IDs (pids) assigned by the system to the message-passing processes.

As described in Section 10.3.2, Prism provides predefined psets for certain standard groups of processes; for example, the set of all processes in an error state is a predefined pset. You can also define your own psets, as described in Section 10.3.3; for example, you can define a pset to be those processes in which variable *x* is greater than 0. Section 10.3.5 describes how to delete psets.

If you don't specify a pset as a qualifier to a command (that can take a pset qualifier), the command is executed on the *current* pset; many graphical actions also apply to processes in the current set. The concept of the current pset is described in Section 10.3.6. Section 10.3.7 describes the current process, which is a distinguished process within a pset.

Section 10.3.8 describes the `cycle` pset, which is a predefined pset with special characteristics.

10.3.1 Using the Psets Window

You can use the `Psets` window to view the current status of the processes in your program and to perform many of the actions associated with psets.

To display the window:

- **From the menu bar** – Choose the Psets selection from the Debug menu.
- **From the command window** – Issue the command `show psets on dedicated`.

FIGURE 10-1 shows the `Psets` window for a 4-process message-passing program.

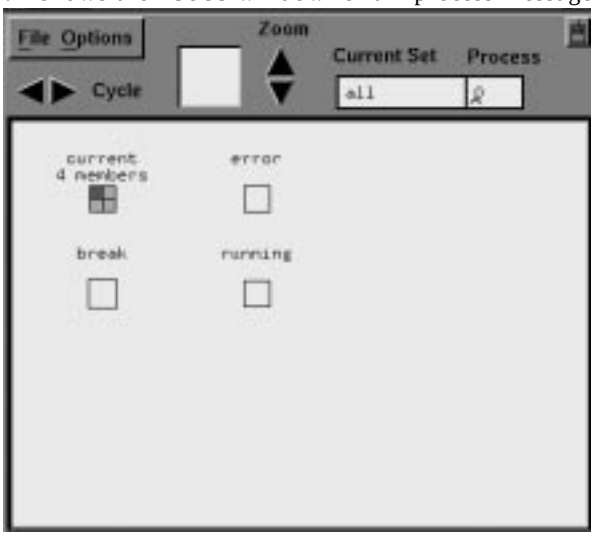

FIGURE 10-1 Psets WIndow

The various components of the window are described in detail in later sections. Here is a brief overview:

- The main area of the window shows psets and their members. Processes that are members of a set are shown as black (or colored) cells within a rectangle that represents the entire set of processes that make up the message-passing program.
- The current process (see Section 10.3.7) for each pset is shown in gray (or, on a color workstation, a darker shade of the color in the other squares). The current pset (see Section 10.3.6) is shown in the upper left corner of the window.
- You can cycle through the `cycle` pset (see Section 10.3.8) by clicking on the left and right arrows labeled Cycle at the top left of the control panel.
- If you have many psets and a large number of processes, you can use the Zoom arrows to zoom in or out on these psets. The box next to the arrows shows what part of the entire display you are seeing; you can drag the mouse through this box to pan through the display.
- You can view and change the current pset and current process via the boxes at the top right of the window
- The Options menu at the top left of the window lets you hide, display, create, and delete psets. See Sections 10.3.3 through 10.3.5.
- The File menu lets you close the `Psets` window.

10.3.2 Predefined Psets

MP Prism provides the predefined psets described below.

- **`all`** – This set contains all the processes in the program; it is the default pset.
- **`running`** – This set contains all processes that are currently executing.
- **`error`** – This set contains all processes that have encountered an error.
- **`interrupted`** – This set contains the processes that were most recently forcibly interrupted by the user. See Section 10.4.4 for a discussion of the `interrupt` command and a further explanation of this pset.
- **`break`** – This set contains the processes that are currently stopped at breakpoints.
- **`stopped`** – This set contains all processes that are currently stopped. It is the union of the sets `error`, `interrupted`, and `break`.
- **`done`** – This set contains all processes that have terminated successfully.

Except for the pset `all`, these sets are *dynamic*; that is, as a program executes, Prism automatically adjusts the contents of each set to reflect the program's current state.

In addition, there are two set names that have special meaning in MP Prism: `current` and `cycle`. They are discussed in Sections 10.3.6 and 10.3.8, respectively.

10.3.3 Defining Your Own Psets

10.3.3.1 Syntax for Defining a Pset

This section describes the syntax you can use to specify a pset. As described below, you can assign a name to a pset you specify using this syntax; this provides a useful shorthand for complicated pset specifications.

First, let's look at the syntax you use to specify a pset as an argument to a command:

To apply a command to a specific pset, use the `pset` keyword, followed by a process specification. Put this `pset` clause at the end of the command line (but before an `on` *window* clause, if any). Thus,

```
print x pset error
```

prints the values of the variable **`x`** in the predefined pset `error`. (See Section 10.7 for a discussion of printing variables in MP Prism.)

Now let's look at the ways in which you can specify your own pset as part of this `pset` clause:

Specify an individual process number. An individual process can constitute a pset. Thus,

```
print x pset 0
```

prints the value of **x** in process 0.

Specify the name of a pset. You name a pset using the `define pset` command, as described in the section "Naming Psets," below. Thus,

```
print x pset foo
```

prints **x** in the processes you have defined to be members of pset `foo`.

Specify a list of process numbers. Separate the numbers with commas. Thus,

```
print x pset 0, 4, 7
```

prints x in processes 0, 4, and 7.

Ranges and strides are allowed. Use a colon between two process numbers to indicate a range. Use a second colon to indicate the stride to be used within this range. Thus,

```
print x pset 0:10
```

prints x in processes 0 through 10. And

```
print x pset 0:10:2
```

prints **x** in processes 0, 2, 4, 6, 8, and 10.

You can combine comma-separated process numbers and range specifications. For example,

```
print x pset 0, 1, 3:5, 8
```

prints **x** in processes 0, 1, 3, 4, 5, and 8.

Specify a union, difference, or intersection of psets. To specify the union of two psets, use the symbol +, |, or ||. For example,

```
print x pset 0:2 + 8:10
```

prints **x** in processes 0, 1, 2, 8, 9, and 10.

```
print x pset foo | bar
```

prints **x** in processes that are members of either pset `foo` or pset `bar`.

Prism evaluates the pset expression from left to right. If a process returns `true` for the first part of the expression, it is not evaluated further. In the above example, if a process is a member of `foo`, its value of x is printed; Prism does not check its membership in `bar`.

Use a minus sign to specify the difference of two psets. For example,

```
print x pset stopped - foo
```

prints x in all processes that are stopped except those belonging to the pset foo.

To specify the intersection of two psets, use the &, &&, or * symbol. For example,

```
print x pset foo & bar
```

prints **x** in processes that are members of both pset foo and pset bar. If a process returns false for the first part of the expression, it is not evaluated further. In the above example, if a process is not a member of foo, Prism doesn't bother checking its membership in bar; it won't be printed in any case.

Prism must evaluate a pset expression in each process at the time the command is executed; the processes must be stopped for Prism to do this. The evaluation fails if any of the processes being evaluated are running. Using the predefined pset stopped on the left of an intersection expression is a useful way of ensuring that a command applies only to stopped processes. Thus,

```
print x pset stopped & foo
```

prints x only in the members of foo that are stopped.

