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 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.

Copyright © 1991–1994 Thinking Machines Corporation

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

- The execution pointer moves to the current execution point.
- A message is printed in the command window.
- If you specified that an action was to accompany the event (for example, the printing of a variable's value), it is performed.
- If the event is a trace, execution then continues. If it is a breakpoint, execution does not resume until you explicitly order it to (for example, by choosing **Continue** from the **Execute** menu).

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

See Section 10.5.1 for a discussion of events in Node Prism.

You can define events so that they occur:

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

These are referred to as *triggering conditions*.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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.

Copyright © 1991–1994 Thinking Machines Corporation

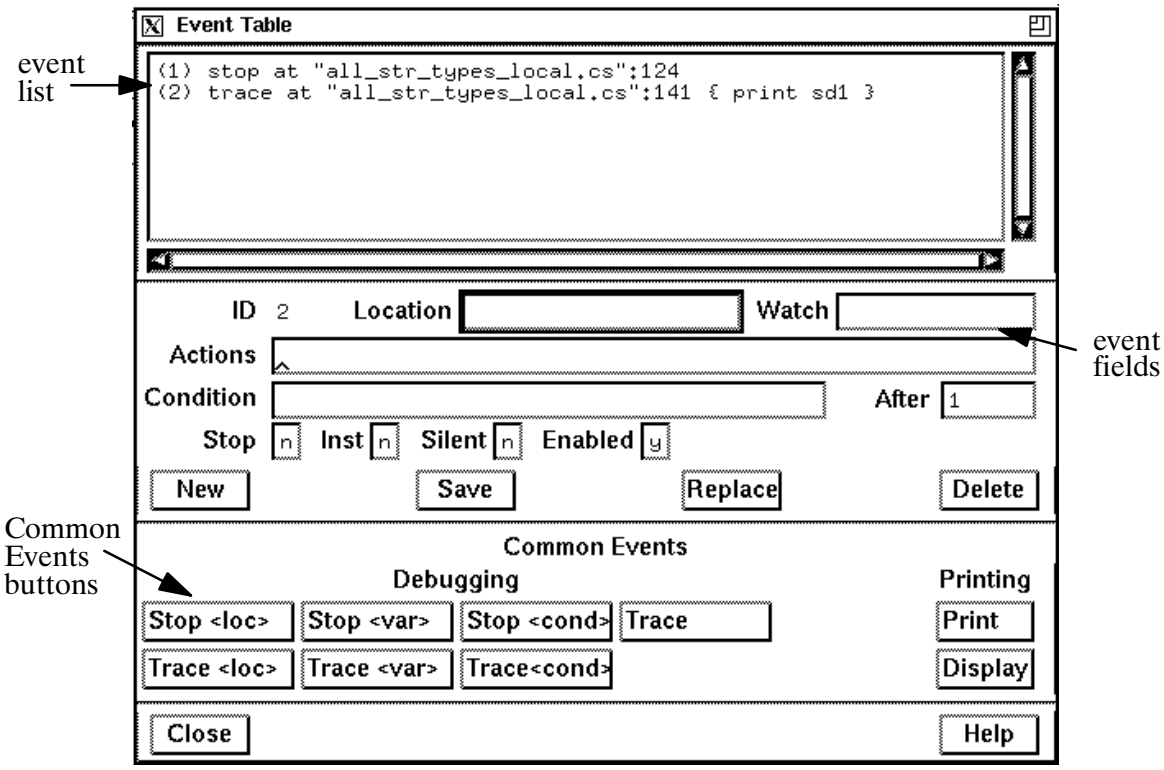

Figure 14. The event table.

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.

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

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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.

Copyright © 1991–1994 Thinking Machines Corporation

- **Inst.** Use this field to specify whether to display a disassembled assembly-language instruction when the event occurs.
- **Silent.** Use this field to specify whether or not the event is to cause a message to appear in the command window when it occurs.
- **Enabled.** Use this field to specify whether the event is enabled. Putting an **n** in this field disables the event; it still exists, but it does not affect program execution.

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

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

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

4.2.2 Adding an Event

You can either add an event 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).

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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.

