The
Connection Machine
System

Prism User's Guide

Version 1.2
March 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, March 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

Contents

About This Manual

Objectives of This Manual

This manual explains how to use the Prism programming environment to develop, execute, debug, and analyze the performance of programs on a CM-2, CM-200, or CM-5 Connection Machine system.

Intended Audience

The manual is intended for application programmers developing programs in C (including C* and C/Paris) or Fortran (including CM Fortran and Fortran/Paris). We assume you know the basics of developing and debugging programs, as well as the basics of using a CM. 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.

Revision Information

This manual has been revised to incorporate changes made for Prism Version 1.2.

Organization of This Manual

The manual contains these chapters:

Chapter 1 **Introduction**
Gives an overview of Prism.

Chapter 2 **Using Prism**
Provides general information about using Prism.

Chapter 3 **Loading and Executing a Program**
Describes how to load and execute a program in Prism.

Related Documents

Refer to the release notes for last-minute information on Prism. The release notes are available on-line by choosing the **Release Notes** selection from the **Help** menu or by issuing the command **`help release`**.

The *Prism Reference Manual* provides reference descriptions of all Prism commands.

For general information about developing and running programs on a CM-2 or CM-200 series Connection Machine system, consult the *CM User's Guide*. For information on the CM-5, consult the manuals in the CM-5 documentation set.

For complete information about CM Fortran, consult the volume *Programming in Fortran* in the Thinking Machines Corporation documentation set. For complete information about C*, consult the volume *Programming in C**.

Notation Conventions

The table below displays the notation conventions used in this manual:

Convention	Meaning
`boldface`	Prism, UNIX, and CMOST commands, command options, and file names. Also, language elements, such as keywords, operators, and function names, when they appear embedded in text.
Ctrl-D	Simultaneous keystrokes are shown with a connecting hyphen. To type the **Ctrl-D** combination, for example, press the **D** key while holding down the **Control** key.
italics	Parameter names and placeholders in command formats.
`boldface` `typewriter`	In interactive examples, user input is shown in **`bold typewriter`** font and system output is shown in `typewriter` font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1264
Telephone:	(617) 234-4000

Chapter 1

Introduction

The Prism programming environment is an integrated graphical environment within which users can develop, execute, debug, and analyze the performance of programs written for the CM-2, CM-200, or CM-5 Connection Machine system. It provides an easy-to-use, flexible, and comprehensive set of tools for performing all aspects of Connection Machine programming. Prism operates on terminals or workstations running the X Window System. 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.

1.1 Overview

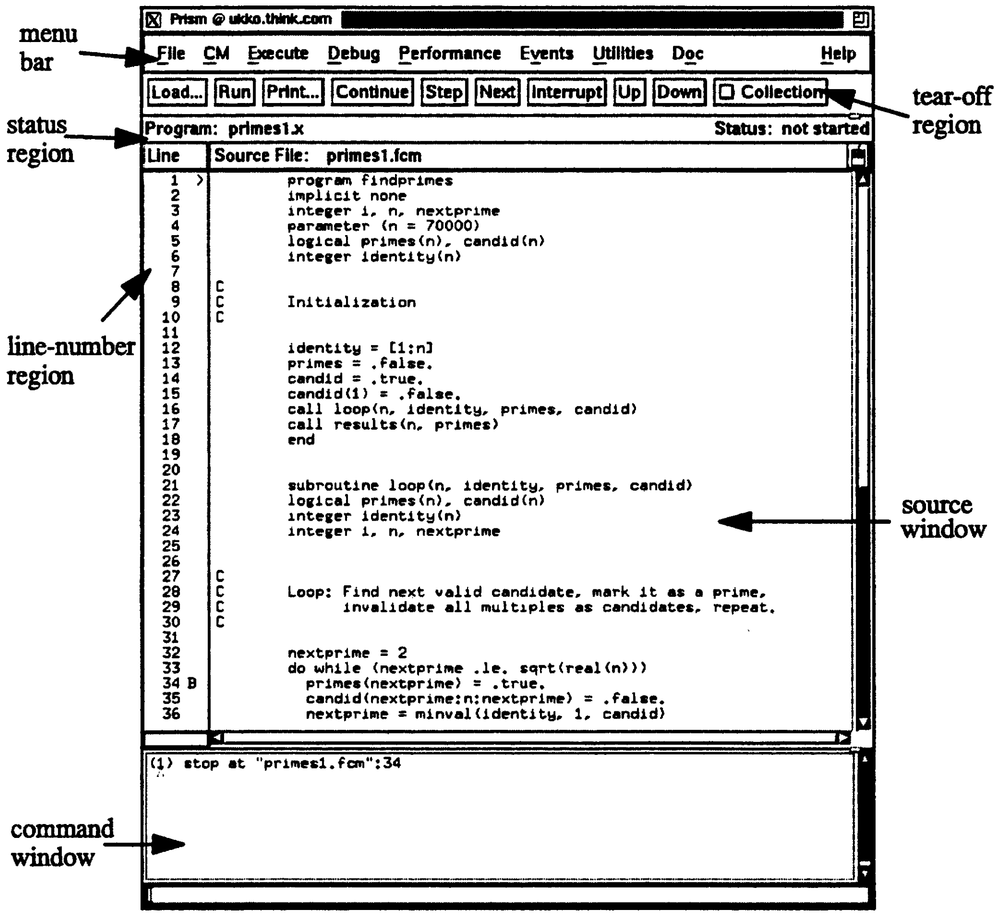
You can either load an executable program into Prism, or start from scratch by calling up an editor and a UNIX shell within Prism and using them to write and compile the program.

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

- execute the program
- debug the program
- analyze the program's performance
- visualize data

1.2 The Look and Feel of Prism

Figure 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.



```
 1 >        program findprimes
 2          implicit none
 3          integer i, n, nextprime
 4          parameter (n = 70000)
 5          logical primes(n), candid(n)
 6          integer identity(n)
 7
 8    C
 9    C     Initialization
10    C
11
12          identity = [1:n]
13          primes = .false.
14          candid = .true.
15          candid(1) = .false.
16          call loop(n, identity, primes, candid)
17          call results(n, primes)
18          end
19
20
21          subroutine loop(n, identity, primes, candid)
22          logical primes(n), candid(n)
23          integer identity(n)
24          integer i, n, nextprime
25
26
27    C
28    C     Loop: Find next valid candidate, mark it as a prime,
29    C           invalidate all multiples as candidates, repeat.
30    C
31
32          nextprime = 2
33          do while (nextprime .le. sqrt(real(n)))
34 B          primes(nextprime) = .true.
35            candid(nextprime:n:nextprime) = .false.
36            nextprime = minval(identity, 1, candid)
```

```
(1) stop at "primes1.fcm":34
```

Figure 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, a breakpoint is set at line 34.

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. CM-2 and CM-200 users can attach to and detach from a CM from within Prism.

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

Chapter 3 discusses these topics in more detail.

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.

1.5 Visualizing Data

In data parallel computing, it is often important to obtain a visual representation of the data elements that make up an array or parallel variable. 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 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 complete discussion of visualizing data.

1.6 Analyzing Program Performance

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

- processing time on the front end (for a CM-2 or CM-200) or partition manager (for a CM-5)
- processing time on the CM (for a CM-2 or CM-200) or the nodes (for a CM-5)
- time spent doing various forms of communication
- time spent performing I/O

The performance data is displayed as histograms and percentages for each computing resource. For each resource, you can also obtain data on usage for each procedure and each source line in the program. You can save the performance data in a file and re-display it at a later time.

In addition, a performance advisor provides information about and analysis of the data that Prism collects.

Performance data is available for both C* and CM Fortran programs.

See Chapter 6 for a complete 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 anything else). 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, the Help Index provides a list of entries on which you can obtain information. Clicking on an entry displays the topic in which the entry is discussed. Each topic may have a list of related topics, subtopics, terms, and commands associated with it; you can click on any of these to open a new window displaying information about the selected item.

An on-line tutorial gives you hands-on experience in using Prism on a sample program.

In addition to help on Prism itself, an interface is provided to on-line documentation for the entire Connection Machine system. You can call up a manual page for a CM command or library routine, or view the portions of the CM documentation set that are most relevant to a specific question.

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

1.9 Customizing Prism

Prism provides various ways in which you can change aspects of how it 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. To learn:

- **What to do before entering the Prism programming environment,** see Section 2.1.
- **How to enter the Prism programming environment,** see Section 2.2.
- **How to perform actions within Prism,** see Section 2.3.
- **How to use the menu bar,** see Section 2.4.
- **How to use windows, dialog boxes, and lists,** see Section 2.5.
- **How to use the source window and line-number region,** see Section 2.6.
- **How to use the command window,** see Section 2.7.
- **How to use Prism with Paris programs,** see Section 2.8. Read this section only if you are going to be running Paris programs on a CM-2 or CM-200 series Connection Machine system.
- **How to write expressions in Prism,** see Section 2.9.
- **How to issue UNIX commands,** see Section 2.10.
- **How to leave Prism,** see Section 2.11.

The best way to learn how to use Prism is to try it out for yourself. We encourage you to do this as you read this chapter. A quick way to try out some of the major features of Prism is to take its on-line tutorial. To do this, left-click with your mouse on **Help** in the menu bar at the top of the main Prism window. Then left-click on **Tutorial** in the menu. A window will appear with instructions that will guide you through loading, executing, and analyzing a sample program.

2.1 Before Entering Prism

2.1.1 Supported Languages

You can work on Fortran, CM Fortran, C, and C* programs within Prism. Performance data is available only for C* and CM Fortran programs. On the CM-5, the C* or CM Fortran program can run either on the nodes or on the vector units.

2.1.2 Compiling and Linking Your Program

Note these points in 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.
- To obtain performance data about a C* or CM Fortran program, compile it with the `-cmprofile` option. For CM-2/200 C* programs, you must not turn off optimization by compiling with the `-O0` option.
- You can combine the `-g` and `-cmprofile` options. For slicewise and CM-5 CM Fortran programs, however, this slows execution and distorts performance data. In CM Fortran programs (both slicewise and Paris) and CM-5 C* programs, you can do some debugging by specifying `-cmprofile` alone. This can cause problems in setting breakpoints and stepping through a program, however.
- To use Prism for a Paris program on a CM-2 or CM-200, you must link with the library `libprism2.a`.
- To use Prism for a C/Paris program, you must include this text on the `cc` command line:

```
-Zcc "-u _CMC_dbx"
```

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).

- To use a CM-2 or CM-200 series Connection Machine system, you must be logged in to a front end that is connected to a CM.
- To use a CM-5, you must be logged in to a CM-5 partition manager.

Prism works under these X servers:

- MIT X11R4
- NCD X11R4
- Silicon Graphics X11R4

Prism works under these window managers:

- `twm`
- `tvtwm`
- `mwm`
- `gwm`
- `olwm`
- `uwm`

Make sure 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
```

Note for CM-2 and CM-200 users: You may want to attach to the CM before entering Prism. Prism provides a utility for attaching to a CM, but you may need more flexibility than this utility provides. For information on the Prism utility, see Section 3.4. For general information on attaching to a CM, see the *CM User's Guide*.

2.2 Entering Prism

2.2.1 Invoking Prism

To enter Prism, issue the `prism` command at your UNIX prompt. When you enter Prism, you see the main window shown in Figure 1 in Chapter 1.

2.2.2 Command-Line Options

Loading a Program

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

```
% prism primes1.x
```

You can also load a process that is currently running. Add its process ID after the name of the program. For example:

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

You can obtain the process's process ID by issuing the `ps` or `cmps` command.

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

See Chapter 3 for more information on loading a program.

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. See Appendix B for information on commands-only Prism.

Specifying X Toolkit Options

You can also 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 meaningless, of course, if you use `-C` to run Prism in commands-only mode.

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 start-up. Similarly, use the form

```
% prism > logfile
```

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 Section 9.5 for information on `.prisminit`.

2.3 Within Prism

Within the Prism environment, 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.

This manual uses these terms when discussing the mouse:

- The *mouse pointer* is the graphical image that appears on the screen and represents the current location of the mouse. Mouse actions are generally related to the location of the mouse pointer.
- *Left-clicking* refers to pressing the left mouse button, *right-clicking* refers to pressing the right button, etc. If we don't mention which button to click, the left button is assumed.
- To *double-click* the mouse is to press the left mouse button twice in rapid succession. This is a shortcut for selecting items in lists.
- To *drag* the mouse pointer is to move it while holding down a mouse button; you do this, for example, to select text in the source window. To *slide* the mouse pointer is to move it without holding down a mouse button.

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:

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.
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	**Shift-Tab** performs the same function as **Tab**, but moves 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 (for example, help windows), 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 a mnemonic (see below) to display a menu. 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.

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:

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.

In addition, you can use *mnemonics* and *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 Appendix A for a list of all 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 UNIX debugger **dbx**. There are differences, however; we recommend that you 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; see Figure 2.

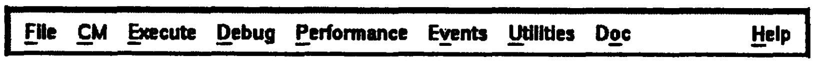


Figure 2. The menu bar.

Each title is associated with a pull-down menu, from which you can perform actions within Prism; see Figure 3 for an example. (The **CM** title appears only if you are running Prism from a CM-2 or CM-200 front end.)

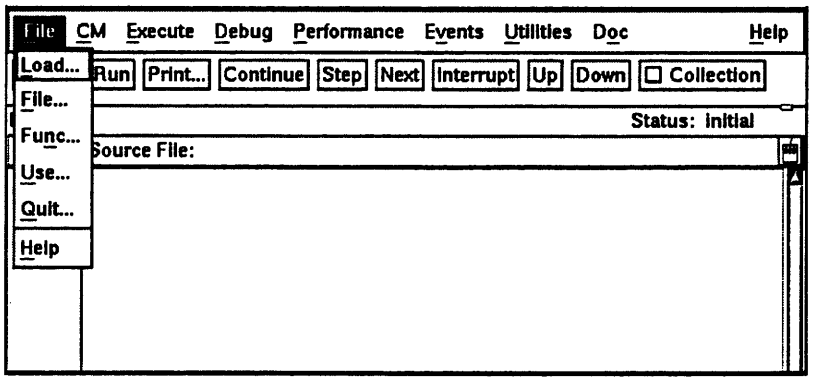


Figure 3. A Prism menu.

2.4.1 With a Mouse

To display the pulldown menu associated with a title in the menu bar, left-click on the title with your mouse. The menu appears beneath the title. To browse through the menus, drag the mouse pointer over the menu-bar titles; each menu appears in turn.

There are two ways to choose an item from a menu:

- Drag the mouse pointer until it is on the selection you want, and then let go.
- Slide the pointer onto the selection you want, and then left-click the mouse.

To make a menu disappear, either drag the mouse pointer off the menu and then release the mouse button, or slide the mouse pointer off the menu and then left-click the mouse.

2.4.2 From the Keyboard

To go to the menu bar, press the **F10** key.

When the location cursor is in the menu bar, you can use the left and right arrow keys to display the menus under each title. To select an item from a menu, use the up and down arrow keys to move through the items until the one you want is highlighted. Then press **Return** or the spacebar to choose it.

Another way to display menus from the keyboard is to type the mnemonic associated with the menu, as described below.

Mnemonics

Mnemonics are a quick way of displaying menus, or of choosing items from a menu, by simply typing a letter.

Each title on the menu bar, and each item in their pulldown menus, has a letter underlined. For example:

File

The underlined letter is the mnemonic for the item. Typing the mnemonic is equivalent to clicking on the item with the mouse. For example, typing **F** from the menu bar displays the **File** pulldown menu. The menu stays on the screen until you make a choice from it or press the **Esc** key to cancel it.

Note these points in using mnemonics:

- To use a mnemonic for a menu bar, the location cursor must be in the menu-bar region. If the cursor is outside this region, press the **F10** key to move the location cursor to the menu bar. Or press the **Meta** key along with the mnemonic.
- The mnemonics for items in a pulldown menu work only when the menu is displayed. Thus, to display the dialog box for loading a file, type **F** (in the menu bar region) to display the **File** menu, and then type **L** to display the **Load** dialog box.

To perform a function directly, without displaying the pulldown menu, you can use keyboard accelerators instead.

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 are:

Ctrl-F1	**Run**
Ctrl-F2	**Continue**
Ctrl-F3	**Interrupt**
Ctrl-F4	**Step**
Ctrl-F5	**Next**
Ctrl-F6	**Where**
Ctrl-F7	**Up**
Ctrl-F8	**Down**
Ctrl-F9	**Collection**

2.4.3 What Happens When You Choose an Item

When you choose an item, Prism performs the action associated with the item. If it needs more information, it displays a window or dialog box, which you use to provide Prism with the information. If a menu selection has "..." after it, a window or dialog box is displayed when you choose it. Later chapters in this guide describe the actions associated with specific titles in the menu bar.

2.5 Using Windows, Dialog Boxes, and Lists

This section discusses general aspects of using windows, dialog boxes, and scrollable lists in Prism. If you are familiar with other X applications, you can skip this section.

2.5.1 Windows

Windows are used to display information that you may want to keep on your screen; they also sometimes request information from you. The main Prism display is a window. The source window and the command window are *panes* within this main window. Other windows are displayed as the result of choosing menu items.

Prism does not determine how window operations are performed (except for panes — like the source window — that appear within other Prism windows); this is under the control of the window manager for your X Window System. Different window managers let you operate windows in different ways; check the documentation for your window manager, or ask your system administrator for more information. Generally, you will be able to perform these actions:

- **Delete the window.** It is possible to delete windows. It is also possible to "destroy" a window. *Do not destroy any Prism windows.* This will kill Prism.

- **Turn the window into an icon.** An *icon* is a small graphical image representing a window. Often you can close a window by clicking on a box in the upper left corner; this creates an icon. You can then click on the icon if you want to open it again.

- **Move the window.** Often you can do this by clicking on a horizontal bar along the top of the window, then dragging the window to its new location.

- **Change the size of the window.** Often you can do this by clicking on the small *resize box* in a corner of the window, then dragging the window until it is the size you want.

- **Scroll through the window,** if the text is larger than the size of the window. Typically there is a *scroll bar* at one side of the window; The scroll bar has arrows at the top and bottom; you can click on the arrows to move the text in the window up or down. The scroll bar also has an *elevator* that shows where the text is located in terms of the total text that can be displayed. For example, the elevator is at the top when the beginning text is displayed. You can drag this elevator to scroll quickly through the text. You can also click above or below the elevator to move up or down a page.

  There can also be a *horizontal* scroll bar that lets you scroll through text that is too wide to be displayed in the window.

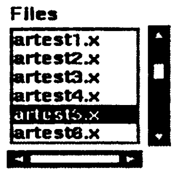


Figure 4. Scroll bars.

2.5.2 Dialog Boxes

Prism uses dialog boxes to obtain information from you. Once you have provided the information, the dialog box goes away. For example, the **Run (args)** selection from the **Execute** menu displays a dialog box. You can close it without providing information by clicking on **Cancel.**

Your window manager may also let you perform actions such as moving a dialog box or turning it into an icon.

If a dialog box gets buried beneath windows, reissue the command that displayed the dialog box; the box will reappear at the front of your screen.

2.5.3 Lists

Many windows and dialog boxes in Prism contain lists that you can scroll through (for example, in choosing a program to load). If the list is too long for the space provided, there is a scroll bar to the right, which operates as described in Section 2.5.1. You can also use the up and down arrow keys to move through the list.

If names in the list are too long for the space provided (for example, because files have long pathnames), there is a scroll bar beneath the list; you can use this scroll bar to scroll horizontally. You can also use the left and right arrow keys to scroll across the line.

Typically, you will be selecting an item from the list. To do this, first highlight it by clicking on it or moving to it with the up or down arrow key; then click on the **Select** button (or its equivalent), or press the **Return** key. Or you can simply double-click, rapidly, on the item.

2.6 Using the Source Window and Line-Number Region

2.6.1 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 filename 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 upper 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.

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 do this, the focus must be in the *command* window.) 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 you specify, and repositions the file at the first occurrence it finds. The *?string* command searches backward in the file for the string you specify.

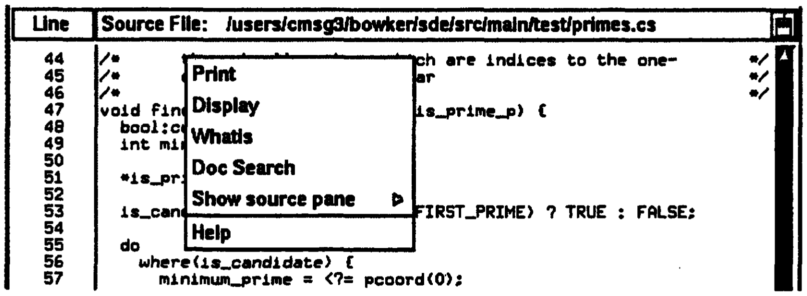
You can display different files by choosing the **File** or **Func** selection from the **File** menu; see Section 3.6. You can also move *between* files. Prism keeps a list of the files you have displayed. With the mouse pointer in the source window, do this to move through the list:

- 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.

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 5. 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.


Line
Source File: /users/cmsg3/bowker/sde/src/main/test/primes.cs
Print
Display
WhatIs
Doc Search
Show source pane
Help