Specify a condition to be met. Put braces around an expression that evaluates to true or false in each process. Processes in which the expression is true are part of the set. Thus,

```
print x pset { y > 1 }
```

prints **x** in processes where *y* is greater than 1. And

```
print x pset all - { y == 1 }
```

prints *x* in all processes except those in which *y* is equal to 1.

Membership in a set defined with this syntax can change based on the current state of your program; such a pset is referred to as *variable*. See the section "Evaluating Variable Psets," below, to learn how to update the membership of a variable pset.

For this syntax to work, the variable must be active in all processes in which the expression is evaluated. If the variable isn't active in a process, you get an error message and the command is not executed. To ensure that the command is executed, use the intrinsic isactive in the pset definition. The expression isactive**(***variable***)** returns true if *variable* is on the stack for a process or is a global. Thus, you could use this syntax to ensure that x is printed:

```
print x pset stopped && {isactive(x)}
```

10.3.3.2 Naming Psets

You can assign a name to a pset. This is convenient if you plan to use the set frequently in your Prism session.

Use the syntax described above to specify the pset. You can use any name except the names that Prism predefines; see Section 10.3.2. The name must begin with a letter; it can contain any alphanumeric character, plus the dollar sign (**$**) and underscore (_).

- **From the Psets window** – Choose `Define Set` from the `Options` menu. A dialog box is displayed that prompts for the name and definition of the pset. Click on `Create` to create the pset.
- **From the command line** – Issue the `define pset` command.

For example,

```
define pset odd 1:31:2
```

creates a pset called `odd` containing the odd-numbered processes between 1 and 31.

```
define pset xon { x .NE. 0 }
```

defines a pset consisting of those processes in which `x` is not equal to 0. Note that `x` must be active in all processes for this syntax to work. As described above, you can use the intrinsic `isactive` to ensure that `x` is active in the processes that are evaluated. For example,

```
define pset xon { isactive(x) && (x .NE. 0) }
```

Both versions create a variable pset whose contents will change based on the value of `x`. See below for more discussion of variable psets. Finally, note that all processes must be stopped for this syntax to work. To ensure that the definition applies only to stopped processes, use this syntax:

```
define pset xon stopped && { isactive(x) && (x .NE. 0) }
```

Dynamic user-defined psets are deleted when you reload a program. To get a list of these psets before reloading, issue the command `show psets`. You can then use this list to help reissue the `define pset` commands. See Section 10.3.4 for more information about `show psets`.

10.3.3.3 Evaluating Variable Psets

We have already discussed how to create variable psets—sets whose contents can change as the program executes. Prism evaluates the membership of such a set when it is defined. If no processes meet the condition (for example, because the program is not active), Prism prints appropriate error messages, but the set is defined.

Subsequently, you can re-evaluate the membership of the pset by issuing the `eval pset` command, specifying the name of the pset as its argument. For example,

```
eval pset xon
```

evaluates the membership of the pset `xon`. This causes the display for the pset to be updated in the `Psets` window.

Note that this evaluation will fail if:

- Processes are running that need to be polled in evaluating the pset; or
- The pset's definition contains a variable that is not active in any of the processes being polled

For example, if you issue this command:

```
define pset foo { x > 0 }
```

you must make sure that all processes are stopped, and `x` is active on all processes, when you issue the command

```
eval pset foo
```

To ensure that the evaluation succeeds, you would need to use the more complicated syntax described above:

```
define pset foo stopped && { isactive(x) && (x > 0) }
```

This ensures that the evaluation takes place only in processes that are stopped and in which `x` is active.

If an evaluation fails, the membership of the pset remains what it was before you issued the `eval pset` command.

You can use the `eval pset` command in event actions; see Section 10.6.1.

Note the difference between *dynamic* and *variable* psets. The membership in both can change as a program executes. Dynamic psets are predefined sets like `stopped` and `interrupted`; Prism automatically updates their membership as the program executes. Variable psets are defined by the user, and the user must explicitly update their membership by issuing the `eval pset` command.

10.3.4 Viewing the Contents of Psets

10.3.4.1 From the Psets Window

The easiest way to view the contents of psets is to use the Psets window.

By default, the window displays the current pset (which starts out being the predefined pset `all`), and the psets `break`, `running`, and `error`. When you create a new pset via the `define pset` command, that set is also displayed automatically.

The processes within a pset are numbered starting at the upper left, increasing from left to right and then jumping to the next row. You can display information about them as follows:

- Shift-click on a cell to view the Prism ID number of the process it represents.
- Shift-click elsewhere in the pset rectangle (for example, on a border) to display all the ID numbers of the processes in the pset.
- Shift-middle-click on a cell to view the process's Solaris pid and the hostname of the node on which it is running.
- Shift-middle click elsewhere in the rectangle to display the entire list of pids and hostnames for the processes in the pset.

To display a pset, choose the `Show` selection from the `Options` menu in the `Psets` window This displays a list of psets; the predefined psets are at the top, followed by any user-defined set names. Click on a set name, and that set is displayed in the window.

To hide a pset, choose the Hide selection from the Options menu. Once again, this displays the list of predefined and user-defined psets. Click on a set name to remove that set from the display.

Note that hiding a pset doesn't otherwise affect its status; it still exists and can be used in commands.

Note also that there are choices All Sets and all in the Show and Hide submenus. The All Sets choice refers to all psets; the all choice refers to the predefined pset `all`.

If you have too many psets to be shown in the display window of the Psets window, the navigator rectangle to the right of the Cycle arrows lets you pan through the psets. The white box in the rectangle shows the position of the display area relative to all the psets that are to be displayed:

You can either drag the box or click at a spot in the rectangle. The box moves to that spot, and the display window shows the psets in this area of the total display.

To display more psets at the same time, click on the Zoom up arrow to the right of the navigator rectangle. This reduces the size of the boxes representing the psets. Clicking on the Zoom down arrow increases the size of these boxes. By default, the boxes are at their highest zoom setting.

10.3.4.2 From the Command Line

Use the `show pset` command to print the contents of the pset you specify. For example, the command

show pset stopped

might produce this response:

The set contains the following processes: 0:3.

The `show pset` command is discussed further in Section 10.3.6.

The `show psets` command displays the contents and status of all psets.

```
(prism all) show psets
foo:
  definition = 0:31:2
  members = 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
  current process = 0
break:
  definition = break
  members = nil
  current process = (none)
done:
  definition = done
  members = 0:31
  current process = 0
interrupted:
  definition = interrupted
  members = nil
  current process = (none)
error:
  definition = error
  members = nil
  current process = (none)
running:
  definition = running
  members = nil
  current process = (none)
stopped:
  definition = stopped
  members = nil
  current process = (none)
current:
  definition = 6, 9, 12
  members = 6,9,12
  current process = 6
cycle:
  definition = 6, 9, 12
  members = 6,9,12
  current process = 6
all:
  definition = all
  members = 0:31
  current process = 12
```

10.3.5 Deleting Psets

You can delete named psets that you have defined. You cannot delete any predefined pset except `cycle`; see Section 10.3.8. To delete a pset:

- **From the Psets window** – Choose the Delete selection from the Options menu. This displays a list of psets that you can delete. Click on the name of the pset you want to delete. If it is currently displayed in the Psets window, it disappears.
- **From the command line** – Issue the `delete pset` command, using a pset qualifier to specify the name of a user-defined pset. For example,

  ```
  delete pset xon
  ```

deletes the pset named `xon`.