Copyright © 1991–1994 Thinking Machines Corporation

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
   ```

4.2.6 Enabling and Disabling Events

You can disable and enable events. When you disable an event, Prism keeps it in the event list, but it no longer affects execution. You can subsequently

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

enable it when you once again want it to affect execution. This can be more convenient than deleting events and then redefining them.

From the event table: The event table has an **Enabled** field. By default, there is a **y** in this field, meaning that the event being defined or edited is enabled. Click on the field and change the **y** to an **n** to disable the event. The event remains in the event list, but is labeled **(disabled)**. You can then edit the event as described in Section 4.2.4 and change the field back to a **y** to enable the event once again.

From the command line: Issue the **disable** command to disable an event. Use the event's ID as the argument. You can obtain this ID from the event list in the event table, or by issuing the **show events** command.

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

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

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

4.3 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

Copyright © 1991–1994 Thinking Machines Corporation

- 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.

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

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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

See Section 10.5.1 for a discussion of the line-number region in Node Prism.

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.

Copyright © 1991–1994 Thinking Machines Corporation

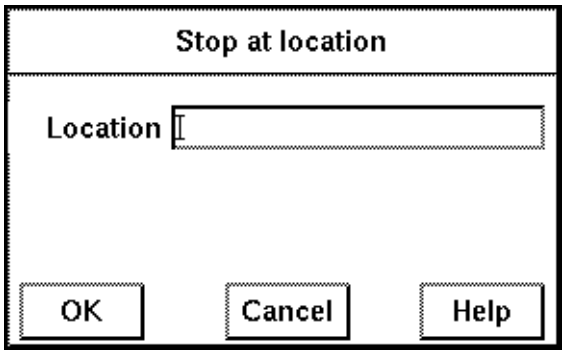

Figure 15. The Stop <loc> dialog box.

- **Stop <var>** prompts for a variable name. The program stops when the variable's value changes. The variable can be an array, an 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

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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,

Copyright © 1991−1994 Thinking Machines Corporation

{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.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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. If you trace a source line, Prism displays a **T** next to the line in the line-number region.

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

Copyright © 1991–1994 Thinking Machines Corporation

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.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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.

Copyright © 1991–1994 Thinking Machines Corporation

4.5 Displaying and Moving through the Call Stack

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

See Section 10.5.2 for a discussion of displaying the call stack in Node Prism.

4.5.1 Displaying the Call Stack

From the menu bar: Choose the **Where** selection from the **Debug** menu. The **Where** window is displayed; see Figure 16. The window contains the call stack; it is updated automatically when execution stops or when you issue commands that change the stack.

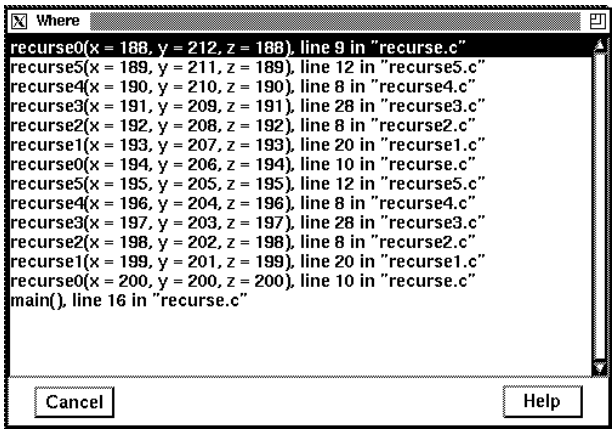

Figure 16. The Where window.

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 run-time 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`** initializa-

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

tion 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.

See Section 10.5.4 for a discussion of examining the contents of vector-unit registers in Node Prism.

Copyright © 1991–1994 Thinking Machines Corporation

4.6.1 Displaying Memory

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

```
0x10000/
```

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

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

```
&x/
```

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

```
0x000237f8:  0x3f800000
```

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

```
0x1000+100/
```

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

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

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

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

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation

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

Copyright © 1991–1994 Thinking Machines Corporation

4.7 Debugging Message-Passing Programs on a CM-5

As of CMOST Version 7.3, the standard method for debugging message-passing programs on a CM-5 is via Node Prism. See Chapter 10 for complete information on Node Prism.

If you are running an older version of CMOST, you can use the node-level debugger **pndbx**.

To start **pndbx** from within Prism, 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*.

If your CM-5 is running CMOST Version 7.3, choosing the **PN Debug** selection starts up Node Prism.

Version 2.0, April 1994
Copyright © 1991–1994 Thinking Machines Corporation