```
/*                          ch are indices to the one-        */
/*                          ar                                */
/*                                                            */
void fin                    is_prime_p) {
  bool:c
  int mi

  *is_pr

  is_can                    FIRST_PRIME) ? TRUE : FALSE;

  do
    where(is_candidate) {
      minimum_prime = <?= pcoord(0);
```

Figure 5. The popup menu in the source window.

You can print the value of a variable or expression directly from the source window, without displaying the menu. To do this, press the **Shift** key while selecting the variable or expression. A visualizer is displayed, showing the value(s) of the variable or expression.

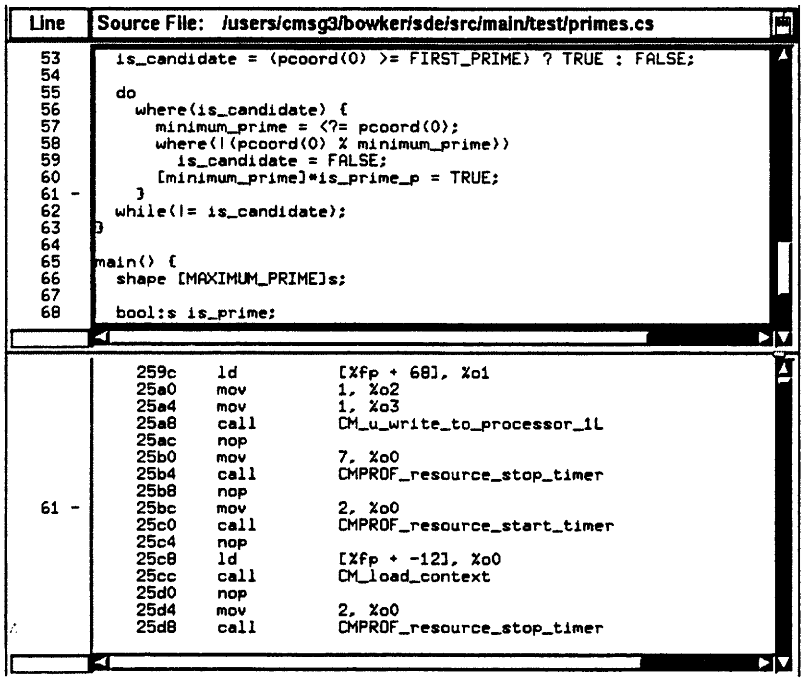
You can display the definition of a function by pressing the **Shift** key while selecting the name of the function in the source window. This is equivalent to choosing the **Func** selection from the **File** menu and selecting the name of the function from the list; see Chapter 3. Do not include the 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, as described above.
3. Click on **Show source pane** in the popup menu.
4. This displays another menu. 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 6.



```
Line    Source File:  /users/cmsg3/bowker/sde/src/main/test/primes.cs
 53       is_candidate = (pcoord(0) >= FIRST_PRIME) ? TRUE : FALSE;
 54
 55       do
 56         where(is_candidate) {
 57           minimum_prime = <?= pcoord(0);
 58           where(!(pcoord(0) % minimum_prime))
 59             is_candidate = FALSE;
 60           [minimum_prime]*is_prime_p = TRUE;
 61 -       }
 62       while(!= is_candidate);
 63     }
 64
 65     main() {
 66       shape [MAXIMUM_PRIME]s;
 67
 68       bool:s is_prime;
```

```
             259c    ld        [%fp + 68], %o1
             25a0    mov       1, %o2
             25a4    mov       1, %o3
             25a8    call      CM_u_write_to_processor_1L
             25ac    nop
             25b0    mov       7, %o0
             25b4    call      CMPROF_resource_stop_timer
             25b8    nop
 61 -        25bc    mov       2, %o0
             25c0    call      CMPROF_resource_start_timer
             25c4    nop
             25c8    ld        [%fp + -12], %o0
             25cc    call      CM_load_context
             25d0    nop
             25d4    mov       2, %o0
             25d8    call      CMPROF_resource_stop_timer
```

Figure 6. A 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.

NOTE: If you have used the CMAX Converter to translate a Fortran 77 program into CM Fortran, you can use the Prism's split-screen technique to view the Fortran 77 version of a CM Fortran program, as well as its assembly code. The use of Prism with CMAX is described in Appendix C.

2.6.2 The Line-Number Region

The line-number region shows the line numbers associated with the source code displayed in the source window. Figure 7 shows a portion of a line-number region, with a breakpoint set.

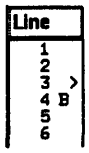


Figure 7. The line-number region.

The > symbol in the line-number region in Figure 7 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 Chapter 4.

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

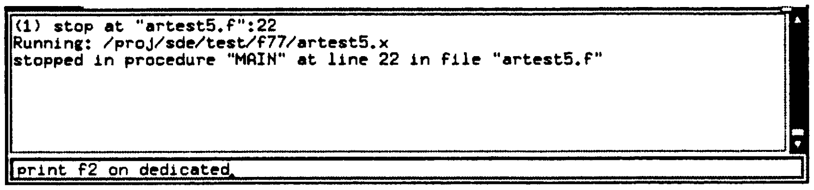
- The – symbol is the *scope pointer*; it indicates the current source position (that is, the scope). Prism uses the current source position to interpret names of variables. When you scroll through source code, the scope pointer moves to the middle line of the code that is displayed. Various Prism commands 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 8 shows a command window, with a command on the command line and messages in the history region.



```
(1) stop at "artest5.f":22
Running: /proj/sde/test/f77/artest5.x
stopped in procedure "MAIN" at line 22 in file "artest5.f"
```

```
print f2 on dedicated
```

Figure 8. The command window.

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.

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 filename as an argument, you cannot follow it with a semicolon, because Prism can't tell if the semicolon is part of the filename.

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 is 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** (abbreviation **com**). This sends output to the command window; this is the default.
- **dedicated** (abbreviation **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** (abbreviation **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: **cmcoldboot**, **cmfinger**, **edit**, **email**, **make**, and **sh**.

2.7.4 Logging Commands and Output

As we 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. 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 we 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 Using Prism with Paris Programs

NOTE: Read this section only if you are going to run Paris programs on a CM-2 or CM-200 series Connection Machine system.

You can work on Paris programs in the Prism programming environment. However, before printing a Paris parallel variable, array, pointer, or structure, or using it in an expression, you must tell Prism its type. For example, if you define a variable of length 32 and use it in Paris routines dealing with `floats`, you need to tell Prism that the variable represents a `float`.

There are two ways of defining types:

From the source window: (This method does not work for structures, pointers, or arrays.) Select the variable in the source window, as described in Section 2.6.1. Then right-click to display a popup menu. Choose **Define Type** from this menu. (NOTE: **Define Type** appears in this menu only if you are logged in to a CM-2 or CM-200 front end.) Choosing **Define Type** displays another menu, listing C types. Click on the type of the variable. You can then operate on the variable within Prism.

From the command window: Issue the **type** command, specifying the type of the variable, structure, pointer, or array. For example,

```
type unsigned int a
```

specifies that `a` is a parallel `unsigned int`.

```
type struct foo bar
```

specifies that `bar` is a parallel `struct` of type `foo`.

```
type float array_a[10][20]
```

specifies that `array_a` is a parallel 10-by-20 array of `floats`.

```
type int *ptr
```

specifies that `ptr` is a pointer to a parallel `int`.

As mentioned above, you have to define the types only once during a Prism session. To avoid having to do this during every session, put the appropriate **type** commands in your `.prisminit` file, or in a special initialization file for each Paris program you are working on; see Section 2.7.5 to learn how to use such initialization files.

2.9 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.

2.9.1 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:

1. 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, 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 3.6.

2. 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.

3. 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.

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 leftmost symbol in the name is the file, followed optionally by the procedure, followed by the variable name. Each is preceded by a `. 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 leftmost name 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.

2.9.2 Using Fortran Intrinsic Functions in Expressions

Prism supports the use of a subset of Fortran intrinsic functions in writing expressions; the intrinsics work for all languages that Prism supports, except as noted below. For complete information on the intrinsics, see the *CM Fortran Reference Manual.*

The intrinsics, along with the supported arguments, are:

- `AIMAG`(*complex number*) — Returns the imaginary part of a complex number. Works for Fortran and CM Fortran only.

- `ALL` (*logical array*) — Determines whether all elements are true in a logical array. Works for Fortran and CM Fortran only.
- `ANY` (*logical array*) — Determines whether any elements are true in a logical array. Works for Fortran and CM 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 and CM Fortran only.
- `DSIZE` (*array*) — Counts the total number of elements in the array.
- `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.
- `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 and CM Fortran only.
- `SUM` (*array*) — Computes the sum of all elements of a numeric array.

The intrinsics can be either upper- or lowercase.

2.9.3 Writing C* Expressions

Prism currently does not parse many parts of the C* language. Here are some of the features of C* that Prism doesn't recognize:

- The functions `pcoord`, `shapeof`, `rankof`, and `dimof` are not supported.
- Dot notation is not understood.
- Shapes are not known objects.
- Current context isn't respected or understood.

- Reduction operators are not supported, except as listed below.
- The communication intrinsic functions are not supported.
- The auto-increment and auto-decrement operators are not understood.

Using C* Reduction Operators

Prism allows the use of these C* reduction operators in writing expressions:

`>?=`	Computes the maximum value of an array or parallel variable.
`<?=`	Computes the minimum value of an array or parallel variable.
`*=`	Computes the product of all elements of an array or parallel variable.
`+=`	Computes the sum of all elements of an array or parallel variable.

You can also use certain Fortran intrinsics, as described in Section 2.9.2.

Passing Parallel Variables to Functions

If you call a C* function in Prism, you must pass parallel variables to it by reference, not by value. For example, if you have a parallel integer `p1`, you would pass it to function `f` as follows:

```
f(&p1)
```

Using Array-Section Syntax in C* Left Indexes

In Prism, you can use left indexes as you normally would to specify an element of a C* parallel variable. For example:

```
[112][147]pvar1
```

In addition, you can use array-section syntax from Fortran 90 to specify a range of elements. This syntax is useful, for example, if you want to print the values of only a subset of the elements of a parallel variable. The syntax is:

lower-bound: *upper-bound*: *stride*

where:

lower-bound	is the lowest-numbered coordinate you choose along the axis; it defaults to 0.
upper-bound	is the highest-numbered coordinate you choose along the axis; it defaults to the highest-numbered coordinate for the axis.
stride	is the increment by which elements are chosen between the lower bound and upper bound; it defaults to 1.

For example,

```
[5:25:5]pvar1
```

specifies elements 5, 10, 15, 20, and 25 of the parallel variable **pvar1**.

```
[0:10][100:110:2]pvar2
```

specifies the elements of the parallel variable **pvar2** that have coordinates 0 through 10 along axis 0 and coordinates 100, 102, 104, 106, 108, and 110 along axis 1.

For more information about array sections, see the *CM Fortran Programming Guide*.

2.9.4 Using C and C* Arrays in Expressions

Prism handles arrays slightly differently from the way C and C* handle them.

In a C or 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 and 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 and C* arrays in Prism. (The **print** command and visualizers are discussed in Chapter 5.)

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

```
print &a + 2
```

2.9.5 Hints for Detecting NaNs and Infinities

Prism provides expressions 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)
```

2.10 Issuing UNIX Commands

You can issue UNIX and CMOST commands from within Prism.

From the menu bar: Choose the **Shell** selection from the **Utilities** menu. Prism creates a UNIX shell. The shell is independent of Prism; you can issue UNIX commands from it just as you would from any UNIX 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 UNIX shell. If you include a UNIX command line as an argument, the command is executed, and the results are displayed in the history region.

Some UNIX 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 UNIX. For example,

```
cd /allen/bin
```

changes your working directory to `/allen/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 filenames 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 UNIX.

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 UNIX.