See Section 10.6.1 for a discussion of the effect of deleting a pset on events that have been defined to affect the members of that set.

10.3.6 Current Pset

The command syntax described in Section 10.3.3 lets you apply a command to a specific pset. If you don't use this syntax, the command is applied to the *current* pset; `current` is a predefined pset name in Prism. In addition, many graphical actions in MP Prism apply only to the members of the current set.

When a program is first loaded, the current pset is the default pset, `all`.

You can change the current pset via the `Psets` window or from the command line.

- **From the Psets window** – There are several ways of changing the current pset via the `Psets` window:
  - If the set is displayed in the `Psets` window, simply double-click anywhere in its display (for example, on its name, or in the box beneath its name).
  - Choose the Set Pset selection from the Options menu. This displays a list of psets. Click on the name of the set you want to be current.
  - Edit the name of the pset in the box below Current `Set` at the top right of the Psets window, then press Return.

  When you change the current set, the new name appears in the Current Set box in the Psets window, and the current set shown at the top left of the psets area changes to reflect the contents of the new set.
- **From the command line** – Issue the `pset` command. For example,

  ```
  pset foo
  ```

  changes the current pset to `foo`.

You can also use the `pset` command with the pset-specification syntax described in Section 10.3.3. For example,

```
pset 0:15:3
```

You cannot change the current pset to one that has no members. If you try to do so, nothing happens in the Psets window, and you get a message like this one in the history region of the command window:

```
Cannot set current pset to running -- it is empty.
```

10.3.6.1 Finding Out the Current Pset

MP Prism provides many ways of finding out the current pset:

- As described in the previous section, the name of the current pset appears in the Current Set box at the top right of the Psets window.
- The name of the current pset appears in the status region in MP Prism's main window.
- Issuing the `pset` command without arguments displays the current set.
- The `(prism)` prompt on the command line and in commands-only MP Prism identifies the current pset. For example, Prism's response to the `pset` command in the previous section would look like this:

```
(prism all) pset foo
(prism foo)
```

Note – In giving examples of MP Prism commands, the (`prism`) prompt is used only when necessary to show the effect of a command.

To list the processes in the current pset, issue the `show pset` command without arguments:

```
(prism foo) show pset
pset 'current' is defined as 'foo'.
The set contains the following processes: 1,2.
```

The Psets window also displays the processes in the current pset.

10.3.6.2 Current Pset and Dynamic Psets

Section 10.3.2 described dynamic psets—predefined sets like `running`, `stopped`, and `interrupted`, whose contents Prism automatically updates during the execution of the program.

If you choose a dynamic pset to be the current pset, you create a *static* pset that consists of the processes that are members of the dynamic set at the time you issue the `pset` command (or otherwise choose it to be the current set). To make this clear, the `(prism)` prompt changes to list the processes that are members of this static set. For example, if processes 0, 1, and 13 are the only processes that are stopped, the `pset` command has this effect:

```
(prism all) pset stopped
(prism 0:1, 13)
```

Output of the `show pset` command is explicit under these circumstances:

```
(prism all) pset stopped
(prism 0:1. 13) show pset
The current set was created by evaluating the pset
'stopped' once at the time when it became the current set.
The set contains the following processes: 0:1, 13.
```

Issuing the `pset` command with no arguments displays the same information.

Note that the `(prism)` prompt can become quite long if there are many processes in a current pset derived from a dynamic pset. By default, the prompt length is limited to 25 characters. You can change this default by issuing the `set` command with the `$prompt_length` variable, specifying the maximum number of characters to appear in the pset part of the prompt. For example, this command shortens the prompt `long_pset_name` to `long_pset`:

```
(prism long_pset_name) set $prompt_length=9
(prism long_pset)
```

10.3.6.3 Current Pset and Variable Psets

Section 10.3.3 describes how to create variable psets—user-defined psets whose membership can change in the course of program execution. You use the `eval pset` command to update the membership of a variable pset. If you make a variable pset your current set, its membership is determined by the most recent `eval pset` command you have executed for the set. If you have not executed an `eval pset` command to update the set's membership, the membership continues to be what it was when you created the set.

10.3.7 The Current Process

Each pset has a current process. The current process has a variety of uses in MP Prism:

- The source window displays the source code executing in the current process of the current pset.

- The `Where` graph centers around the call stack of the current pset's current process; see Section 10.6.2.
- The current process determines the scope used in interpreting the names of variables; see Section 10.6.3.

By default, the current process is the lowest-numbered process in the set. You can change this as described below.

- **From the Psets window** – Use one of these methods to change the current process via the `Psets` window:
  - Click on the cell representing the process in the displayed pset. The cell turns gray (or, on color workstations, a darker shade of the color for the other processes).
  - To change the current process in the current pset, you can also edit the number in the box under Process at the top right of the window, then press Return.
- **From the command line** – Issue the `process` command to specify another current process for the current pset. For example,

```
(prism all) process 2
The current process is now 2.
```

Issue the `process` command without any arguments to print the current process of the current pset.

When you change a current process, by any of the methods described above, the pset keeps this new current process until you explicitly change it. That is, if you switch to a different current set, then switch back to the original set, the original set will still have the same current process.

10.3.8 The Cycle Pset

In debugging a message-passing program, you may often want to look in turn at each individual process within a pset—for example, to see what the problem is for each process in the `error` pset. The `cycle` pset provides you with a convenient way of doing this.

You create a `cycle` pset out of an existing pset. If the existing set is dynamic, the `cycle` set is statically fixed when you create it. You can then cycle through each process in this set to examine it in turn.

By default, the `cycle` set is equivalent to the current set. You can set it to some other set via the `define pset` command, as described in Section 10.3.3. For example,

```
(prism all) define pset cycle foo
```

copies `foo` into the `cycle` set.

You can cycle through the processes in the `cycle` set as follows:

- **From the Psets window** – Use the Cycle arrows at the top left of the window to cycle through the members of the `cycle` set. Click on the right arrow to cycle up through the members of the set; click on the left arrow to cycle down through the members.

  Clicking on a Cycle arrow does this:

  - It advances the current process in the `cycle` pset to be the next member in the set.
  - It makes the current pset consist of only this process.
- **From the command line** – Use the `cycle` command. This has the same effect as clicking on the right cycle arrow in the `Psets` window. For example, this Prism session defines a pset, makes it the current set, and then cycles through its members:

```
(prism all) define pset foo 0:3
(prism all) pset foo
(prism foo) cycle
(prism 1) cycle
(prism 2) cycle
(prism 3) cycle
(prism 0)
```

Note that changing the `cycle` pset erases any previous cycling information. For example, if you do the following:

1. **Make `foo` the current set and cycle partway through it.**
2. **Make `bar` the current set.**
3. **Once again make `foo` the current set.**

then you start at the beginning again when you cycle through the members of `foo`.

From the source-window popup menu – Choose `Cycle` from this menu to advance to the next member of the `cycle` pset.

10.3.8.1 Cycle Visualizer Window

MP Prism includes a Cycle window type for visualizing data. When you print a variable's value to the Cycle window, the value changes to that of the variable in the new process whenever you cycle through the members of the `cycle` pset. For more information, see Section 10.7.

10.3.9 Using Psets in Commands

As mentioned at the beginning of Section 10.3, some commands can take a pset as a qualifier; they are listed at the end of this section. Put this qualifier after any arguments to the command, but before the optional `on` *window* syntax that specifies the window in which output is to be displayed. A command with a pset qualifier applies only to the processes in the set. If you omit the qualifier, the command applies to the processes in the current set.

Thus,

```
stop at 12 pset error
```

sets a breakpoint at line 12 for the processes in pset **`error`**.

```
where pset 0:10 on dedicated
```

displays the `Where` graph for processes 0 through 10. See Section 10.6.2 for a description of the `Where` graph.

```
trace at 12 if x > 10
```

creates a trace event for the members of the current pset.

Note that this last command applies only to the members of the current pset. To apply it to all processes, use the syntax

```
trace at 12 if x > 10 pset all
```

Many commands, of course, cannot logically take a pset qualifier. You get an error message if you try to issue one of these commands with a pset qualifier.

Here are the Prism commands that *can* take a pset qualifier:

```
address/
assign
call
catch
cont, contw
display
ignore
interrupt
next, nexti,
print
pstatus
return
step, stepi
stop, stopi
trace, tracei
wait
whatis
where
```

10.4 Executing a Program in MP Prism

You start execution of a program in MP Prism just as you do in DP Prism—by issuing the `run` command or choosing the Run or Run (args) selection from the Execute menu. See Section 3.4. You can also attach to an already-running program using the `attach` command, as described below.

10.4.1 Attaching and Detaching

You can use the `attach` command in MP Prism to attach to a running task. Specify the task's task ID on the `attach` command line. Note that if you use a process ID as an argument to `attach`, you will be attached to that individual process, not to the entire task.

You can use the `detach` command to set a debugged task free. You can detach from your task if you started it up with `run` or `tmrun`, and, of course, you can also detach if you previously attached.

MP Prism only lets you detach when all the processes in the task are stopped (to make sure we remove all breakpoints before detaching). The detach operation itself sets them all running again, outside control of the debugger.

10.4.2 Quitting

Issuing the `quit` command terminates the debugging session. As mentioned above, before quitting, MP Prism will kill your debugged process if it was started with `run`, or it will detach from it if you previously attached.

10.4.3 Stepping and Continuing Through a Program

In MP Prism, menu actions such as Step and Next apply to the processes of the current pset.

DP Prism (like other debuggers) waits for a `step`, `next`, or `cont` command to finish executing before letting you issue most other commands. This behavior is unnecessarily restrictive in MP Prism, however; therefore, if one process or set of processes is executing code, you can still issue commands that affect other processes. For example, you can switch to a different pset and start or stop execution of its processes.

10.4.4 Interrupting and Waiting for Processes

It is useful in debugging message-passing programs to wait for a specific process or processes to stop executing, or to be able to interrupt execution of individual processes. MP Prism therefore provides the commands `interrupt` and `wait`.

Use the `interrupt` command to forcibly interrupt execution of a specified process or processes. For example,

```
interrupt pset 0
```

interrupts execution of process 0.

```
interrupt pset running
```

interrupts all running processes.

Using the `interrupt` command resets the predefined pset `interrupted` so that it includes the newly interrupted processes. Processes leave this pset when they continue execution.

In MP Prism, the Interrupt selection from the Execute menu interrupts processes in the current pset that are running.

Use the `wait` command to wait for a specified process or processes to stop execution. A process is considered to have stopped if it has entered the `done`, `break`, `interrupted`, or `error` state.

There are two versions of the `wait` command:

- Use the syntax `wait` or `wait every` to wait for every member of the specified pset to stop. If no pset is specified, the command applies to the current pset. Thus,

  ```
  (prism notx) wait every
  ```

  waits for every process in the pset `notx` to stop. The current process will be whatever it would normally be; see Section 10.3.6. This is the default behavior of the `wait` command.
- Use the syntax `wait any` to wait for any member of the specified pset to stop. If no pset is specified, the command applies to the current pset. When the first process stops, it becomes the current process of this pset. Thus,

  ```
  wait any pset foo
  ```

  waits for the first process in pset `foo` to stop.

There are corresponding Wait Any and Wait Every selections in MP Prism's Execute menu. They apply to the processes of the current set.

You can end the wait by

- typing Ctrl-c; this does not affect processes that are running

- choosing the Interrupt selection from the Execute menu; this stops processes that are running, as well as ending the wait

Note that, if you prefer that `step` and `next` commands wait for processes to finish executing before letting you issue other commands, you can issue them along with the `wait` command. For example,

```
step; wait
```

This says: Execute the next line, then wait for all processes in the current pset to finish execution.

If you use this command sequence frequently, you can provide an alias for it via the `alias` command. Prism provides the default alias `contw` for these commands:

```
cont; wait
```

10.4.5 Execution Pointer

In DP Prism, the **>** symbol in the line-number region points to the next line to be executed; see Section 2.6. In a message-passing program, there can be multiple execution points within the program. MP Prism marks all the execution points for the processes in the current set by a **>** in the line-number region (or a ***** if the current source position is the same as the current execution point). Shift-click on this symbol to display a pop-up window that shows the process(es) for which the symbol is the execution pointer.

10.4.6 Finding Out Execution Status

Issue the `pstatus` command to find out the execution status of processes. Without a pset qualifier, it displays the execution status of the members of the current set. For example,

```
(prism all) pstatus
process 0: running
process 1: stopped in procedure ″pawn_moves″ at ″chess.c″:49
process 2: interrupted in procedure ″construct_move″ at
″chess.c″:1187
process 3: interrupted in procedure ″rook_check″ at ″chess.c″:746
```

Use a pset qualifier to find out the execution status of the members of the specified pset.

10.4.7 Executing a Program in Commands-Only MP Prism

When you issue the `run` command to execute a program in the commands-only version of MP Prism, the program starts up in the background. If the program needs to read terminal input, you must then issue the `fg` command at the `(prism)` prompt to run the program in the foreground. You cannot execute Prism commands while the program is executing in the foreground. To have the program run in the background and regain the `(prism)` prompt, type Ctrl-z.

10.5 Combining DP and MP Prism

If you have an executable that combines data parallel and message-parallel modules, such as an HPF program calling an EXTRINSIC (F77_LOCAL) subroutine, Prism switches as needed between DP Prism and MP Prism when you step through your code, set breakpoints, etc. For example, if you step from a data parallel routine into a local message-passing routine, the user interface will change to that of MP Prism, and you can display and manipulate psets as described in this chapter.