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
```

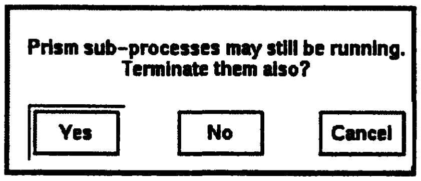
prints the current setting of the `EDITOR` environment variable. Or, issue `printenv` or `setenv` with no arguments to print the settings of all your environment variables.

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 UNIX shell), Prism displays this message before exiting:


Prism sub-processes may still be running.
Terminate them also?

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. To learn:

- **How to load a program into Prism,** see Section 3.1.
- **How to associate a core file with a loaded program,** see Section 3.2.
- **How to attach to and detach from a running process,** see Section 3.3.
- **How to attach to a CM-2 or CM-200 and perform other CM-related tasks,** see Section 3.4. This section is for CM-2 and CM-200 users only.
- **How to execute a program,** see Section 3.5.
- **How to change the current file and the current function,** see Section 3.6.
- **How to specify the directories to be searched for source files,** see Section 3.7.

We assume in this chapter that you already have an executable program that you want to run within Prism. You can also develop the program from scratch by calling up an editor within Prism; see Chapter 7.

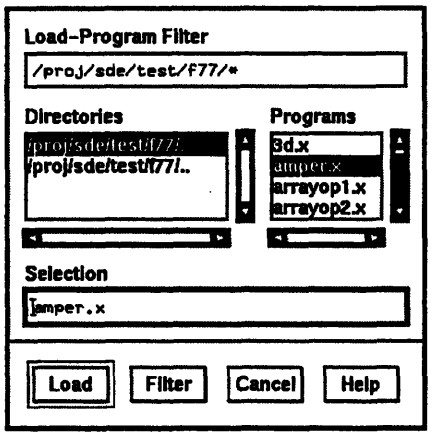
3.1 Loading a Program

Before you can execute, debug, or analyze the performance of 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 described below.

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 9.



Load-Program Filter: /proj/sde/test/f77/*

Directories:
/proj/sde/test/f77/.
/proj/sde/test/f77/..

Programs:
3d.x
amper.x
arrayop1.x
arrayop2.x

Selection: amper.x

Figure 9. The Load dialog box.

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 pathname in the **Selection** box, then click on **Load**. To put the file's pathname 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 filenames in the **Programs** list; the box uses standard UNIX 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 programs in *directory-name* are to be displayed. Click on **Filter** to display the filenames 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, analyze, 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 3.7.

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; if you were running the program on a CM-2 or CM-200, you cannot print the values of parallel variables or CM-resident arrays.

3.3 Attaching to and Detaching from a Running Process

As we described in Chapter 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. To do this:

1. Find out the process's process ID by issuing the UNIX command `ps` (or `cmps` if the process is running on a CM-5 or under timesharing on a CM-2 or CM-200).

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 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 3.5.2.

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.

3.4 Attaching to and Detaching from a CM-2 or CM-200

NOTE: Read this section only if you are going to run programs on a CM-2 or CM-200 series Connection Machine system.

To run a program on a CM-2 or CM-200, you must be attached to a CM resource (that is, one or more sequencers on the CM). You can attach in any of the standard ways. For example, you can issue the `cmattach` command before entering Prism, and then issue the `prism` command from a `cmattach` subshell.

You can also attach, detach, cold boot, turn Paris safety on and off, and obtain information about CM users from within Prism. To attach, you must be running Prism from a front end that is physically connected to a CM-2 or CM-200 series Connection Machine system. The CM menu discussed in this section does not appear if you are not running Prism from a front end.

For complete information on attaching, detaching, cold booting, and turning safety on and off, see the *CM User's Guide*.

3.4.1 Attaching from within Prism

You can attach to a CM resource from within Prism. Attaching automatically cold boots the resource.

From the menu bar: Choose the **CMattach** selection from the **CM** menu. A dialog box is displayed; see Figure 10. By default, you are attached to the highest-numbered sequencer available on the first CM available, whether it is timeshared or exclusive.

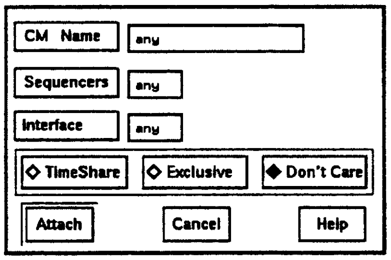


Figure 10. The CMattach dialog box.

In the **CM Name** text-entry box, specify the name of the CM to which you want to attach. The default is **any**, meaning that you will accept any available CM. Left-click on **CM Name** to display a menu of available CMs. You can either click on a choice from this menu or type the name of the CM in the box.

In the **Sequencers** box, specify the number(s) of the sequencer(s) to which you want to attach. The default is **any**, meaning that you will accept any available sequencer(s). Left-click on **Sequencers** to display a menu of legal sequencer sets. You can either click on a choice from this menu or type your choice in the box. If you choose **any** in both boxes, you are attached to the highest-numbered sequencer on the first CM that is available.

In the **Interface** box, specify the number of the front-end bus interface by which you want to attach to the CM. Once again, the default is **any**, meaning that you will accept any available interface. Left-click on **Interface** to display a menu of interfaces. You can either click on a choice from this menu or type your choice in the box.

Click on **TimeShare** if you want to attach to a timeshared CM resource. Click on **Exclusive** if you want to attach to a CM resource running in exclusive mode. In both cases, the diamond next to the word turns black, indicating your choice. **Don't Care** is the default choice; it means that you will take whatever resource is available, whether it is timeshared or exclusive. You can choose only one of the three.

When you have specified the information, click on **Attach** to attach to the CM resource (if it is available). You receive messages in the history region of the command window telling you if the attach was successful. You may also receive messages from CM System Software in the xterm from which you started Prism.

Click on **Cancel** or press the **Esc** key to close the dialog box without attempting to attach.

Note that this dialog box gives you less control over the characteristics of the CM resource than other methods. For example, you cannot specify the version of the microcode that the resource is to run. For maximum control over the characteristics of the CM resource, you must attach from within your program or by calling `cmattach` before you enter Prism.

From the command window: Issue the `cmattach` command to attach to a CM. By default you are attached to the highest-numbered sequencer available on the first CM available. The Prism version of `cmattach` accepts this subset of the shell-level `cmattach` options:

`-e`	Attach in exclusive mode only.
`-t`	Attach in timeshared mode only.
`-C` *name*	Attach to the specified CM.
`-i` *number*	Attach via the specified interface.
`-S` *seq-set*	Attach to the specified sequencer or sequencer set.

Auto-attaching in Prism

As of Version 6.1 of CM System Software, a program can attach automatically to an available CM resource, run, and then detach. This works in Prism — with one complication. If an auto-attached program has a segmentation fault, it will remain attached to the CM; this allows you to examine the values of CM-resident data. To detach, you must issue this command from the command line to force the program to exit:

```
cont kill
```

(This also kills the program.) You cannot use the **CMdetach** selection described below. **CMdetach** detaches Prism from a CM; it cannot detach an auto-attached program.

3.4.2 Detaching from within Prism

You can detach from a CM resource from within Prism, whether or not you attached to it from within Prism.

From the menu bar: Choose the **CMdetach** selection from the **CM** menu. A dialog box is displayed, asking if you really want to detach. Click on **Detach** to proceed. Click on **Cancel** or press the **Esc** key to close the dialog box without detaching.

You receive a message in the history region of the command window when you are detached.

From the command window: Issue the `cmdetach` command to detach from the CM resource; in this case, you are not asked if you are sure you want to detach.

3.4.3 Cold Booting

To cold boot the CM resource to which you are attached, choose the **CMcoldboot** selection from the **CM** menu, or issue the `cmcoldboot` command from the command window. The Prism version of `cmcoldboot` accepts all the options supported by the shell-level `cmcoldboot`.

3.4.4 Turning Safety On and Off

To turn Paris safety on or off for the **CM** resource to which you are attached, choose the **CMsetsafety** selection from the **CM** menu. This is a toggle switch. Choosing the selection fills in the toggle box to the left of the menu selection; safety is now on. Choosing it again turns safety back off. From the command window, issue `cmsetsafety on` to turn safety on, and `cmsetsafety off` to turn it off.

Note that safety should be turned off when you are collecting performance data; otherwise your results will be inaccurate. If you attempt to run a program with collection and safety both on, Prism turns safety off and displays a message informing you of this. (It will then turn safety back on when you turn collection off.)

3.4.5 Obtaining Information about CM Users

Choose the **CMfinger** selection from the **CM** menu to display information about what CMs are available and who is using them.

From the command window, issue the `cmfinger` command to obtain the same output.

3.5 Executing a Program

To execute a program, you must first load it, as described in Section 3.1. If the program uses a CM-2 or CM-200, you must be attached; see Section 3.4. Once you start the program running, you can step through it, and interrupt and continue execution.

3.5.1 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 11. If you have more arguments than fit in the input box, they scroll to the left. Click on the **Run** button to start execution.

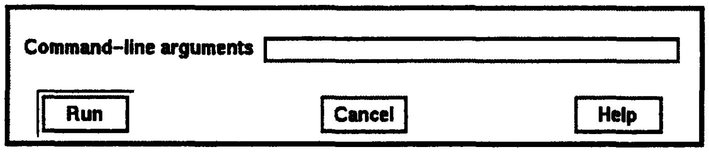


Figure 11. The Run (args) dialog box.

- **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`.

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.

3.5.2 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.

3.5.3 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.)

NOTE: If you compiled your CM Fortran program with the `-cmprofile` option instead of `-g`, a single step may execute several lines of code; this is caused by the way `-cmprofile` creates the symbol table information.

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.

3.5.4 Interrupting and Continuing Execution

To interrupt execution, choose **Interrupt** from the **Execute** menu or type **Ctrl-c**. The status changes to **stopped**, 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.

3.5.5 Status Messages

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

Table 1. Status messages.

Message	Displayed when:
initial	Prism starts up without a program loaded.
loading	Prism is loading a program.
not started	A program is loaded but not yet started.
running	A program is running.
stopped	A program has stopped at a breakpoint or signal.
terminated	A program has run to completion and the process has gone away.

3.6 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 2.9.1 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 filename in the **Selection** box.

  NOTE: The **File** window displays only files compiled with the `-g` switch.

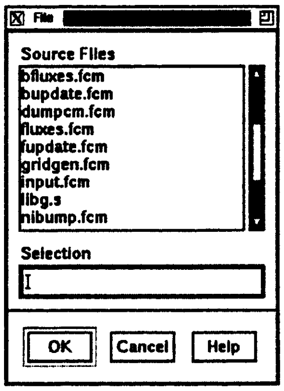


Figure 12. The File window.

- **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. (CM 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.

3.7 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 13. To add a directory, type its pathname in the **Directory** box, then click on **Add.** To remove a directory, click on it in the directory list; its pathname appears in the **Directory** box; then click on **Remove.**

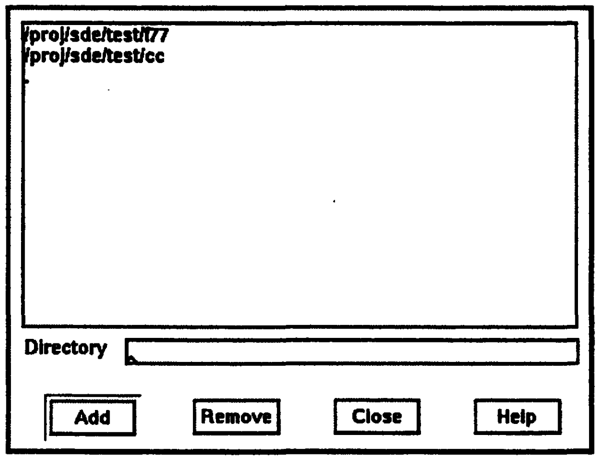

/proj/sde/test/f77
/proj/sde/test/cc
.

Figure 13. The 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 is, 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 perform certain basic kinds of debugging in Prism. It also describes how to use *events* to control the execution of a program.

To learn:

- **What events are,** see Section 4.1, below.
- **How to use the event table,** see Section 4.2.
- **How to set breakpoints,** see Section 4.3.
- **How to trace program execution,** see Section 4.4.
- **How to display and move through the call stack,** see Section 4.5.
- **How to examine the contents of memory and registers,** see Section 4.6.
- **How to obtain an interface to pndbx to do node-level debugging on a CM-5,** see Section 4.7.

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.

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.

These 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

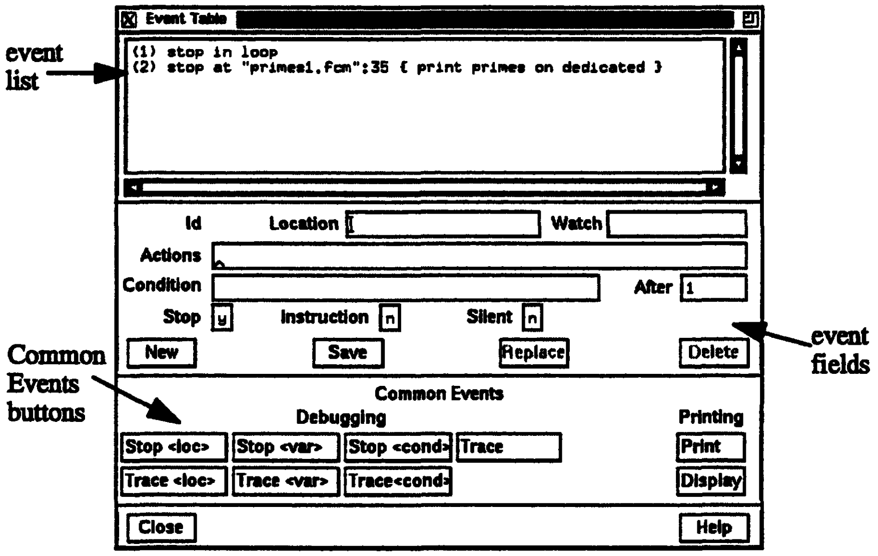
You display the event table by choosing the **Event Table** selection from the **Events** menu.

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

4.2.1 Description of the Event Table

Figure 14 shows the 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 14, the events have been assigned the IDs 1 and 2.



```
(1) stop in loop
(2) stop at "primes1.fcm":35 { print primes on dedicated }
```

After: 1
Stop: y
Instruction: n
Silent: n

Figure 14. The event table.

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"* : *linenumber* 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 2.9 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.
- **Instruction.** 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.

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 from scratch, 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 from scratch:

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 Saving Events

Events that you create for a program are deleted when you load another program, reload the current program, or leave Prism. You can use Prism commands to save your events to a file, and then execute them from the file rather than individually. Follow this procedure:

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.

NOTE: If you compiled your CM Fortran program with the **`-cmprofile`** option instead of **`-g`**, you may be unable to set breakpoints at certain lines.

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.

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.

See Section 2.6.2 for more information on the line-number region.

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.

What Happens in a Split Source Window

As described in Section 2.6.1, you can split the source window to display source code and the corresponding assembly code, or (if you are using CMAX) CM Fortran code and the corresponding Fortran 77 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.

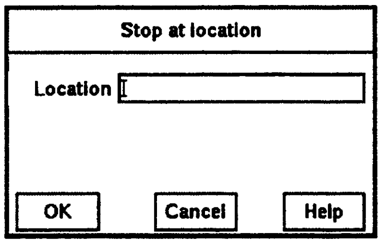


Figure 15. The Stop <loc> dialog box.

- **Stop <var>** prompts for a variable name. The program stops when the variable's value changes. The variable can be an array, array section, or a parallel variable, in which case execution stops any time any element of the array or variable 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 2.9 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.

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` "*file-name*" : *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. 10`

This form of the command slows execution considerably, unless you combine it with the `at` *line* syntax. See Section 2.9 for more information on writing expressions in Prism.

cmd is any Prism command (except `attach`, `core`, `detach`, `load`, `return`, `run`, and `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.

Examples

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

Stop execution the tenth time in function `foo` and print `a`.

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

Stop at line 17 of file `bar` if `a` is equal to 0.

```
stop a
```

Stop whenever **a** changes.

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

Stop the third time **a** equals 5.

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

Every time the program stops execution, print **a** and do a stack trace.

For Machine Instructions

To set a breakpoint at a front-end or partition-manager 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.

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. 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. (For CM Fortran programs, it steps into procedures *unless* they are in files compiled with the `-nodebug` option.)

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 2.9 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.

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*

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.

Deleting Traces via the Command Window

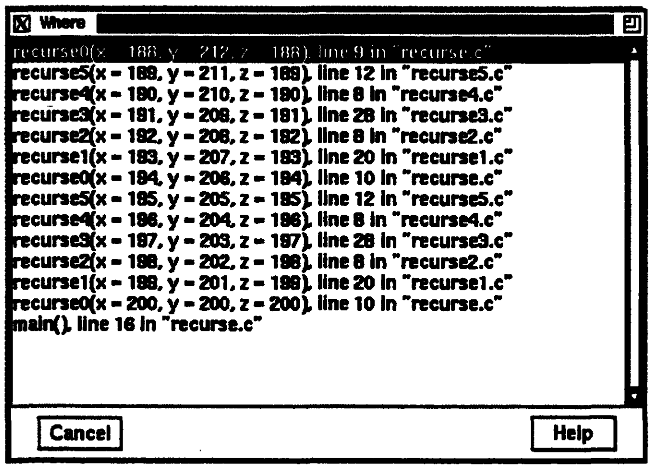
To delete a trace, use the **`show events`** command to obtain the ID associated with the trace, then issue the **`delete`** command with the ID as its argument. See Section 4.3.3.

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.

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 16. The window contains the call stack; it is updated automatically when execution stops or when you issue commands that change the stack.