10.6 Debugging in MP Prism

Debugging a message-passing program can be considerably more complex than debugging a serial or data parallel program, since you are in effect debugging multiple individual programs concurrently. MP Prism's concept of psets lets you focus your debugging efforts on the processes that are of particular interest. This section describes the following areas in which debugging in MP Prism, with its psets, is different from debugging in DP Prism (described in Chapter 4):

- Events — Section 10.6.1
- The `Where` graph — Section 10.6.2
- Scope — Section 10.6.3
- Examining local process core files —Section 10.6.4

10.6.1 Events in MP Prism

Events in MP Prism can take a pset qualifier. You can specify this in an event field in MP Prism's event table, as shown in FIGURE 10-2.

FIGURE 10-2 Pset Field in MP Prism's Event Table

If you don't supply a pset qualifier, the event applies to the current pset. If you create the event before changing the current set, the event applies to the default set, which is `all`.

Thus,

```
stop in receive pset notx
```

sets a breakpoint in the `receive` routine for the processes in the set `notx`. Each process in the set stops when it reaches this routine. It is possible, of course, that some processes may never reach this routine. This becomes an issue when you include actions in an event; see below.

If all the processes in the pset have stopped, you can continue them by issuing a command like

```
cont pset notx
```

Here is another example:

```
stop if x > 10
```

This command stops execution for any process in the current pset if the process's value for the variable `x` is greater than 10.

Prism evaluates the expression in the condition locally—that is, separately for each process. Similarly, if `a` and `b` are arrays,

```
stop if sum(a) > sum(b)
```

stops execution for a process in the current set if the sum of the values of **`a`** in that process is greater than the sum of the values of `b`.

All processes that are stopped at breakpoints are members of the predefined pset `break`.

10.6.1.1 Events and Dynamic Psets

If you use a dynamic pset as a qualifier for an event, its membership is evaluated when you issue the command defining the event. Thus, the command

```
stop at 10 pset interrupted
```

creates a breakpoint only in the processes that are interrupted at the time the command is issued. If no processes are currently interrupted, you receive an error message.

One result of this is that you cannot define events that involve dynamic psets before the program starts execution.

10.6.1.2 Events and Variable Psets

If you use a user-defined variable pset as a qualifier, its membership is determined by the most recent `eval pset` command you issued for that pset.

As is the case with dynamic psets, you cannot define events that involve variable psets before the program starts execution.

10.6.1.3 Actions in Events

Events in MP Prism can take action clauses, just as they can in DP Prism. For example,

```
stop at 10 {print x} pset foo
```

prints `x` for the pset `foo` when the members of `foo` are stopped at line 10.

Important – Associating an action with an event forces a global synchronization at the breakpoint or tracepoint. In the example above, every process in pset `foo` must stop at line 10 before `x` can be printed. If a member does not stop at line 10, the action never takes place. In a trace event, all processes in the pset must stop at the specified place and synchronize; the action then takes place, and the processes automatically continue execution.

You can include an `eval pset` command as an event action. For example,

```
stop in send {eval pset sending}
```

evaluates the pset `sending` when all the members of the current pset are stopped in `send`. You receive error messages if it is impossible to evaluate membership in a pset (for example, because a variable in the set definition is not active).

Note these limitations in using event actions in MP Prism:

- You cannot include the following commands that manipulate psets:

  - `define pset`
  - `delete pset`
  - `process`
  - `pset`
- You cannot include a pset qualifier in the action. The command in the action clause takes its pset from the pset of the event.
- As in DP Prism, you cannot include commands that affect program execution, specifically
  - `cont` and `contw`
  - `run`
  - `step` and `stepi`
  - `next` and `nexti`
  - `wait`
- As in DP Prism, you cannot include the `load`, `reload`, `return`, and `core` commands.

10.6.1.4 Displaying Events by Process

Issue the `show events` command with a process number as an argument to display all events associated with that process. For example,

```
(prism all) show events 0
(1) stop at 10 pset 0
(3) stop at 575 {print x] pset all
(7) trace y pset bar
```

Issuing `show events` with no arguments has its standard behavior; that is, it prints out all events.

10.6.1.5 Events and Deleted Psets

If you create an event that applies to a particular pset, and subsequently delete the pset, the event continues to exist. Its printed representation, however, is changed so that it shows the processes that were members of the pset at the time you deleted the set.

10.6.1.6 Using the Line-Number Region

Section 2.6 describes how the line-number region displays breakpoints and tracepoints in DP Prism. MP Prism provides a variation of this feature:

- It displays a `B` next to a line number if all processes in the current pset have a breakpoint set at that line.

- It displays a `b` if some but not all of the processes in the current pset have a breakpoint set at that line.
- It displays a `T` if all processes in the current pset have a tracepoint set at that line.
- It displays a `t` if some but not all of the processes in the current pset have a tracepoint set at that line.

If there is a mixture of breakpoints and tracepoints set on the line, Prism uses the `B-b-T-t` sequence to determine what letter to display. For example, if a line has a breakpoint set in one process and a tracepoint set in all processes, Prism displays a `b`.

As in DP Prism, you can shift-click on the letter in the line-number region to display the complete event (or events) associated with it.

10.6.2 Where Graph

In DP Prism, choosing Where from the Debug menu displays the call stack for the program; see Section 4.5. A message-passing program, however, can have multiple call stacks, one for each process. To show the relationships among these call stacks, MP Prism provides a *Where graph*; this window displays a snapshot of the dynamic call graph of the program. Information is displayed for all processes that are not running.

To display the Where graph:

- **From the menu bar** – Choose `Where` from the `Debug` menu.
- **From the command line** – Issue the command

```
where on dedicated
```

A window like the one shown in FIGURE 10-3 is displayed.

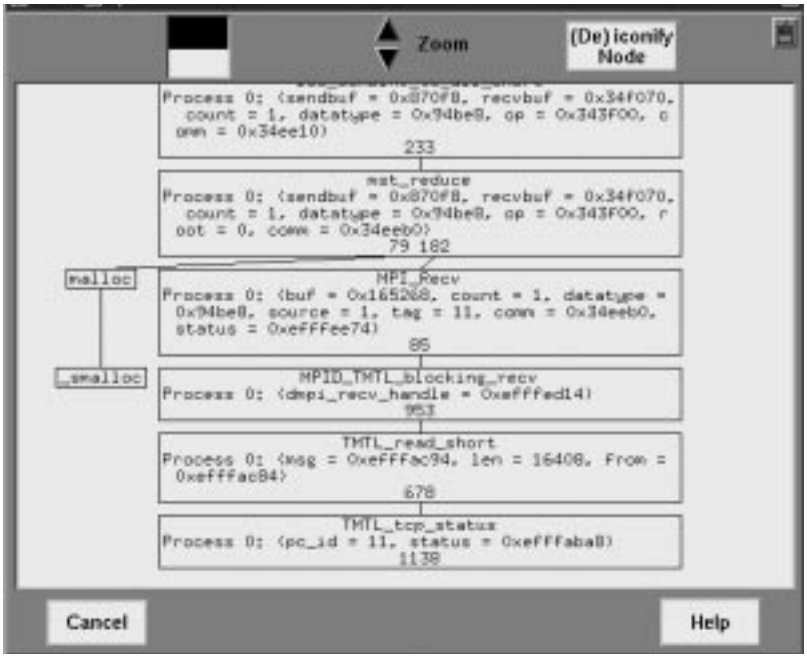

FIGURE 10-3 Where Graph

The Where graph centers on the current process of the current pset—that is, the processes related to it are lined up in a single column. In FIGURE 10-3, process 0 is the current process. If you change the current process, the Where graph rearranges itself. The default zoom level of the Where graph shows the arguments for the current process.

At the bottom of each box are line numbers indicating where processes branch. Thus, in FIGURE 10-3, one or more processes call `malloc` at line 79 of `mst_reduce`.

Shift-click in each function's box to display a popup window showing the numbers of the processes with this function in their call stack, along with their arguments.

10.6.2.1 Panning and Zooming in the Where Graph

As FIGURE 10-4 shows, the Where graph can get quite large, so MP Prism provides methods for panning through it and zooming in and out.

The white box in the navigator rectangle at the top of the window shows the position of the display area relative to the entire Where graph. You can either drag the box or click at a spot in the navigator. The box moves to that spot, and the window shows the Where graph in this area of the total display.

To display more of the `Where` graph at the same time, click on the Zoom down arrow to the right of the navigator. This reduces the size of the boxes representing the functions and removes information. FIGURE 10-4 shows the `Where` graph of FIGURE 10-3, zoomed out one level. Note that the information about the current process's arguments is gone.

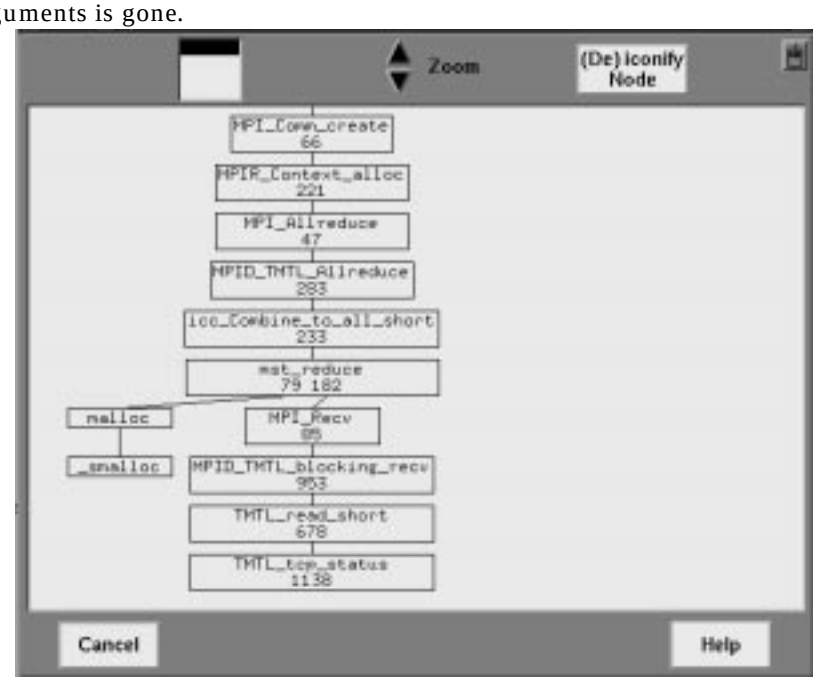

FIGURE 10-4 Where Graph, Zoomed Out One Level

Zooming out one more level removes the line numbers, and one more level after that removes the function names, leaving only boxes connected by lines. You can still shift-click on a box to display information about it.

Clicking on the Zoom up arrow increases the size of the function boxes and includes more information in them. FIGURE 10-5 shows the Where graph of FIGURE 10-3, zoomed in. In this case, the Where graph shows, for each function, the processes that

have that function in their call stack. As in the `Psets` window, the processes are represented as bitmaps of cells, numbered starting at the upper left, increasing from left to right and then jumping to the next row.

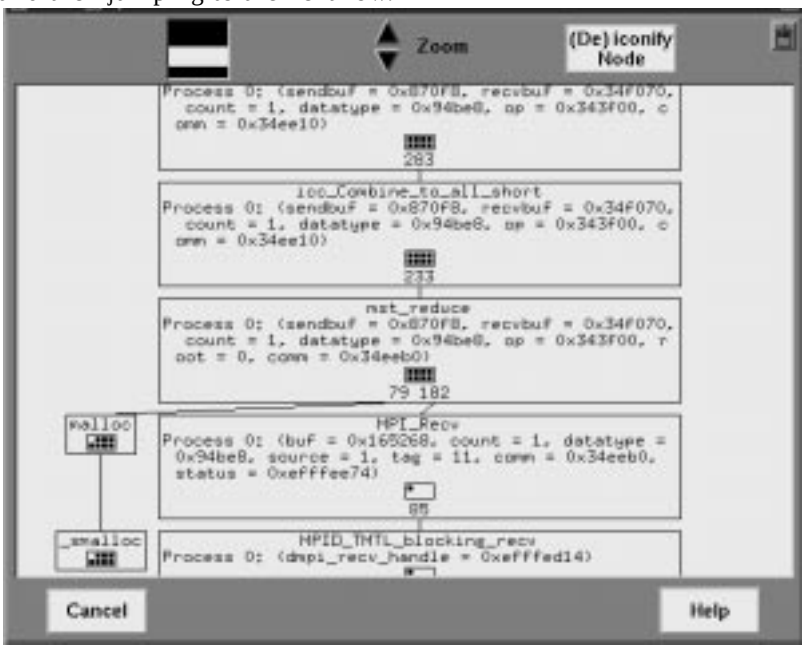

FIGURE 10-5 Where Graph, Zoomed In

Zooming in another level shows all arguments for all processes.

10.6.2.2 Shrinking Selected Portions of the Where Graph

You can shrink selected portions of the Where graph. This is useful if you want to see the overall structure of the graph, but in addition want to focus on certain functions.

Middle-click on a function to iconify it and all of its children. Middle-click on an iconified function to re-expand it and its children to the current zoom level.

Alternatively, you can click on the (De)iconify `Node` button next to the Zoom arrows at the top of the Where graph. This changes the mouse pointer to a target. You can then left-click on a function to iconify it and its children. If it is already iconified, left-clicking on it will re-expand it and its children. To cancel the operation, left-click anywhere outside of the boxes surrounding the functions.

10.6.2.3 Moving Through the Where Graph

When you first display the `Where` graph, the `main` function is highlighted. You can left-click on a function to highlight it. Or, you can move through the Where graph via the keyboard:

- Use the up arrow key to move to the parent of the highlighted function.
- If line numbers are visible in the highlighted function, by default the leftmost number is selected by having a box drawn around it. Use the left and right arrows to select other line numbers in the function. You can then use the down arrow key to highlight the function called at the selected line.

10.6.2.4 Making a Function the Current Pset

Pressing the spacebar while in the Where graph does the following:

- It changes the current function to be the function that is highlighted in the `Where` graph.
- This function is displayed in the source window.
- It creates a new current pset, with the same name as the function, and containing the processes with this function in their call stack. The current process of this current set is the lowest-numbered process in the set.