```
recurse0(x = 188, y = 212, z = 188), line 9 in "recurse.c"
recurse5(x = 189, y = 211, z = 189), line 12 in "recurse5.c"
recurse4(x = 190, y = 210, z = 190), line 8 in "recurse4.c"
recurse3(x = 191, y = 209, z = 191), line 28 in "recurse3.c"
recurse2(x = 192, y = 208, z = 192), line 8 in "recurse2.c"
recurse1(x = 193, y = 207, z = 193), line 20 in "recurse1.c"
recurse0(x = 194, y = 206, z = 194), line 10 in "recurse.c"
recurse5(x = 195, y = 205, z = 195), line 12 in "recurse5.c"
recurse4(x = 196, y = 204, z = 196), line 8 in "recurse4.c"
recurse3(x = 197, y = 203, z = 197), line 28 in "recurse3.c"
recurse2(x = 198, y = 202, z = 198), line 8 in "recurse2.c"
recurse1(x = 199, y = 201, z = 199), line 20 in "recurse1.c"
recurse0(x = 200, y = 200, z = 200), line 10 in "recurse.c"
main(), line 16 in "recurse.c"
```

Figure 16. The 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.

Note for CM Fortran users: When a CM Fortran program starts up on a CM-2 or CM-200, the operating system calls a runtime routine named **`main`**. This routine does some initialization, and then calls **`MAIN`**, which is the name of your CM Fortran main program. Prism displays this **`main`** initialization routine as the first routine in the call stack. On the CM-5, the call to **`main`** is preceded by a call to a routine named **`CMTS_ScalarMain`**.

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 previous format you specified.

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 **f0** register.

Register names for a SPARC are:

$pc	program counter
$np	next program counter
$fsr	floating status register
$fq	floating queue
$wim	window invalid mask
$tbr	trap base register
$g0-$g7	global registers
$i0-$i7	input registers
$l0-$l7	local registers
$o0-$o7	output registers
$sp	synonym for **$o6**
$fp	synonym for **$i6**

$f0-$f31	floating-point registers
$y	Y register

4.7 Using pndbx on a CM-5

For CM-5 users, Prism provides an interface to the node-level debugger **pndbx**.

To start **pndbx**, choose **PN Debug** from the **Utilities** menu (this selection appears only if you are running Prism from a CM-5 partition manager). The toggle box next to **PN Debug** is filled in, indicating that node-level debugging is turned on. If a program is running, Prism starts up **pndbx** in a separate window; if a program isn't running, a message is displayed in the command window informing you that **pndbx** will start up the next time you run a program. The interface to **pndbx** will be created each time you run a program until you choose **PN Debug** again to turn it off.

For information on using **pndbx**, see the *CMMD User's Guide*.

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.

Section 5.1 is an overview of visualizing data. To learn:

- **How to choose the variable or expression whose values are to be visualized,** see Section 5.2.
- **How to work with graphical visualizers,** see Section 5.3.
- **How to visualize structures and pointers,** see Section 5.4.
- **How to print the type of a variable,** see Section 5.5.
- **How to change the values of a variable,** see Section 5.6.
- **How to change the radix of data,** see Section 5.7.

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. The data can reside on either the front end or the CM (for a CM-2 or CM-200), or on the partition manager or the nodes (for a CM-5). 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 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 5.2.4)

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 Limitations

Note these limitations on visualizing data:

- You occasionally cannot print or display a CM-resident variable or array after interrupting execution; you receive an error message if you try. If this occurs, you must do a **Step** or **Next** after the interrupt before attempting to print.

- 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.

- Recall that you must define the type of a Paris parallel variable before you can print or display its values. (See Section 2.8.)

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 17 shows an example of the **Print** dialog box.

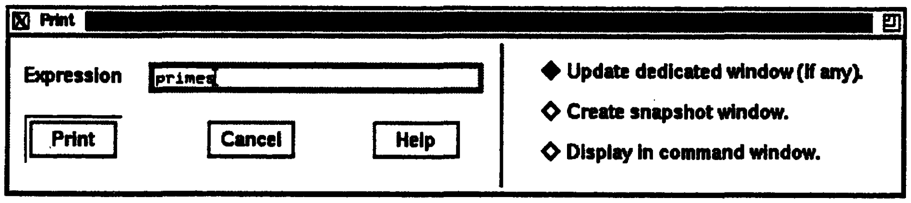

Print
Expression
primes
Print
Cancel
Help
◆ Update dedicated window (if any).
◇ Create snapshot window.
◇ Display in command window.

Figure 17. The Print dialog box from the Debug menu.

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 popup 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.

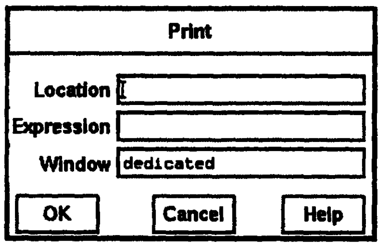
NOTE: Prism prints the correct variable when you choose it in this way, even if the scope pointer sets a scope that contains another variable of the same name.

5.2.3 Printing and Displaying from the Event Table and the Events Menu

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

From the Events Menu

The **Print** dialog box (see Figure 18) 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.



Window: dedicated

Figure 18. The Print dialog box from the Events menu.

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.

From the Event Table

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.

5.2.4 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 on dedicated
```

create two 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 complete list of register names.

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. In CM Fortran, the expression can operate on a conformable array. In C*, it can operate on a parallel variable of the same shape as the variable 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.

In a C* program where **pvar1** and **pvar2** are of the same shape,

```
where (pvar1 > 0) display pvar2 on dedicated
```

displays as active only the elements of **pvar2** for which the value of the corresponding element of **pvar1** is greater than 0.

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

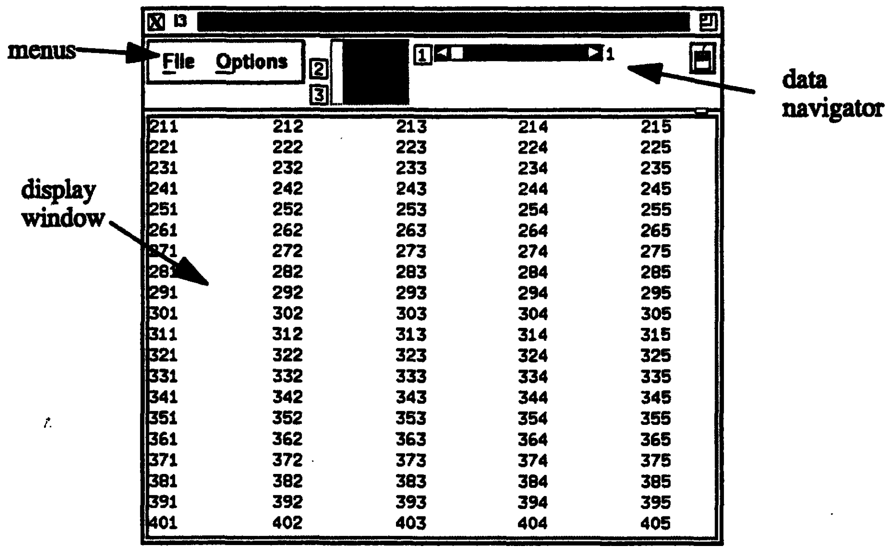
```
where (a .eq. maxval(a)) print a
```

prints the element of `a` that has the largest value. (This is equivalent to the MAXLOC intrinsic function.) See Section 2.9 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.

5.3 Working with Visualizers

The window that contains the data being printed or displayed is called a *visualizer.* Figure 19 shows a visualizer for a 3-dimensional array.



```
211    212    213    214    215
221    222    223    224    225
231    232    233    234    235
241    242    243    244    245
251    252    253    254    255
261    262    263    264    265
271    272    273    274    275
281    282    283    284    285
291    292    293    294    295
301    302    303    304    305
311    312    313    314    315
321    322    323    324    325
331    332    333    334    335
341    342    343    344    345
351    352    353    354    355
361    362    363    364    365
371    372    373    374    375
381    382    383    384    385
391    392    393    394    395
401    402    403    404    405
```

Figure 19. A 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 data navigator looks different depending 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 19 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.

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 depressed, 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.

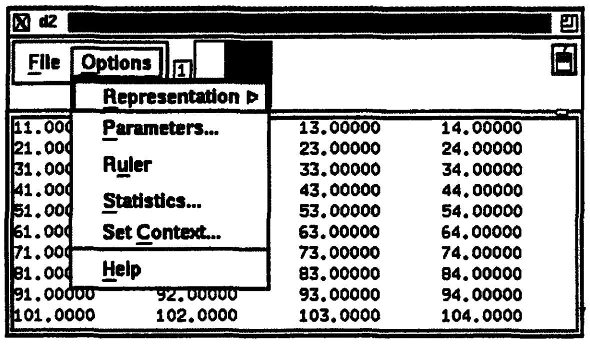
Choose **Update** from this menu to update the display window for this variable, using the value(s) at the current program location. See Section 5.3.5 for more information on updating a visualizer.

Choose **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 20.



```
11.000                    13.00000       14.00000
21.000                    23.00000       24.00000
31.000                    33.00000       34.00000
41.000                    43.00000       44.00000
51.000                    53.00000       54.00000
61.000                    63.00000       64.00000
71.000                    73.00000       74.00000
81.000                    83.00000       84.00000
91.00000      92.00000    93.00000       94.00000
101.0000      102.0000    103.0000       104.0000
```

Figure 20. The Options menu in a visualizer.

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 how 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 **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 21 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.

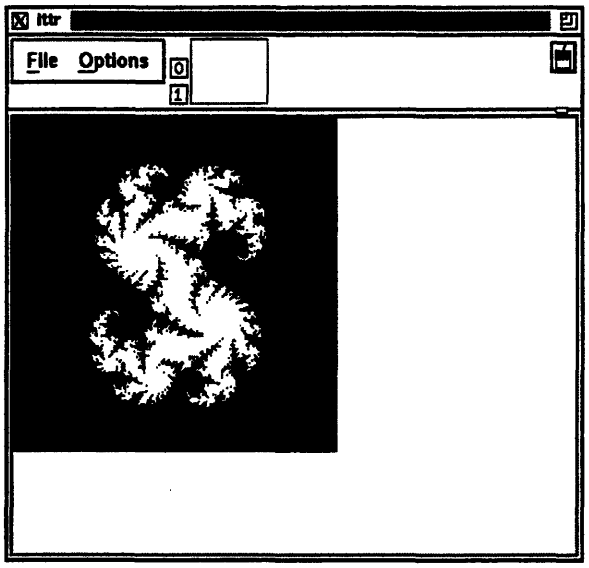


Figure 21. A dither visualizer.

- 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 22 shows a threshold representation of a 3-dimensional array.

  For complex numbers, Prism uses the modulus.

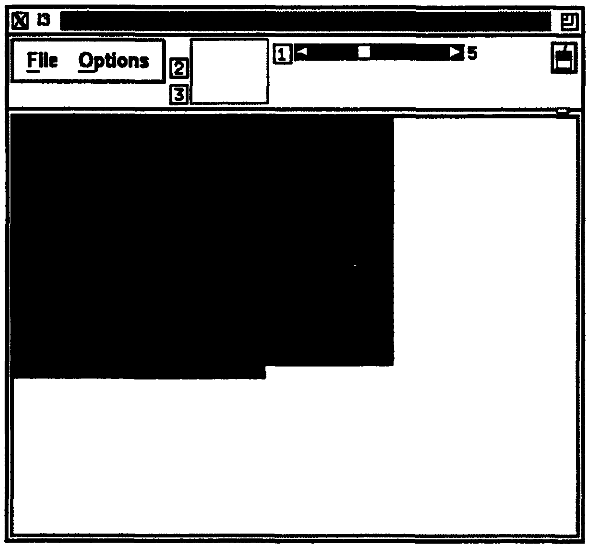


Figure 22. A threshold visualizer.

- 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 23 shows a graph visualizer for a 1-dimensional array.

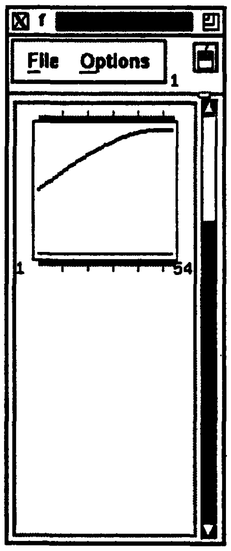


Figure 23. A 1-dimensional graph visualizer.

- 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 24 is an example.

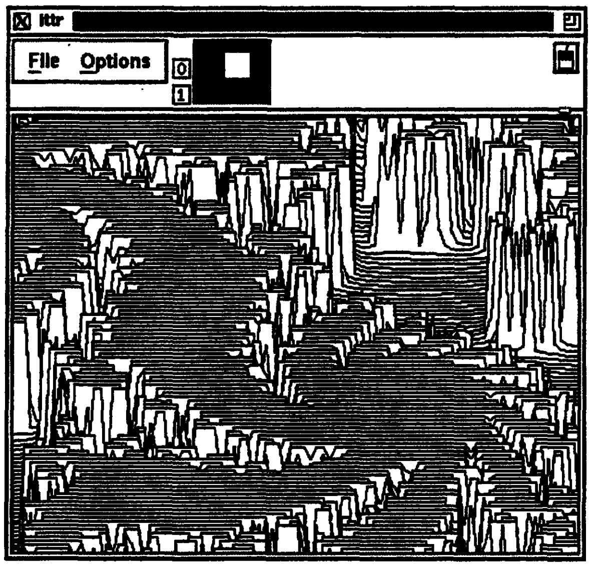


Figure 24. A surface visualizer.

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 2.9.2). 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 25 shows a vector visualizer.

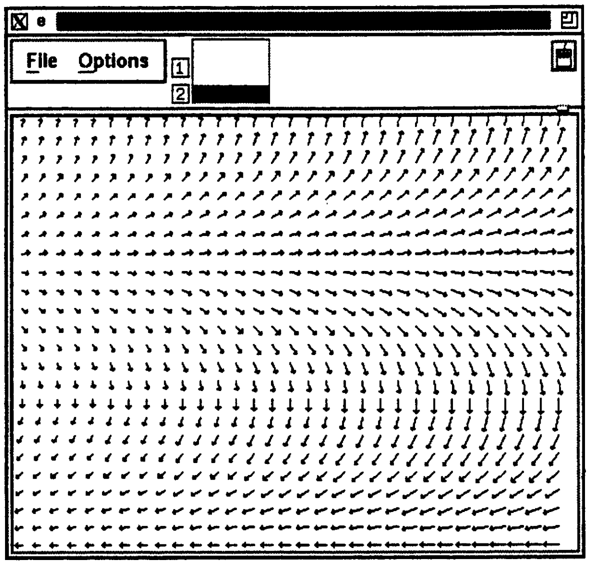
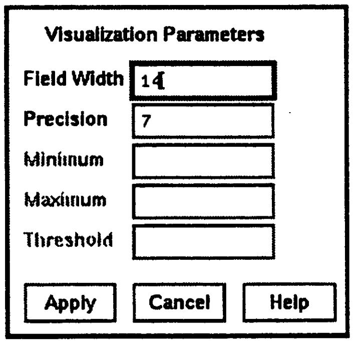


Figure 25. A vector visualizer.

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 26. If a parameters is grayed out or missing, it does not apply to the current representation.



Field Width: 14
Precision: 7
Minimum:
Maximum:
Threshold:

Figure 26. The Visualization Parameters dialog box.

The parameters 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 `$d_precision` argument (for doubles) or `$f_precision` argument (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.

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 27 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.

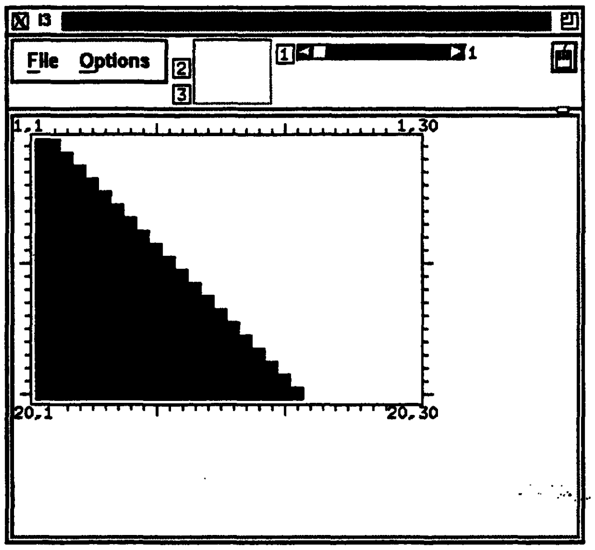


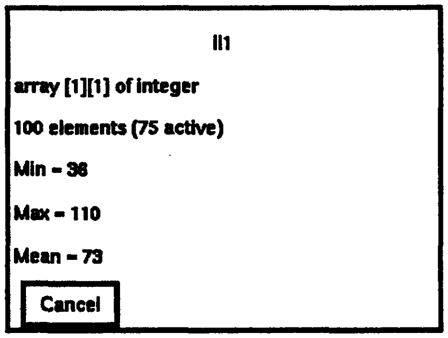
Figure 27. A threshold visualizer with a ruler.

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 28 gives an example of the **Statistics** window.



```
ll1
array [1][1] of integer
100 elements (75 active)
Min = 36
Max = 110
Mean = 73
```

Figure 28. Statistics for a visualizer.

For complex numbers, Prism uses the modulus.

Setting the Context

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 29 shows the **Set Context** dialog box.

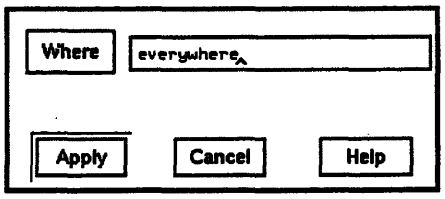


Figure 29. The Set Context dialog box.

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 you can click on the **Where** button to display a menu. The choices in the menu are **everywhere** and **other.**

- Choosing **everywhere,** as mentioned above, makes 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 2.9 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.

5.3.5 Updating and Closing the Visualizer

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

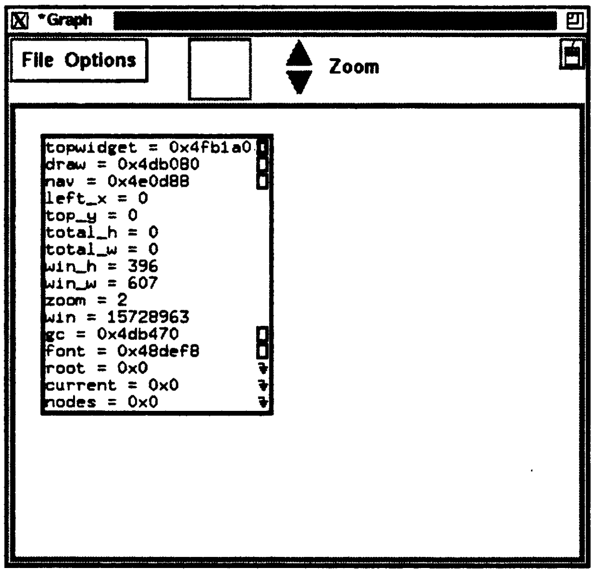
If you created the visualizer by issuing a **Print** command, its display window is grayed out when the program resumes execution and the values in the window are outdated. To update the values, choose **Update** from the visualizer's **File** menu.

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

5.4 Visualizing Structures

If you print a pointer or a structure (or a structure-valued expression) in a window, a *structure visualizer* appears. One exception: C* parallel structures are displayed in a regular visualizer, because they can't contain pointers.

Figure 30 shows an example of a structure visualizer.



```
topwidget = 0x4fb1a0
draw = 0x4db080
nav = 0x4e0d88
left_x = 0
top_y = 0
total_h = 0
total_w = 0
win_h = 396
win_w = 607
zoom = 2
win = 15728963
gc = 0x4db470
font = 0x48def8
root = 0x0
current = 0x0
nodes = 0x0
```

Figure 30. A 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.