10.6.2.5 Issuing the `where` Command in MP Prism

Issuing the `where` command by default displays (in the history region) the call stack consecutively for each process in the current set (or in the pset you specify via the `pset` qualifier).

Issuing the command

```
where on dedicated
```

displays the `Where` graph, as described above.

Issuing the command

```
where on snapshot
```

puts the history-region output into a window; it does not create a Where graph.

10.6.3 Scope in MP Prism

See Section 4.5.2 for a discussion of scope in DP Prism.

In MP Prism, the scope of the current process determines the scope for resolving the names of variables. See Section 10.3.7 for a discussion of the current process.

If a command applies to a pset other than the current set, Prism uses the scope of that set's current process.

It is possible that other members of the pset will have different scopes from that of the current process, or that its scope level will not even exist in these processes. In these cases, you receive an error message when you try to issue a command (for example, `print` or `display`) that requires a consistent scope. To solve the problem, you can do one of the following:

- Restrict your pset so that it contains only members with the same scope.
- If the current process's scope level does not exist in other processes in the set, you can use the `up` command to move up its call stack to a point where it has a scope level that does exist in the other processes.
- If different processes in the set have different scopes, you can issue the `up` and `down` commands as needed to ensure that they all have the same scope.

Commands such as `pset` and `process` that affect scope print the current function when you issue them.

10.6.4 Examining Process Core Files

You can use Prism to examine a core file created for a message-passing program. To do this, specify the core filename on the command line, after the name of the executable program. For example,

```
% prism a.out core
```

When Prism comes up, you can issue commands like `where` and `print` to inspect the state of your process at the time the core dump was taken. But note these restrictions:

- You actually start DP Prism rather than MP Prism, since there is only one core file. Thus, you cannot use psets or other features of MP Prism.
- You cannot issue any execution commands (for example, `run`, `cont`, or `step`).
- You cannot change the values of variables via the `assign` command.

Also, note that you cannot use the `core` command to examine a core file once you have started MP Prism.

Finally, note that if multiple processes dumped core, the resulting core file may be overwritten, and therefore invalid.

10.7 Visualizing Data in MP Prism

See Chapter 5 for general information on visualizing data in Prism.

When you print or display an object in MP Prism, the data is shown for all processes in the pset you specify (in the current pset, if you do not include a pset qualifier). Choosing the Print or Display selection from the Debug menu prints or displays data for processes in the current pset.

If there is only one process in the pset, the visualizer that is displayed is no different from the visualizer you would see in DP Prism.

If there is more than one process in the pset, Prism adds a dimension to the visualizer. The extra dimension represents the processes in the set. For example, if the variable is scalar, Prism displays a 1-dimensional array that represents the value of the variable in each process. If you are printing a 1-dimensional array, Prism uses a 2-dimensional visualizer.

For C programs, axis 0 represents the processes. For Fortran 77 programs, the highest-numbered axis represents the processes.

Prism can aggregate data from multiple processes only if the expression has the same size and number of dimensions in each process; if it doesn't, Prism prints an error message.

In the example shown in FIGURE 10-6, the variable `board` is an 8x8 array (representing a chess board); the current pset contains four processes. Therefore, MP Prism displays a 3-dimensional visualizer. Axis 0 represents the processes. The figure shows the values of `board` in the first process in the set. You would drag the white bar in the slider portion of the data navigator to display the values in the other processes in the set. (Note that, for a 2-dimensional Fortran array, where axis 3 would represent the processes, you might want to rearrange the display axes so that axis 3 is on the slider. You can do this by clicking in the box to the left of the slider and changing the number to a 3.)

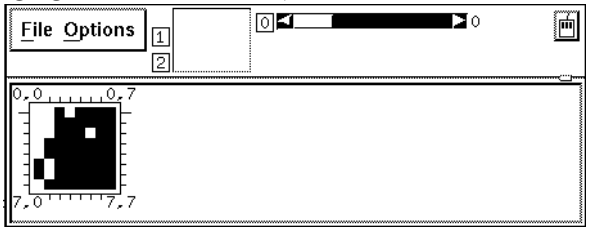

FIGURE 10-6 Visualizer in MP Prism (Threshold Representation)

To find out the value and process number for an element, shift-click on the element.

Printing to the history region, or in commands-only Prism, works the same way. Axis 0 represents the processes. Here is some of the history-region output for the data shown in FIGURE 10-6:

```
(prism all) print board
board =
process 0
(0,0,0:4) 4   1    0    3    0 (0,0,5:7)  -1          -4
(0,1,0:4) 2   1    0    0    0 (0,1,5:7)   0    -1      0
(0,2,0:4) 3   1    0    0    0 (0,2,5:7)   2    -1     -3
(0,3,0:4) 5   0    0    0   -1 (0,3,5:7)   0     0      -5
(0,4,0:4) 4   0    0   -2    0 (0,4,5:7)   0     0      -6
(0,5,0:4) 0   1    0    0    0 (0,5,5:7)   0    -1       0
(0,6,0:4) 0   1    0    0    0 (0,6,5:7)   0    -1       0
(0,7,0:4) 6  -1    0    0    0 (0,7,5:7)   0    -1      -4
process 1
(1,0,0:4) 4   1    0   3     0 (1,0,5:7)  -1    0      -4
(1,1,0:4) 2   1    0    1    0    ...
```

Note that the elements of axis 0 do not necessarily correspond to the numbers of the processes they represent. For example, if you were visualizing a variable in pset (1, 3, 5, 7), element 0 of axis 0 would represent process 1, element 1 would represent process 3, etc.

MP Prism provides a `Cycle` visualizer window you can use to display the values of a variable in the `cycle` pset; see Section 10.3.8. If you issue the command

```
print x on cycle
```

Prism displays a window containing the value of `x` in the current process of the current pset. If you then issue the `cycle` command or otherwise cycle through the members of the `cycle` pset, this window automatically updates to display the value of `x` in the next member of the set. This provides a convenient way of examining a variable in a series of processes.

10.8 Customizing MP Prism

You can customize MP Prism just as you customize DP Prism; see Chapter 9. Changes you make in one apply to the other. Both MP Prism and DP Prism use the same `.prisminit` file. This could lead to error messages if you bring up DP Prism and the file contained MP Prism-specific commands. Therefore, Prism lets you specify that commands in your `.prisminit` file are for MP Prism only by bracketing them with `#ifdef MP` and `#endif`. For example,

```
alias c cont
#ifdef MP
pset 0
alias c ”cont; wait every”
#endif
```

These commands define `c` to aliases differently in DP Prism and MP Prism, and set the initial pset to 0 in MP Prism.

To provide this feature, Prism must preprocess the `.prisminit` file; by default it does not do this. To tell Prism to preprocess the file, use the Prism resource `Prism.cppPath`, specifying the path to your C preprocessor as its setting; typically, this is `/lib`. Thus, you would set the resource as follows:

```
Prism.cppPath:/lib
```

See Section 9.4 for information on setting Prism resources. Note, however, that commands-only Prism is not aware of the settings of Prism resources such as `Prism.cppPath`, unless the settings are contained in the system-wide Prism `app-defaults` file.

10.9 Using MP Prism With PVM Programs

You can use Prism with PVM message-passing programs, with the following limitations:

- All instances of the PVM program must have the same executable.
- You must attach to a running PVM program from the Prism command line, as described below.
- Prism must be running on a node on which a PVM daemon is running.
- Prism does not work with dynamic PVM programs. Prism only sees the processes that exist when it attaches to the PVM program. You can subsequently add other processes, but information about them will not be available in Prism.

Use the `-pvm` option to the `prism` command to specify that you are going to be working on a PVM program.

As mentioned above, Prism support for PVM programs is limited to attaching to a running PVM program. To attach to a PVM program, you specify the process ID (pid) of any process in the PVM program; Prism obtains the other pids from the PVM daemon. You can obtain the pid by issuing the `ps` command.

Here is an example of attaching to a PVM program:

```
% prism -pvm pvm_exec 652
```

Host Prism starts up on your login node. MP Prism attaches to the running processes of the PVM program `pvm_exec`. Note that you shouldn't typically run Prism via the `tmrun` command in this situation. Prism must run on a node that has a PVM daemon also running on it, and you cannot guarantee that this will be the case if you execute Prism via `tmrun`.

Once you have attached to the PVM program, you can do anything you normally do in Prism when working on a message-passing program.

10.10 Using MP Prism With Sun MPI Programs

You can use MP Prism with Sun MPI programs, or other C or Fortran programs that use a library of MPI routines.

See Section 10.2.2 for information about entering MP Prism.

You can use all features of MP Prism as described in this chapter to work on your Sun MPI program.

Note the key advantage of using Prism with a Sun MPI program: The Sun MPI program is viewed as a single parallel program; all processes of the parallel program are visible from within a single Prism session. You do not have to attach a separate debugger to each Sun MPI process.

10.10.1 Setting MPI_INIT_TIMEOUT

Sun MPI has timeouts built into the software to help detect when there are problems starting an MPI task. However, these timeouts can be triggered erroneously when you are debugging programs, such as when using Prism, and should therefore be disabled prior to using a debugger on a Sun MPI program. The environment variable `MPI_INIT_TIMEOUT` can be used to lengthen or disable the timeout time. When `MPI_INIT_TIMEOUT` is set to a positive integer, the timeout value is set to that time in seconds. When it is set to 0 or a negative integer, the timeout is disabled. The default value is 600 seconds (10 minutes).

For example, to disable timeouts (in a C shell):

```
% setenv MPI_INIT_TIMEOUT -1
```

Again in a C shell, to set timeouts to 5 minutes:

```
% setenv MPI_INIT_TIMEOUT 300
```

APPENDIX **A**

Commands-Only Prism

You can run Prism in a commands-only mode, without the graphical interface. This is useful if you don't have access to a terminal or workstation running X. All Prism functionality is available in commands-only mode except features that require graphics (for example, visualizers). See Section A.1.

If you are using an Xterm, you can also run a commands-only version of Prism that lets you redirect the output of certain commands to X windows. This may be preferable to users who are used to a command-line interface for debugging, but want to take advantage of some of Prism's graphical features. See Section A.5.

For further information on individual commands, read the sections of the main body of this guide dealing with the commands, and read the reference descriptions in the *Prism Reference Manual*.

A.1 Specifying the Commands-Only Option

To enter commands-only mode, specify the `-C` option on the `prism` command line. You can also include other arguments on the command line; for example, you can specify the name of a program, so that Prism comes up with that program loaded. X toolkit options are, of course, meaningless. See Section 2.2.2, "Command-Line Options" for more information on command-line options.

When you have issued the command

```
% prism –C -np 4 a.out
```

you receive this prompt:

```
(prism all)
```

You can issue most Prism commands at this prompt, except for commands that apply specifically to the graphical interface; these include `pushbutton`, `tearoff`, and `untearoff`.

A.2 Issuing Commands

You operate in commands-only Prism just as you do when issuing commands on the command line in graphical Prism; output appears below the command you type, instead of in the history region above the command line. You cannot redirect output using the `on window` syntax. You can, however, redirect output to a file using the `@ filename` syntax.

Commands-only Prism supports the editing key combinations supported by graphical Prism, plus some additional combinations. Here is the entire list:

- **Ctrl-a** – Moves to the beginning of the line.
- **Ctrl-b** (or **Ctrl-h**) – Moves back one character.
- **Ctrl-c** – Interrupts execution.
- **Ctrl-d** – Deletes the character under the cursor.
- **Ctrl-e** – Moves to the end of the line.
- **Ctrl-f** – Moves forward one character.
- **Ctrl-j** – (or **Ctrl-m**)
  Done with input (equivalent to pressing the Return key).
- **Ctrl-k** – Deletes to the end of the line.
- **Ctrl-l** – Refreshes the screen.
- **Ctrl-n** – Displays the next command in the commands buffer.
- **Ctrl-p** – Displays the previous command in the commands buffer.
- **Ctrl-u** – Deletes to the beginning of the line.

When printing large amounts of output, commands-only Prism displays a `more?` prompt after every screenful of text. Answer `y` or simply press the `Return` key to display another screenful; answer `n` or `q`, followed by a carriage return, to stop the display and return to the `(prism)` prompt.

You can adjust the number of lines Prism displays before issuing the `more?` prompt by issuing the `set` command with the `$page_size` variable, specifying the number of lines you want displayed. For example, issue this command to display 10 lines at a time:

```
(prism) set $page_size = 10
```

Set the `$page_size` to 0 to turn the feature off; Prism will not display a `more?` prompt.

A.3 Useful Commands

This section describes some commands that are especially useful in commands-only Prism.

Use the `list` command to list source lines from the current file. For example,

```
(prism) list 10, 20
```

prints lines 10 through 20 of the current file.

Use the `show events` command to print the events list. Use the `delete` command to delete events from this list.

Use the `set` command with the `$print_width` variable to specify the number of items to be printed on a line. The default is 1.

A.4 Leaving Commands-Only Prism

Issue the `quit` command to leave commands-only Prism and return to your Solaris prompt.

A.5 Running Commands-Only Prism From an Xterm: The -CX Option

Issue the `prism` command with the `-CX` option from an Xterm to start up a commands-only Prism that lets you redirect the output of certain commands to X windows. The information presented earlier in this chapter about commands-only Prism also applies to this version, except that this version lets you redirect output using the `on window` syntax.

You can redirect the following output to X windows:

- **visualizers** (including structure visualizers) – `print` or `display` command
- **Where** graph (MP Prism only) – `where` command
- **Psets** window (MP Prism only) – `show psets` command

To redirect the output, issue the appropriate command with the `on dedicated` or `on snapshot` syntax, just as you would in graphical Prism. For example, this command displays a visualizer for x in a dedicated window:

```
(prism) print x on dedicated
```

In addition, you can display help windows from within windows that you pop up in this way.

Index

D

E

F

G

H

I

K

L

M

N

O

P

T

U

V

W

X