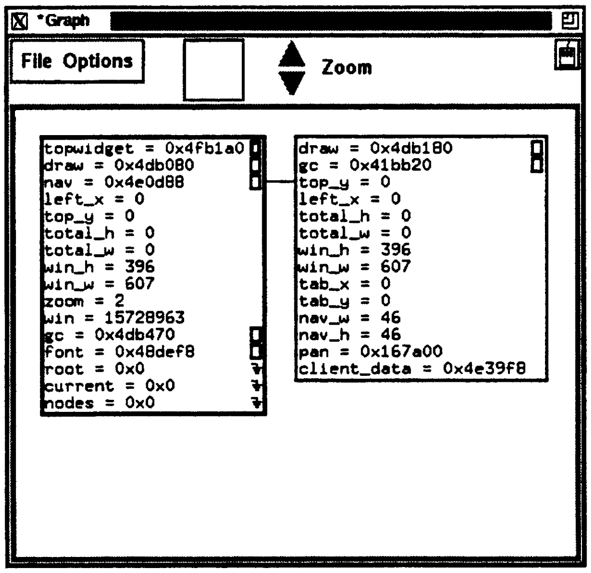
You can perform various actions within a structure visualizer, as described below.

5.4.1 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 30 changes the visualizer as shown in Figure 31.
- **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.



```
topwidget = 0x4fb1a0        draw = 0x4db180
draw = 0x4db080             gc = 0x41bb20
nav = 0x4e0d88              top_y = 0
left_x = 0                  left_x = 0
top_y = 0                   total_h = 0
total_h = 0                 total_w = 0
total_w = 0                 win_h = 396
win_h = 396                 win_w = 607
win_w = 607                 tab_x = 0
zoom = 2                    tab_y = 0
win = 15728963              nav_w = 46
gc = 0x4db470               nav_h = 46
font = 0x48def8             pan = 0x167a00
root = 0x0                  client_data = 0x4e39f8
current = 0x0
nodes = 0x0
```

Figure 31. A 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.

5.4.2 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 32 shows part of a complicated structure visualizer in which we have zoomed out.

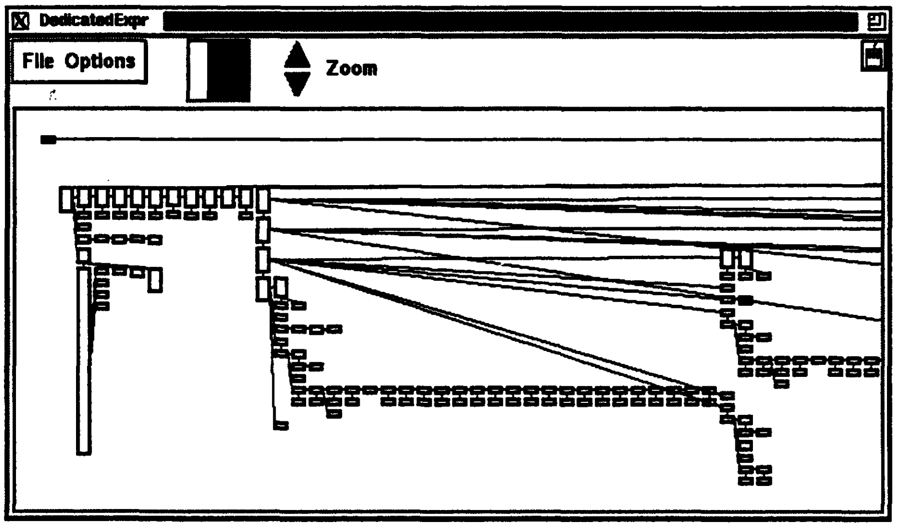


Figure 32. Zooming out in a structure visualizer.

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

5.4.3 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).

5.4.4 More about Pointers in Structures

Note the following about pointers in structure visualizers:

- Null pointers — for example, `root` in Figure 31 — 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.

5.4.5 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.

5.4.6 Visualizing Dynamic Arrays and Union Members

You can include special functions in your program to describe data structures to Prism in more detail than the compiler alone can provide. This allows Prism's structure visualizer to display the additional information. Specifically, you can tell Prism:

- which union member of a structure is currently valid
- which pointers in a structure are really dynamically sized arrays

Note these points about the functions discussed in this section:

- Providing these functions is optional. Prism's structure visualizer still expands structure nodes without them, but the dynamic arrays look like pointers, and every union member is printed instead of only the currently valid member.
- You never call these functions directly; rather, Prism calls them as necessary.

Calling the Functions

You can provide a special function for each C structure type whose definition you wish to augment. The function has this definition:

```
int prism_define_name (pointer)
struct name *pointer;
```

where *name* is the type name of the structure.

When visiting a node of a structure, the structure visualizer first checks for the existence of such a function. If the function does not exist, the node is expanded normally. If the function does exist, it is called with one argument: a pointer to the specific instance of the structure type being expanded.

Your **prism_define_*name*** function must call several auxiliary functions, defined in the Prism run-time library, to augment the structure's definition. These functions are:

- `void prism_struct_init()`

  Call this at the top of the function.

- `int prism_struct_return()`

The function should end with

```
return prism_struct_return();
```

- `void prism_add_array (char *fieldname,`
  `int nelements)`

  Call this for each pointer field in the structure that is really a dynamic array. **`fieldname`** is the name of the pointer field of the structure, and **`nelements`** is the current length of the array pointed to by **`fieldname`**.

- `void prism_add_union (char *union_name,`
  `char *union_element)`

  Call this for each field of a structure that is of type **`union`**. **`union_name`** is the name of the member that is a union. **`union_element`** is the name of the field in the union that is currently valid.

An Example

This sample program shows the use of these functions.

```
struct Graph {
  int nlines;      /* length of lines array */
  float *lines;
  int nvertices;   /* length of vertices array */
  float *vertices;
};

struct Value {
  int type;        /* gives type of currently valid
                      union member */
  union {
    int ival;
    float fval;
    double dval;
  } val;
};

#define INTEGER 0
#define FLOAT 1
#define DOUBLE 2

prism_define_Graph(g)
struct Graph *g;
```

```
{
  prism_struct_init();

  prism_add_array( "lines", g->nlines);
  prism_add_array( "vertices", g->nvertices);

  return prism_struct_return();
}

prism_define_Value(v)
struct Value *v;
{
  prism_struct_init();

  if (v->type == INTEGER)
    prism_add_union( "val", "ival");
  else if (v->type == FLOAT)
    prism_add_union( "val", "fval");
  else if (v->type == DOUBLE)
    prism_add_union( "val", "dval");

  return prism_struct_return();
}
```

The functions **prism_define_Graph** and **prism_define_Value** provide additional information about the structure types **Graph** and **Value**, respectively.

5.5 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 popup 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.

5.5.1 What Is Displayed

Prism displays the information about the variable in the command window. If a CM Fortran array is CM-resident, that information is included. For example:

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

5.6 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, a Fortran array section, or a C* left-indexed parallel variable.

If the left-hand side is a CM Fortran array or array section, the right-hand side must be a conformable expression. For example, if **array1** and **array2** are conformable,

```
assign array1 = array2 + 2
```

adds 2 to each element of **array2** and assigns the result to **array1**.

Similarly, if the left-hand side is a C* parallel variable, the right-hand side must be of the same shape. For example, if **p1** and **p2** are parallel variables of the same shape,

```
assign p1 = p2
```

assigns the value of each element of **p2** to the corresponding element of **p1**.

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

5.7 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
```

Chapter 6

Obtaining Performance Data

Prism lets you collect performance data on your C* or CM Fortran program. Collecting and analyzing performance data can help you uncover and correct bottlenecks that slow down a program.

Section 6.1 is an overview of obtaining performance data in Prism. To learn:

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

6.1 Overview

Prism helps you determine where your C* or CM Fortran 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 send/get 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.

Prism aggregates performance data separately for the front end (or partition manager) and for the CM (or the nodes); these are referred to as the *serial subsystem* and *parallel subsystem*, respectively. This is necessary because both subsystems contribute independently to a program's execution time.

In addition to displaying the data, Prism provides a *performance advisor* that gives an interpretation of the data. See Section 6.6.2 for more information on it.

6.2 Writing and Compiling Your Program

Performance data is available for C* and CM Fortran programs. To collect performance data, you must compile your program with the `-cmprofile` option. Don't use the `-O0` option to turn off optimization for a CM-2/200 C* program.

If your program calls an individual routine not compiled with the `-cmprofile` option (such as a routine from a CM library like CMSSL), serial-subsystem data will be available for that routine, as well as summary data on use of the parallel subsystem. Specific information on parallel-subsystem resources will not be available. A routine not compiled with `-cmprofile` cannot call a routine compiled with `-cmprofile`.

Performance data is available if you compile your CM Fortran or CM-5 C* program with the `-cmsim` option and run it on a Sun-4. The timings will effectively be those for a one-node CM-5.

6.2.1 Including Timers within Your Program

Prism collects performance data using the CM timing utility; see the *CM Fortran User's Guide* or *C* User's Guide* for a description of this utility. By default, Prism uses timers 5-63, leaving timers 0-4 available for use within the program itself. If you want to use more than five timers, use the environment variable **CMPROF_N_USER_TIMERS** to specify the number. For example, if you want to use 10 timers, set the variable as follows (for the C shell):

```
% setenv CMPROF_N_USER_TIMERS 10
```

This reserves timers 0-9 for use within your program. Program execution becomes less efficient, and the performance data becomes more distorted, as you use more timers, leaving fewer available for Prism.

NOTE: If you fail to set this environment variable, and thereby try to use timers that Prism itself is using, the resulting performance data will be incorrect and possibly bizarre (for example, with values well in excess of 100 percent). You receive a warning from Prism if you try to use timers that Prism is using.

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 slightly. The exact degree to which this occurs is program-dependent.
- Interrupting performance (for example, by stopping at a breakpoint and printing values) distorts performance data.

Note these points with regard to serial-subsystem data:

- Running on a heavily loaded front end or partition manager may inflate somewhat the time allocated to its CPU system time. For most accurate results, run your program on a front end or partition manager that is not heavily loaded.
- Paging on the front end or partition manager may cause some discrepancies in the data for the serial subsystem; these discrepancies will be greater on smaller programs.

Note these points with regard to parallel-subsystem data:

- To account for the effect of timesharing on the CM or nodes, Prism normalizes the elapsed time for all resources except I/O, so that it approximates the time the program would have taken to execute on a dedicated system. Since Prism accounts for the effects of timesharing on the parallel subsystem, parallel performance data for all resources except I/O should be consistent between runs of a program.

- Prism does *not* normalize the elapsed time for I/O. For most accurate I/O data, therefore, run your program on a CM (including the front end or partition manager) that is not heavily loaded. If the CM is heavily loaded, the percentage assigned to I/O will be inflated relative to the percentages assigned to the other resources.

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.

On a CM-2 or CM-200, Prism automatically turns safety off if you run a program with collection turned on; it turns safety back on if you subsequently run the program with collection turned off.

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 CM is only accepting batch jobs).

To turn on collection of performance data, set the environment variable **CMPROFILING** to **t**:

```
% setenv CMPROFILING t
```

To turn collection off, set the environment variable to **f**.

To specify the program on which data is to be collected, set the environment variable **CMPROFILING_EXECUTABLE_FILENAME** to the name of the executable program. For example:

```
% setenv CMPROFILING_EXECUTABLE_FILENAME a.out
```

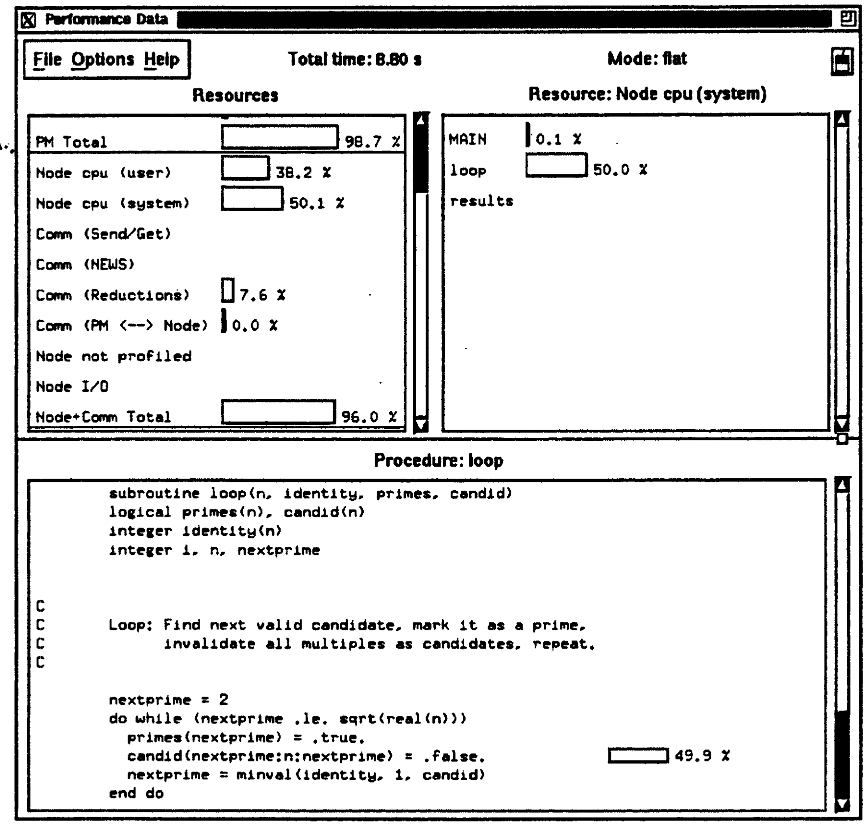
To specify the file to which the performance data is to be sent, set the environment variable **CMPROFILING_DATA_FILENAME** to the name of the file. For example:

```
% setenv CMPROFILING_DATA_FILENAME 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 33 shows an example.



Total time: 8.80 s

Mode: flat

Resources

```
PM Total                 98.7 %
Node cpu (user)          38.2 %
Node cpu (system)        50.1 %
Comm (Send/Get)
Comm (NEWS)
Comm (Reductions)        7.6 %
Comm (PM <--> Node)      0.0 %
Node not profiled
Node I/O
Node+Comm Total          96.0 %
```

Resource: Node cpu (system)

```
MAIN     0.1 %
loop     50.0 %
results
```

Procedure: loop

```
        subroutine loop(n, identity, primes, candid)
        logical primes(n), candid(n)
        integer identity(n)
        integer i, n, nextprime


C
C       Loop: Find next valid candidate, mark it as a prime,
C             invalidate all multiples as candidates, repeat.
C

        nextprime = 2
        do while (nextprime .le. sqrt(real(n)))
          primes(nextprime) = .true.
          candid(nextprime:n:nextprime) = .false.          49.9 %
          nextprime = minval(identity, 1, candid)
        end do
```

Figure 33. The Performance Data window.

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 percentage or the amount of time that the 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.) The total amount of execution time and the display mode for procedures (flat or call-graph) is displayed at the top of the window.

By default, the window displays the percentage of total execution time next to each histogram bar. Choose **Units** from the **Options** menu to change this. You have these choices:

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

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, along with totals for each subsystem.

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 usages by subsystem. Choose **Time** to display the resources in order from the highest usage for a subsystem (at the top) to the lowest.

On a CM-2 or CM-200

The **Resources** pane for a CM-2 or CM-200 series Connection Machine system provides this data:

- **FE cpu (user)** — This is front-end CPU time used by the program.
- **FE cpu (system)** — This is front-end CPU time used by the operating system on behalf of the program.
- **FE I/O** — This is time spent in I/O on the front end.

- **FE Total** is the total of these resources. It represents the program's use of the front-end subsystem.
- **CM cpu (user)** — This is the time that the program spent in processing on the CM. It refers to the amount of time *any* CM processor was active.
- **CM cpu (system)** — This is CM CPU time used by the operating system on behalf of the program.
- **Comm (Send/Get)** — This is the time that the program spent in router communication (sends and gets) on the CM.
- **Comm (NEWS)** — This is the time that the program spent in NEWS communication (also referred to as *grid* communication) on the CM.
- **Comm (Reductions)** — This is the time that the program spent doing data reductions on the CM.
- **Comm (FE<–>CM)** — This is the time spent in communication between the front end and the CM processors.
- **CM I/O** — This is the time spent in I/O between the CM processors and I/O devices.
- **CM not profiled** — This is the time spent on the CM by routines that weren't compiled with the `-cmprofile` option. (These include routines in CM libraries such as CMSSL.) If the routine had been compiled with `-cmprofile`, this time would be allocated to the other CM resources. This resource is not displayed for CM-2/200 C* programs or CM Fortran programs prior to Version 2.1; for these programs, CM time in routines not compiled with `-cmprofile` is not measured.
- **CM Total** is the total of these resources. It represents the program's use of the CM subsystem.

The total use of the front-end subsystem can be less than or equal to 100 percent. Typically, it will be less than 100 percent, because there will be times when the CM is busy and the front end is not.

The total use of the CM subsystem can also be less than or equal to 100 percent. The difference between the total and 100 percent is time during which the CM is idle. To use the CM efficiently, CM idle time should be kept as low as possible.

Note that the use of front-end resources and CM resources can each sum to 100 percent, because both can be busy all the time a program is executing.

On a CM-5

The **Resources** pane provides this data for a CM-5 system:

- **PM cpu (user)** — This is partition-manager CPU time used by the program.
- **PM cpu (system)** — This is partition-manager CPU time used by the operating system on behalf of the program.
- **PM I/O** — This is time spent in I/O on the partition manager.
- **PM Total** is the total of these resources. It represents the program's use of the partition-manager subsystem.
- **Node cpu (user)** — This is the time that the program spent in processing on the nodes. It refers to the amount of time *any* nodes were active.
- **Node cpu (system)** — This is node CPU time used by the operating system on behalf of the program.
- **Comm (Send/Get)** — This is the time that the program spent in router communication (sends and gets) on the CM.
- **Comm (NEWS)** — This is the time that the program spent in NEWS communication (also referred to as *grid* communication) on the nodes.
- **Comm (Reductions)** — This is the time that the program spent doing data reductions on the nodes.
- **Comm (PM<-->Node)** — This is the time spent in communication between the partition manager and the nodes.
- **Node I/O** — This is the time spent in I/O between the nodes and I/O devices.
- **Node not profiled** — This is the time spent on the nodes by routines that weren't compiled with the **`-cmprofile`** option. (These include routines in CM libraries such as CMSSL.) If the routine had been compiled with **`-cmprofile`**, this time would be allocated to the other node resources. This resource is available only as of CM Fortran 2.1 and CM-5 C* Version 7.1; for programs using earlier versions of these languages, time spent in routines not compiled with **`-cmprofile`** is not measured.
- **Node+Comm Total** is the total of these resources. It represents the program's use of the node subsystem.

The total use of the partition-manager subsystem can be less than or equal to 100 percent. Typically, it will be less than 100 percent, because there will be times when the nodes are busy and the partition manager is not.

The total use of the node subsystem can also be less than or equal to 100 percent. The difference between the total and 100 percent is time during which the nodes are idle. To use the nodes efficiently, node idle time should be kept as low as possible.

Note that the use of partition-manager resources and node resources can each sum to 100 percent, because both can be busy all the time a program is executing.

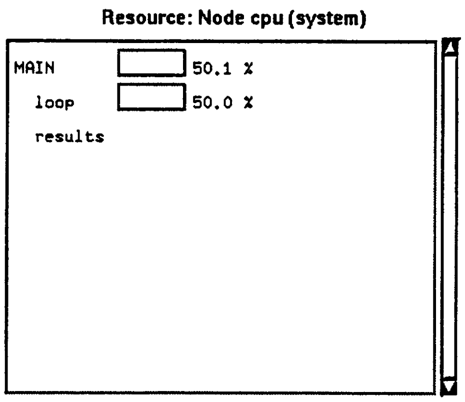
6.5.2 The Procedures Pane

The pane titled **Resources:** *name* in the **Performance Data** window displays histograms showing the utilization of a specific resource or subsystem by each procedure in a program; we call this the *Procedures* pane. You choose the resource or subsystem 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 or subsystem appears in the title of the pane — for example: **Resource: CM Total.**

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 or subsystem.

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 or subsystem. This is useful for determining which procedures are consuming most of the time for the resource or subsystem. The **Procedures** pane in Figure 33 shows the data in flat mode.



Resource: Node cpu (system)

```
MAIN      [    ] 50.1 %
  loop    [    ] 50.0 %
  results
```

Figure 34. A call-graph display.

In call-graph mode, you see which procedures call which other procedures, and the use of the selected resource or subsystem for each individual call. This gives a more detailed picture of the program's behavior. Figure 34 shows the call-graph display for the data shown in the **Procedures** pane in Figure 33. Note in Figure 34 that the time allocated to the **MAIN** routine includes the time spent in **loop**, which it calls.

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 or subsystem, from most to least.
- Choose **Name** to arrange the procedures in alphabetical order.

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 `-cmprofile` option, Prism will display data only for serial-subsystem resources or for the **not profiled** parallel-subsystem resource (if available); as mentioned above, all parallel-subsystem time for the routine is included in the **not profiled** resource.

6.5.3 The Source-Lines Pane

The pane titled **Procedure:** *name* displays performance data associated with each source line in a procedure; we call this 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 or subsystem for which the data is shown is the one displayed in the **Procedures** pane.

For slicewise and CM-5 CM Fortran 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 `-cmprofile`. This produces unoptimized code, however, and overall performance will be much worse.

Also note these points:

- Source-line data is not available for the serial subsystem, or for serial-subsystem resources.
- If a routine is not compiled with the `-cmprofile` 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.

6.6 Interpreting the Data

This section discusses how to make sense of the performance data that Prism provides.

6.6.1 Making Sense of the Times

Recall that the totals for the serial subsystem (front end or partition manager) and the parallel subsystem (CM or nodes) are separate; each can be as high as 100%. The time for one subsystem can be less than 100% if it is idle while the other subsystem is working. The times for both subsystems can be less than 100% if each is idle at times when the other is busy. The ideal is for the parallel subsystem to be busy as close to 100% of the time as possible.

Recall also, as described in Section 6.3, that the load on either the serial subsystem or the parallel subsystem can affect your results. For most accurate results, run on lightly loaded systems.

6.6.2 Isolating Bottlenecks

Prism's performance data gives you a picture of how your program uses system resources. We assume 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. This section describes how to use the performance data to find your program's bottlenecks.

To help you in this analysis, Prism provides a *performance advisor*, which summarizes and analyzes the performance data that Prism has collected. To display this information, choose **Advice** from the **Performance** menu, or issue the command `perfadvice`. You can use this performance advisor, or you can analyze the data on your own, to isolate the bottlenecks in your program. The performance advisor provides answers to the questions discussed below; we believe that following this procedure provides the best method for interpreting the performance data.

We suggest asking these questions to isolate the bottlenecks in your program:

1. **Which of the two subsystems that Prism measures does the program use more heavily?** For example, if total serial-subsystem time is greater than total parallel-subsystem time, then reducing the use of the serial subsystem is likely to provide the greatest performance gains. Reducing the use of the parallel subsystem *may* improve performance, but you may also find that it will have no effect on performance, since the use takes place at the same time that the serial subsystem is also in operation.

2. **Which resource within this subsystem has the highest usage?** If your program uses the parallel subsystem more heavily than the serial subsystem, and Send/Get communication is the most-used parallel-subsystem resource, then you will obtain the greatest performance gains by reducing the use of this resource.

3. **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.

4. **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.

Here is an example of a report from the performance advisor (for a CM-2 or CM-200). Note that it does not include source-line information, because the most-used resource is **FE cpu (user)**, for which source-line data is unavailable.

```
Of the two subsystems that Prism currently measures,
your program makes greater use of the front end.

Within that subsystem, your program makes the greatest
use of the "FE cpu (user)" resource, having kept it busy
71% of the total elapsed time.

Procedure complexarray, called from floatarray, called
from MAIN, makes the greatest use of the "FE cpu (user)"
resource, having kept it busy 69% of the elapsed time.
```

Note that 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 without having to use the performance advisor.

6.6.3 Anomalous Performance Data

It is possible that your performance data will simply appear incorrect. For example, you may get percentages in the thousands or millions.

This problem can occur if you try to use timers that Prism has allocated for its own use. If you use more than five timers in your program, be sure to set the environment variable `CMPROF_N_USER_TIMERS` to the number that you use; see Section 6.2.1.

6.7 Saving and Loading 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.

Follow this procedure:

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 pathname, the filename 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.

3. 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. To learn:

- **How to edit source code,** see Section 7.1, below.
- **How to use the UNIX make utility from within Prism to compile and link source code,** see 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: Choose the **Edit** selection from the **Utilities** menu.

From the command window: Issue the command `edit` from the command line.

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 filename 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 UNIX tool `make`. The `make` utility lets you automatically recompile and relink a program that is broken up into different source files. See your UNIX 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.

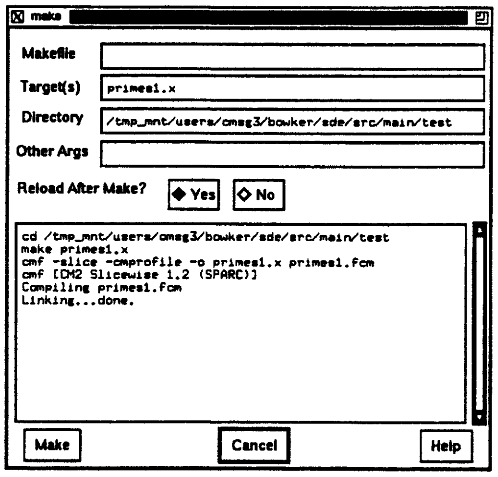
7.2.2 Using the Makefile

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

Prism uses the standard UNIX `make` utility, `/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: Choose **Make** from the **Utilities** menu. A window appears; Figure 35 is an example.



Makefile:
Target(s): primes1.x
Directory: /tmp_mnt/users/cmsg3/bowker/sde/src/main/test
Other Args:
Reload After Make? [✓ Yes] No

```
cd /tmp_mnt/users/cmsg3/bowker/sde/src/main/test
make primes1.x
cmf -slice -cmprofile -o primes1.x primes1.fcm
cmf [CM2 Slicewise 1.2 (SPARC)]
Compiling primes1.fcm
Linking...done.
```

Figure 35. 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 UNIX 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 window 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 UNIX version of **make**.

Chapter 8

Getting Help

This chapter describes how to obtain information about Prism and Connection Machine commands, languages, and libraries. To learn:

- **How to obtain help about Prism,** see Section 8.1.
- **How to obtain CM on-line documentation,** see Section 8.2.
- **How to send electronic mail about Prism,** see Section 8.3.
- **How to join the Prism mailing list,** see Section 8.4.

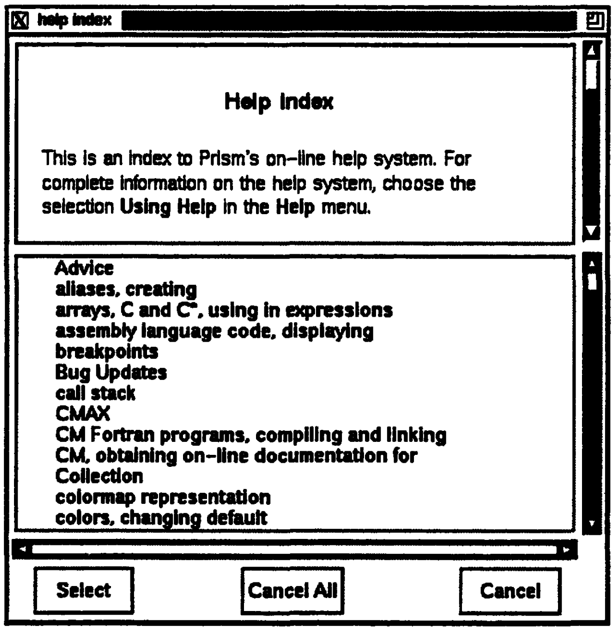
8.1 Getting Help

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.
- The **Help** selection in menus and the **Help** button in windows and dialog boxes provide instructions for using these screen areas.
- Command-line help provides information about commands you can issue from the command window.

8.1.1 Using the Help Index

The *Help Index* is a list of entries about which you can obtain information; see Figure 36.



Help Index

This is an index to Prism's on-line help system. For complete information on the help system, choose the selection **Using Help** in the **Help** menu.

Advice
aliases, creating
arrays, C and C*, using in expressions
assembly language code, displaying
breakpoints
Bug Updates
call stack
CMAX
CM Fortran programs, compiling and linking
CM, obtaining on-line documentation for
Collection
colormap representation
colors, changing default

Figure 36. The Help Index.

Displaying the Help Index

From the menu bar, choose the **Index** selection from the **Help** menu.

Choosing an Entry from the Help Index

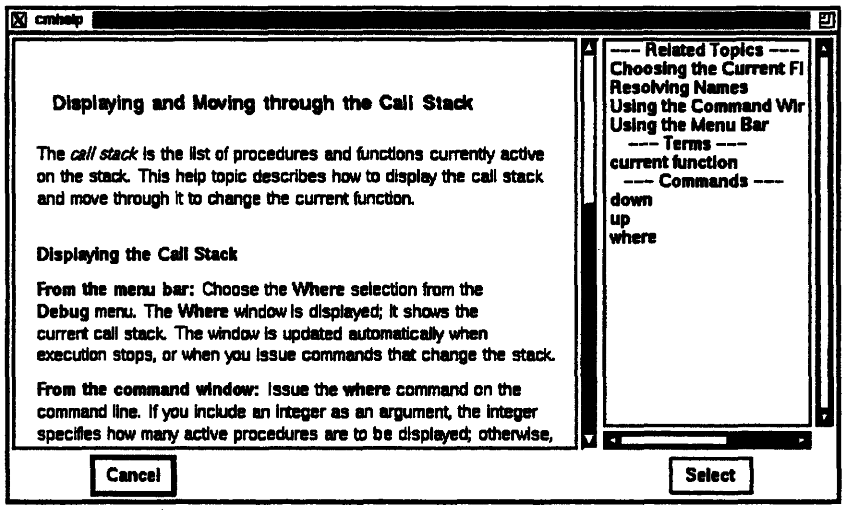
To choose an entry:

With a mouse: Left-click on the entry so that it is highlighted. Use the scroll bar to the right of the list to scroll through the complete list of entries; then click on **Select**. Or simply double-click on the entry.

From the keyboard: Use the up and down arrow keys to move through the entries. Each entry is highlighted as you reach it. Press **Return** to display information about a highlighted entry.

When a Topic Window Is Displayed

Choosing an entry causes Prism to display a window containing the topic in which the entry is discussed. See Figure 37 for an example. If there is more text than will fit in the window, click on the up or down arrow in the scroll bar to move the text up or down a line. Or drag the elevator in the scroll bar up or down to move to the corresponding point in the text. From the keyboard, you can use the up and down arrow keys to move through the text; press the **Control** key along with the arrow key to scroll one-half page of text.



Displaying and Moving through the Call Stack

The *call stack* is the list of procedures and functions currently active on the stack. This help topic describes how to display the call stack and move through it to change the current function.

Displaying the Call Stack

From the menu bar: Choose the **Where** selection from the **Debug** menu. The **Where** window is displayed; it shows the current call stack. The window is updated automatically when execution stops, or when you issue commands that change the stack.

From the command window: Issue the **where** command on the command line. If you include an integer as an argument, the integer specifies how many active procedures are to be displayed; otherwise,

---- Related Topics ----
Choosing the Current Fi
Resolving Names
Using the Command Wir
Using the Menu Bar
---- Terms ----
current function
---- Commands ----
down
up
where

Figure 37. A help topic.

Getting Help on Related Topics, Subtopics, Terms, and Commands

To the right of the topic discussion there may be lists of related topics, subtopics, terms, and commands. Choose an item in these lists in the same way you chose a topic from the main Help Index. Another window is displayed; the original window remains on the screen.

- *Related topics* are topics that may be of interest to the reader of the current topic.
- *Subtopics* provide information about self-contained subjects within a topic area. They too can contain lists of related topics, terms, and commands.
- *Terms* are specialized words or phrases used in the topic discussion. If you are unfamiliar with a term, click on its entry and a brief definition appears in a separate window. If you want further information, click on **Help** in the definition window; another window is displayed showing the topic in which the term is discussed.
- *Commands* provide reference descriptions of Prism commands.

Cancelling Topic Windows

To cancel an individual topic window, click on the **Cancel** button in the window or press the **Esc** key.

To cancel all open topic windows, as well as the Help Index window itself, click on the **Cancel All** button in the Help Index window.

8.1.2 Choosing Other Selections from the Help Menu

In addition to the Help Index, you have several other sources of information you can choose from the **Help** menu:

- Choose **Using Help** to display an overview of the Help system.
- Choose **Release Notes** to display release notes for the current version of Prism. (You should be sure to read the release notes before using Prism, to find out about last-minute information that doesn't get into the documentation.)
- 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 information about it.
- Choose **Commands Reference** to display a list of Prism commands. You can double-click on a command to obtain its reference description.

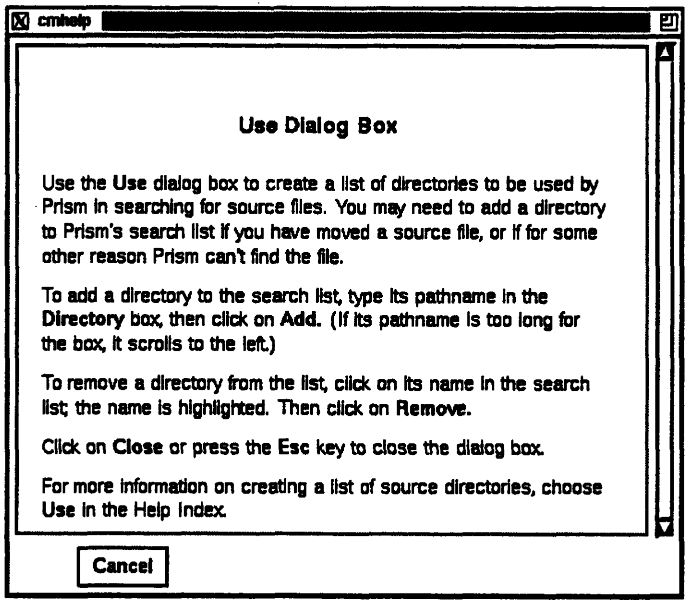
- Choose **Tutorial** to display an on-line tutorial that will guide you through loading, executing, and analyzing a sample program in Prism. We recommend taking this tutorial as a quick way of becoming acquainted with some of Prism's major features.

These choices are also available from the Help Index.

8.1.3 Using Help Selections and Help Buttons

Each menu in Prism has a **Help** selection, and each window and dialog box has a **Help** button.

You can use either the mouse or the keyboard to choose help. A Help window is displayed that gives information about the menu or dialog box; see Figure 38 for an example. It also names the Help Index entry where you can find more detailed information.



Use Dialog Box

Use the **Use** dialog box to create a list of directories to be used by Prism in searching for source files. You may need to add a directory to Prism's search list if you have moved a source file, or if for some other reason Prism can't find the file.

To add a directory to the search list, type its pathname in the **Directory** box, then click on **Add.** (If its pathname is too long for the box, it scrolls to the left.)

To remove a directory from the list, click on its name in the search list; the name is highlighted. Then click on **Remove.**

Click on **Close** or press the **Esc** key to close the dialog box.

For more information on creating a list of source directories, choose **Use** in the Help Index.

Figure 38. The Help window for the Use dialog box.

To cancel the Help window, click on the **Cancel** button or press the **Esc** key.

8.1.4 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.

8.1.5 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 On-Line Documentation

This section discusses how to obtain on-line documentation about CM commands, languages, and libraries within graphical Prism.

Prism offers interfaces to several kinds of on-line documentation:

- UNIX-style manual pages
- plain-text versions of Connection Machine manuals
- release notes
- bug-update files

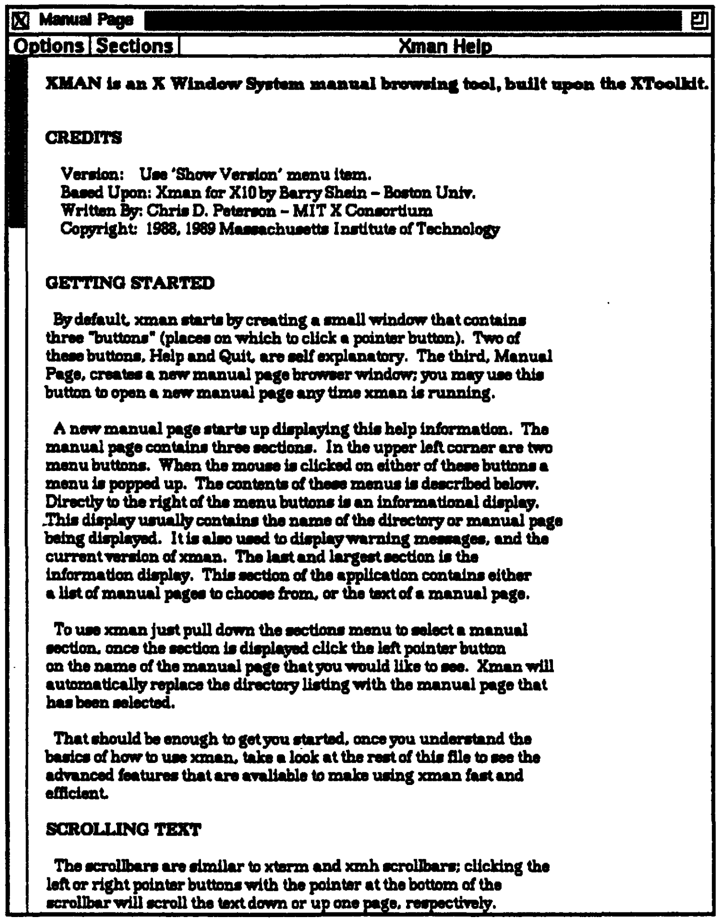
See Section B.4 for a discussion of how to obtain on-line documentation from commands-only Prism.

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 39. Basically, you can search for a manual page by choosing **Search** from the **xman Options** menu. You can use **xman** to view any UNIX manual pages available on your system, not just those related to the CM.

NOTE: If **xman** is not available on your system, you will not be able to use this feature.



XMAN is an X Window System manual browsing tool, built upon the XToolkit.

CREDITS

Version: Use 'Show Version' menu item.
Based Upon: Xman for X10 by Barry Shein – Boston Univ.
Written By: Chris D. Peterson – MIT X Consortium
Copyright: 1988, 1989 Massachusetts Institute of Technology

GETTING STARTED

By default, xman starts by creating a small window that contains three "buttons" (places on which to click a pointer button). Two of these buttons, Help and Quit, are self explanatory. The third, Manual Page, creates a new manual page browser window; you may use this button to open a new manual page any time xman is running.

A new manual page starts up displaying this help information. The manual page contains three sections. In the upper left corner are two menu buttons. When the mouse is clicked on either of these buttons a menu is popped up. The contents of these menus is described below. Directly to the right of the menu buttons is an informational display. This display usually contains the name of the directory or manual page being displayed. It is also used to display warning messages, and the current version of xman. The last and largest section is the information display. This section of the application contains either a list of manual pages to choose from, or the text of a manual page.

To use xman just pull down the sections menu to select a manual section, once the section is displayed click the left pointer button on the name of the manual page that you would like to see. Xman will automatically replace the directory listing with the manual page that has been selected.

That should be enough to get you started, once you understand the basics of how to use xman, take a look at the rest of this file to see the advanced features that are avaliable to make using xman fast and efficient.

SCROLLING TEXT

The scrollbars are similar to xterm and xmh scrollbars; clicking the left or right pointer buttons with the pointer at the bottom of the scrollbar will scroll the text down or up one page, respectively.

Figure 39. The xman window.

8.2.2 Viewing CM Documents

There are two ways of viewing sections of Connection Machine documents available on-line: from the menu bar and from the source window.

From the Menu Bar

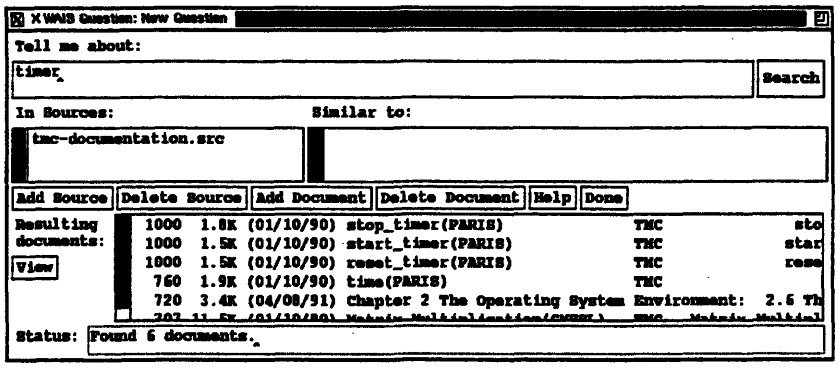
From the menu bar, choose **Online Doc** from the **Doc** menu. This displays a dialog box in which you can enter the topic on which you want information. The topic can be in the form of one or more keywords, or it can be a sentence. Clicking on **Search** in this dialog box passes the topic on to a special version of **xwais**, the X version of Thinking Machines' wide-area information server; see below.

From the Source Window

Select the text on which you want information by dragging the mouse over it. Then right-click the mouse; a popup menu is displayed. Click on **Doc Search** in this menu. Prism passes the selected text as a topic to **xwais**.

Using the xwais Utility

The **xwais** utility displays a window like the one shown in Figure 40. If you have selected text as a topic, it appears in the **Tell me about:** field.



```
Tell me about:
timer

In Sources:                     Similar to:
tmc-documentation.src

Resulting    1000  1.8K (01/10/90) stop_timer(PARIS)              TMC           sto
documents:   1000  1.5K (01/10/90) start_timer(PARIS)             TMC           star
             1000  1.5K (01/10/90) reset_timer(PARIS)             TMC           rese
              760  1.9K (01/10/90) time(PARIS)                    TMC
              720  3.4K (04/08/91) Chapter 2 The Operating System Environment:  2.6 Th

Status: Found 6 documents.
```

Figure 40. The xwais window.

xwais automatically searches the CM documentation for information on your topic. When the search is concluded, the titles of the relevant sections are listed in the **Resulting Documents** field; the sections are rated according to how relevant they are, with the most relevant receiving a score of 1000.

To view a document section, click on its title so that it is highlighted, then click on **View**. The document section appears in a separate window. Click on **Save to File** in this window to save the information to a file; click on **Done** to close the window.

Several of the fields in the **xwais** window are not relevant to a search for on-line documentation. Specifically, you do not need to use the **Similar to:** field, or the **Search, Add Source, Delete Source, Add Document,** and **Delete Document** buttons.

Click on **Done** when you are finished using **xwais.**

8.2.3 Viewing Release Notes and Bug-Update Files

Thinking Machines provides monthly bug-update files for each of its products; these files contain information about newly reported, outstanding, and fixed bugs for the product. It also provides on-line versions of the release notes for its products.

To view a bug-update file, choose **Bug Updates** from the **Doc** menu. A window will appear; the files available at your site should be listed in the **Bug Update Files** list in this window. Check with your system administrator if this list is empty, or if the file you are interested in is missing (the files may have been installed in a different location from that specified when Prism was installed).

To dispay a file, left-click on its name in the list, then click on **OK**. Or simply double-click, rapidly, on the name. The file will appear in your default editor. (Use the **Customize** utility to change this editor; see Section 9.3.2.)

The procedure for viewing release notes is the same. Choose **Release Notes** from the **Doc** menu to display a window that contains a list of Release Note files. Click on the filename, then click on **OK** to display the file in your default editor.

NOTE: Release notes on Prism itself are also available by selecting **Release Notes** in the **Help** menu, as described in Section 8.1.2.

8.3 Sending Electronic Mail about Prism

You can send electronic mail about Prism to Thinking Machines Corporation. We encourage you to write us and let us know what is wrong (or right!) with Prism, or to get an answer to a question.

From the menu bar: Choose the **Email** selection from the **Utilities** menu.

From the command window: Issue the `email` command on the command line.

In each case, Prism displays a window containing an editor. The edit buffer contains the last few Prism error messages, along with instructions on how to send the contents of the buffer — basically, you just save the buffer in the usual way before quitting the editor; the mail is sent automatically. Simply quit the editor if you decide not to send us mail. You can use the **Customize** utility to determine which editor Prism displays; see Section 9.3.2.

We put the last few error messages into the buffer as a convenience, in case you are sending us information about a possible bug. Feel free to add more information, or to delete the error messages and write us about something else entirely.

If you want to talk about Prism more informally with other users, we encourage you to join the `prism-talk@think.com` mailing list, as described below.

8.4 The Prism Mailing List

Thinking Machines maintains a mailing list, `prism-talk@think.com`, where Prism users can discuss how to get the most out of the Prism programming environment. If you have comments, questions, or insights about Prism, we urge you to join this list and share them with others.

Prism developers may monitor this list and, at their discretion, participate in discussions, but they are not obligated to do so. For this reason, bug reports should not be sent to this list. If you want to ensure a prompt, official response to your mail, use the Prism **Email** feature, or contact your Applications Engineer or Thinking Machines' Customer Support group directly.

To join this list, simply send e-mail to `prism-talk-request@think.com`.

Chapter 9

Customizing Prism

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

- **How to use the tear-off region,** see Section 9.1, below.
- **How to set up alternative names for commands and variables,** see Section 9.2.
- **How to change Prism defaults by using the Customize utility,** see Section 9.3.
- **How to change Prism defaults in your X resource database,** see Section 9.4.
- **How to initialize Prism,** see 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 them. Figure 41 shows the buttons that are there by default.

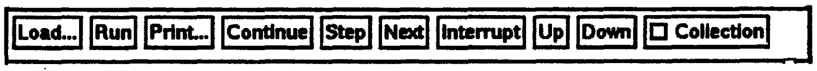


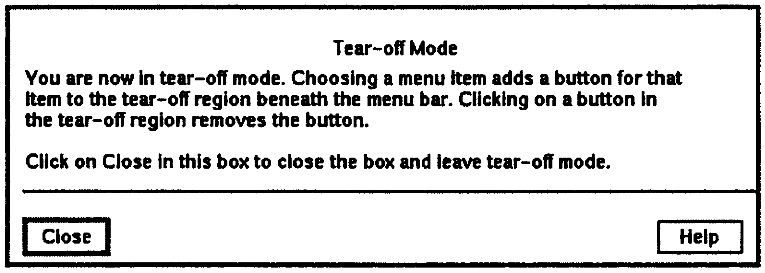
Figure 41. The tear-off region.

Putting menu selections and commands in the tear-off region lets you get access to 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 42.



Tear-off Mode

You are now in tear-off mode. Choosing a menu item adds a button for that item to the tear-off region beneath the menu bar. Clicking on a button in the tear-off region removes the button.

Click on Close in this box to close the box and leave tear-off mode.

Figure 42. The Tear-Off 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 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 Setting Up Alternative Names 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.

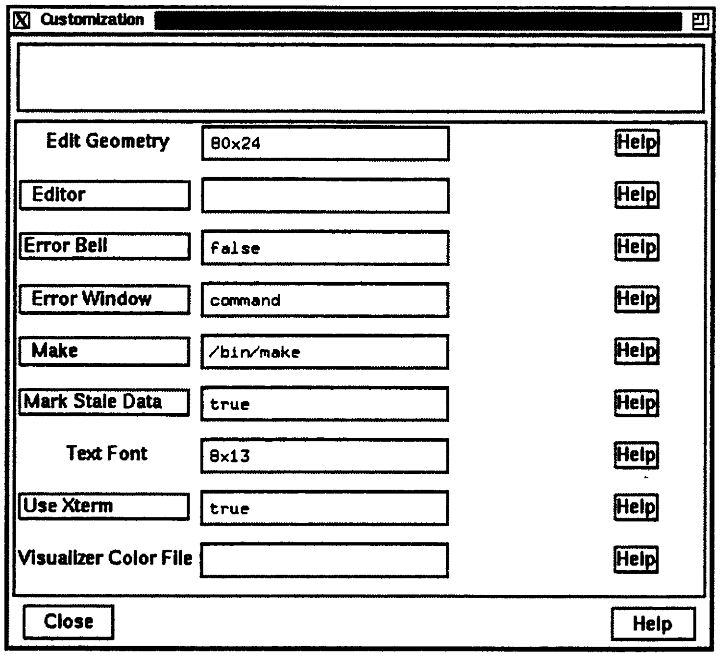
Changes you make via **alias** and **set** last for your current Prism session. To make them permanent, you can add the appropriate commands to your **.prisminit** file; see Section 9.5.

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 de' ults 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, however, much more convenient.

Choosing **Customize** from the **Utilities** menu displays the window shown in Figure 43.



Edit Geometry	80x24
Editor	
Error Bell	false
Error Window	command
Make	/bin/make
Mark Stale Data	true
Text Font	8x13
Use Xterm	true
Visualizer Color File	

Figure 43. The 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, 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** and **Use Xterm**, there are only two possible settings, **yes** and **no**; 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 The 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, or when you display a file via the **Release Notes** or **Bug Updates** selections from the **Doc** 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 **yes** tells Prism to ring the bell of your workstation. Choose **no** (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 UNIX `make` utility, `/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.

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 UNIX 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 yes (the Prism default) to tell Prism to create an xterm in which to display the I/O. Specify **no** 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. To get a list of some of the colors available on your system, you can consult the file **`/usr/lib/X11/rgb.txt`**.

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 up Prism.

9.4 Changing Prism Defaults in Your X Resource Database

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 2 lists the Prism resources.

Table 2. Prism resources.

Resource	Use
Prism.dialogColor	Specify the color for dialog boxes.
Prism.editGeometry	Specify the size and placement of editor window.
Prism.editor	Specify the editor to use.
Prism.errorBell	Specify whether the error bell is to ring.
Prism.errorwin	Specify the window to use for error messages.
Prism*fontList	Specify the font for labels, menu selections, etc.
Prism.helpColor	Specify the color for help windows.
Prism.mainColor	Specify the main background color for Prism.
Prism.make	Specify the `make` utility to use.
Prism.markStaleData	Specify how Prism is to mark stale data in visualizers.
Prism.textBgColor	Specify the background color for widgets containing text.
Prism.textFont	Specify the text font to use for certain labels.
Prism.textManyFieldTranslations	Specify the keyboard translations for dialog boxes that contain several text fields.
Prism.textMasterColor	Specify the color used to highlight the master pane in a split source window.
Prism.textOneFieldTranslations	Specify the keyboard translations for dialog boxes that contain one text field.
Prism.useXterm	Specify whether to use a new xterm for I/O.
Prism.vizColormap	Specify the colors to be used in colormap visualizers.
Prism*XmText.fontList	Specify 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.

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 UNIX **`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. To get a list of some available colors, you can consult the file **`/usr/lib/X11/rgb.txt`**.

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.

Changing Prism's Standard Colors

Use the **Prism.helpColor** resource to change the background color of help windows.

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, help windows, 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.helpColor:  bisque3
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.

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.

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 \n
  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 How 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 UNIX `make` utility, `/bin/make`. Use the resource **Prism.make** to specify the pathname 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.

9.5 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.

Appendix A

Prism Commands

This appendix lists all Prism commands. For complete reference descriptions of these commands, see the *Prism Reference Manual*, or choose the **Commands Reference** selection from the **Help** menu.

Table 3. Prism commands.

Command	Use
/string	Searches forward in the current file for *string*.
?string	Searches backward in the current file for *string*.
address/	Prints the contents of a location in memory.
value=base	Converts a value to a different base.
`alias`	Defines an alias.
`assign`	Assigns the value of an expression to a variable or array.
`attach`	Attaches to a running process.
`call`	Calls a procedure or function.
`catch`	Tells Prism to catch the specified signal.
`cd`	Changes the current working directory.
`cmattach*`	Attaches to a CM resource.
`cmcoldboot*`	Cold boots a CM resource.
`cmdetach*`	Detaches from a CM resource.
`cmfinger*`	Displays information about CM users.
`cmsetsafety*`	Sets safety on or off for a CM resource.
`collection`	Turns collection of performance data on or off.
`cont`	Continues execution.
`core`	Associates a core file with an executable program.
`delete`	Removes an event.

*Available from CM-2 and CM-200 front ends only.

Table 3. Prism commands (cont'd).

Command	Use
`detach`	Detaches from a running process.
`display`	Displays the value of an expression.
`doc`	Displays on-line documentation in commands-only Prism.
`down`	Moves the symbol-lookup context down one level.
`dump`	Prints the names and values of variables.
`edit`	Calls up an editor.
`email`	Sends mail about Prism.
`file`	Sets the source file to the specified filename.
`func`	Sets the current function to the specified function name.
`help`	Lists currently implemented commands.
`hide**`	Hides a pane of a split source window.
`ignore`	Tells Prism to ignore the specified signal.
`list`	Lists source lines.
`load`	Loads a program.
`log`	Creates a log file of your commands and Prism's responses.
`make`	Executes the `make` utility.
`next`	Executes one or more source lines, stepping over functions.
`nexti`	Executes one or more instructions, stepping over functions.
`perf`	Displays performance data.
`perfadvice`	Displays an analysis of performance data.
`perfload`	Loads a performance data file.
`perfsave`	Saves performance data to a file.
`print`	Prints the value of an expression.
`printenv`	Displays currently set environment variables.
`pushbutton**`	Adds a Prism command to the tear-off region.
`pwd`	Prints the current working directory.
`quit`	Leaves Prism.
`reload`	Reloads the currently loaded program.
`return`	Steps out to the caller of the current routine.
`run`	Starts execution of a program.
`select`	Specifies the master pane of a split source window.
`set`	Defines an abbreviation for a variable or expression.
`setenv`	Displays or sets environment variables.
`show**`	Splits the source window.
`show events`	Prints the event list.
`source`	Reads commands from a file.
`status`	Prints the event list.
`step`	Executes one or more source lines.
`stepi`	Executes one or more instructions.

**Not available in commands-only Prism.

Table 3. Prism commands (cont'd).

Command	Use
`stepout`	Steps out to the caller of the current routine.
`stop`	Sets a breakpoint.
`stopi`	Sets a breakpoint at an instruction.
`tearoff**`	Adds a menu selection to the tear-off region.
`trace`	Traces program execution.
`tracei`	Traces instructions.
`type`	Provides type information on Paris parallel variables.
`unalias`	Removes an alias.
`unset`	Removes an abbreviation created by `set`.
`unsetenv`	Removes the setting of an environment variable.
`untearoff**`	Removes a button from the tear-off region.
`up`	Moves the symbol-lookup context up one level.
`use`	Adds a directory to the list to be searched for source files.
`whatis`	Prints the type of a variable.
`when`	Sets a breakpoint.
`where`	Prints a stack trace.
`whereis`	Prints the list of all fully qualified names for an identifier.
`which`	Prints the fully qualified name Prism chooses for an identifier.

**Not available in commands-only Prism.

Appendix B

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). This appendix provides an overview of commands-only Prism. 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.*

B.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 for more information on command-line options.

When you have issued the command, you receive this prompt:

```
(prism)
```

You can issue most Prism commands at this prompt, except for commands that apply specifically to the graphical interface; these include **`pushbutton`**, **`tear-off`**, and **`untearoff`**.

B.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` option, 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.

B.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 **`collection`** command with no arguments to print out the current status of collection (on or off).

Use the **`show events`** command to print the events list. Use the **`delete`** command to delete events from this list.

Use the **`perf`** command to display performance data. Use the **`perfsave`** command to save the performance data in a format that you can later load into the graphical version of Prism (via the **`perfload`** command or the **Load Data** selection from the **File** menu in the **Performance Data** window). Use the **`perfadvice`** command to display an analysis of the performance data.

B.4 Obtaining On-Line Documentation

Use the **`doc`** command to obtain on-line documentation in commands-only Prism. The **`doc`** command is not available in graphical Prism.

Issuing **`doc`** displays a menu of available documents. Choose the number associated with the document you want to view. In most cases, this displays another menu of the chapters within the document. Choose the number associated with the chapter, and the first screenful of text for that chapter is displayed. Answer **y** in response to the **more?** prompt or simply press the **Return** key to display the next screenful. Answer **n** to return to the menu.

If you choose the number associated with bug-update files or release notes from the top-level `doc` menu, Prism displays a menu of the files or release notes available at your site.

From any menu, you can press **p** to return to the previous menu, **q** to return to the `(prism)` prompt, or **m** to display the text of a UNIX or CMOST manual page. When you press **m**, you can either enter the name of the man page immediately, or press **Return** and be prompted for the name of the man page.

Use the syntax

number **@** *filename*

to redirect the text of the document associated with *number* to the file you specify. Use

number **@@** *filename*

to add the text to the end of an existing file.

B.5 Leaving Commands-Only Prism

Issue the `quit` command to leave commands-only Prism and return to your UNIX prompt.

Appendix C
Using Prism with CMAX

You can use Prism with source code you have translated from Fortran 77 to CM Fortran using the CMAX Converter. Prism lets you:

- View both the Fortran 77 source code and the corresponding CM Fortran source code at the same time, using Prism's split source window. Or, you can view either code individually.
- Set breakpoints and create other events in terms of either the Fortran 77 or CM Fortran code.
- View the call stack in terms of either the Fortran 77 or CM Fortran code.
- View performance data in terms of either Fortran 77 or CM Fortran code.

This appendix describes how to use Prism with a CMAX-translated program.

C.1 How Prism Can Display Both Source Files

To display both Fortran 77 and CM Fortran source files, Prism uses a mapping file that is created as part of the CMAX translation. This mapping file is named *filename*`.ttab`, where *filename* is the name (minus the extension) of the Fortran 77 and CM Fortran source files. If Prism can't find this file, it won't be able to load the Fortran 77 source file.

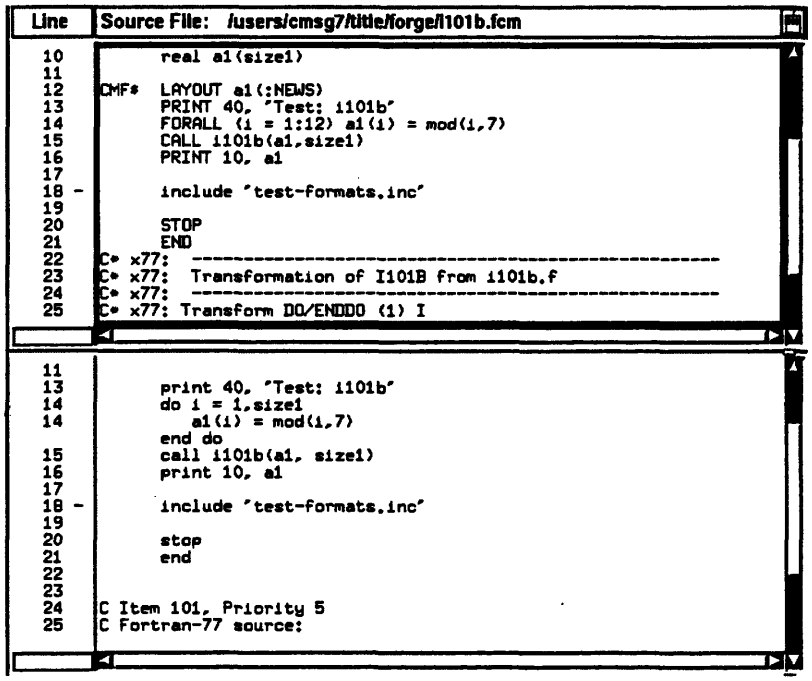
C.2 Splitting the Source Window

Begin by loading the executable CM Fortran program into Prism.

You then split the source window to display the corresponding Fortran 77 code by following these steps:

1. Right-click in the source window to display the source-window popup menu.
2. Choose **Show source pane** from this menu.
3. This displays another menu. Choose **Show .f source** from this menu.

You then see a split screen like that shown in Figure 44. The CM Fortran source code is in the top pane; the corresponding Fortran 77 source code is in the bottom pane.



```
 10          real a1(size1)
 11
 12    CMF*  LAYOUT a1(:NEWS)
 13          PRINT 40, 'Test: i101b'
 14          FORALL (i = 1:12) a1(i) = mod(i,7)
 15          CALL i101b(a1,size1)
 16          PRINT 10, a1
 17
 18 -        include 'test-formats.inc'
 19
 20          STOP
 21          END
 22    C* x77:  -------------------------------------------------
 23    C* x77:  Transformation of I101B from i101b.f
 24    C* x77:  -------------------------------------------------
 25    C* x77: Transform DO/ENDDO (1) I
```

```
 11
 13          print 40, 'Test: i101b'
 14          do i = 1,size1
 14             a1(i) = mod(i,7)
             end do
 15          call i101b(a1, size1)
 16          print 10, a1
 17
 18 -        include 'test-formats.inc'
 19
 20          stop
 21          end
 22
 23
 24    C Item 101, Priority 5
 25    C Fortran-77 source:
```

Figure 44. CM Fortran and Fortran 77 code in a split screen.

When your screen is split, breakpoints you set in the line-number region of one pane also appear at the corresponding line in the other pane.

To return to a single source window, put the mouse pointer in the pane you want to delete, right-click, and choose **Hide this source pane** from the popup menu.

NOTE: You can also choose **Show .s source** from the **Show source pane** menu; this displays the assembly code for the executable program. You can therefore have a three-way split of your source window, displaying the CM Fortran, Fortran 77, and assembly code versions of the program.

C.2.1 From the Command Line

To split the source window, issue the **show** command, using as an argument the file extension of the source file you want to see in the other pane. For example, when you first load the program, only the CM Fortran source code is visible. To see the Fortran 77 source code in a separate pane, issue the command:

```
show .f
```

To return to a single source window, issue the **hide** command, specifying the file extension of the source code you no longer want to see. For example, to display only the Fortran 77 code, issue the command:

```
hide .fcm
```

C.3 Using the Master Pane

When you split the source window, the top (CM Fortran) pane is highlighted; this is the *master pane*. Left- or middle-click in the other pane to make it the master.

The master pane controls several aspects of the way Prism operates when the screen is split. Specifically:

- Scrolling through the master pane causes the slave pane to scroll to the corresponding location. You can scroll the slave pane independently, but this does not cause the master pane to scroll.

- The line numbers shown in the slave pane are in terms of the lines in the master pane. Thus, several Fortran 77 lines in the slave pane could have the same line number, if they were all converted to a single CM Fortran statement. Similarly, if the Fortran 77 pane is master, the CM Fortran pane may skip a line number, if it doesn't have code corresponding to that Fortran 77 line.

- Prism interprets all unqualified line numbers in commands as referring to the source code in the master pane. You can still refer to a line number in the other source code, but you must qualify it with the filename. For example, if CM Fortran is the master pane, you would specify a breakpoint in the Fortran 77 code like this:

  ```
  stop at "foo.f":20
  ```

- Prism displays line numbers for the master source code in the event table, the **Where** window, and in messages in the command window.
- Prism displays the files for the master source code in the **File** window.
- Prism displays master source code in response to a `list` command.
- Source-line histograms in the **Performance Data** window show the master source code.
- The `?` and `/` search commands search in the master pane only.
- If you choose **Edit** from the **Utilities** menu, the master source code appears in the editor.

C.3.1 From the Command Line

To choose the master pane from the command line, issue the `select` command, specifying the extension of the source code you want to be in the master pane. For example, to make the Fortran 77 code be in the master pane, issue this command:

```
select .f
```

C.4 Displaying Corresponding Source Lines

Prism lets you graphically display the line or lines in one pane that correspond to a line in another pane. Press the **Shift** key and left-click in the line-number region next to the line you are interested in. A pound sign (#) appears next to the line; the same character appears next to the corresponding line(s) in the other pane.

C.5 Debugging

You can debug in terms of either the Fortran 77 or CM Fortran source code. For example, setting a breakpoint in one source code causes it to be set in the corresponding location in the other source code.

If you hide the CM Fortran pane, it will look as if you are debugging your Fortran 77 code directly. Note, however, that this is not exactly the case — you are actually debugging a CM Fortran executable program. This means that you will not be able to set a breakpoint at a line that has been optimized away in the translation. For example, if you try to set a breakpoint in the middle of a DO loop, Prism insists on putting the breakpoint at the beginning of the loop. Similarly, you can't print a Fortran 77 variable that has been optimized away.

C.6 Analyzing Performance

You can use Prism to analyze the performance of your translated CM Fortran program. This will tell you if CMAX didn't translate a Fortran 77 construct into the most efficient CM Fortran equivalent. For example:

- Unvectorized loops are treated as serial code in CM Fortran and are executed on the partition manager; if you use Prism's performance analysis feature, these loops will show up as using the **Partition manager (user)** resource (on the CM-5) or the **Front end manager (user)** resource (on the CM-2/200).
- Fortran 77 code that could be translated into a CSHIFT may be translated as an assignment of array sections. This will generate unnecessary Send/Get communication.

Note that you have two choices in working with your program:

- You can edit the Fortran 77 code, and then use CMAX to retranslate the code into CM Fortran.

- You can edit the translated CM Fortran code directly. If you do this, you can no longer use Prism's split source window to view the original Fortran 77 source, since the Fortran 77 code and the CM Fortran code won't be equivalent.

C.7 Using Commands-Only Prism

Although you don't have the benefit of using the split source window, you can use commands-only Prism with a CMAX-translated program. Use the **select** command to specify the version of the source code in which you want to debug. For example, issue this command to debug in terms of the Fortran 77 source code:

```
(prism) select .f
```

To subsequently debug in terms of the CM Fortran source code, issue this command:

```
(prism) select .fcm
```

Appendix D

Glossary

This is a glossary of specialized terms used in Prism. The glossary is also available on-line; choose the **Glossary** selection from the **Help** menu.

alias	An alternative name for a Prism command. You can set up these alternative names via the `alias` command.
breakpoint	An event that stops execution of a program at a specific location, when a condition is met, or when a variable or expression changes its value.
call stack	The list of procedures and functions currently active in a program.
command window	The pane at the bottom of the main Prism window, in which messages are displayed and the user can issue commands.
context	In printing and displaying data, the active elements of a variable or expression. Prism handles active elements differently from inactive elements in certain data representations.
current file	The source file currently being displayed in the source window. When the program is first loaded, this is the file that contains the `main` function. This can change during execution, or as a result of actions you perform in Prism. The current file determines the scope that Prism uses in identifying line numbers and variables.

current function	The function or procedure displayed in the source window. This is the `main` function when a program is first loaded. It can change during execution, or as a result of performing certain actions in Prism. Prism uses the current function to determine the scope it should use in identifying variables.
current line	The source line at which the program is currently stopped. The execution pointer (>) points to the current line in the source window.
data navigator	A component of a visualizer that lets you manipulate the display window relative to the data being visualized.
dialog box	A window used by Prism to obtain information from or provide information to the user.
display window	The pane within a visualizer that shows the data.
event	A breakpoint or trace, along with associated actions, that the user creates to control the execution of a program.
event list	The list of events, displayed as part of the event table or via the **`show events`** command.
event table	A table that lists the events that are to take place during the execution of a program, and provides mechanisms for adding, editing, and deleting these events.
execution pointer	The greater-than symbol (>) that appears in the line number region and points to the next line in the source window to be executed.
Help Index	The list of entries on which help is available.
history region	The area of the command window where Prism displays messages and responses to commands.
I-beam	An I-shaped graphical image that appears in a text entry box to show that text can be entered in it.

immediate action	An action that takes place as soon as a menu selection is chosen; no dialog box or window is displayed to obtain further information from the user.
keyboard accelerator	A sequence of keystrokes that performs an action without the need to display a menu.
line-number region	The area to the left of the source window in which line numbers are displayed. The user can set breakpoints in this region.
location cursor	A graphical image that represents the focus of keyboard actions; it is displayed as a box surrounding the selected object.
menu bar	The line of text across the top of the main Prism window. A pulldown menu is associated with each word along this line; you can choose items from these menus to perform actions in Prism.
mnemonic	A single letter (generally the first letter) underlined in a menu title or menu selection; by typing this letter, you can display the menu or (when the menu is displayed) choose the menu selection.
mouse pointer	The graphical image (for example, an arrow head) that appears on the screen and represents the current location of the mouse.
performance advisor	An analysis that Prism provides of performance data.
Prism resource	A variable that controls an aspect of Prism's behavior. Default values for many Prism resources appear in Prism's file in your system's `app-defaults` directory. You can change these defaults by specifying new values in your X resource database, or by using the **Customize** utility.
qualified name	A version of the name of a variable or function that identifies it more completely within a program. For example, `'foo'bar'x` identifies the variable `x` in the function `bar` in the source file `foo`. Names can be either *partially qualified* or *fully qualified.*

resize box	The small box in the corner of many windows; you can drag this box to resize the window.
resolving names	The procedure by which Prism determines which (of possibly several) variables or procedures with the same name it is to use (for example, in an expression).
resource	In performance analysis, one of the subsystem components (for example, CM cpu time, Send/Get communication) for which Prism measures a program's usage.
scope pointer	The - symbol in the line-number region. It indicates the beginning of the scope that Prism uses for identifying variables.
source window	The pane in Prism's main window where source code is displayed.
subsystem	In performance analysis, one of the independent systems for which Prism measures a program's usage. For a CM-2 or CM-200, they are the front end and the CM. For the CM-5, they are the partition manager and the nodes. The front end and partition manager are referred to as *serial* subsystems. The CM and the nodes are referred to as *parallel* subsystems.
tear-off region	The area beneath the menu bar in Prism's main window. You can move frequently used menu selections and Prism commands to this region to make them more accessible.
trigger condition	An action that causes an event to take place. Trigger conditions include reaching a program location, a logical condition becoming true, and the value of a variable changing.
visualizer	A window that displays scalar or parallel data using one of several visual representations available in the Prism.
watchpoint	An event that occurs when the value of a variable changes.

Index