Prism™ 7.0 Software Reference Manual

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-0089-10
February 2003, Revision 01

Send comments about this document to: `docfeedback@sun.com`

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.

Contents

Tables

Preface

This manual provides reference descriptions of commands available in the Prism™ environment.

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

Using UNIX Commands

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

See one or more of the following for this information:

- *Solaris Handbook for Sun Peripherals*
- AnswerBook2™ online documentation for the Solaris™ operating environment
- Other software documentation that you received with your system

Typographic Conventions

Typeface[*]	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

[*] The settings on your browser might differ from these settings.

Table with descriptions and examples of the typographic conventions that are used in this book.

Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Table with examples of the types of shell prompts that are used in this book.

Related Documentation

Application	Title	Part Number
Sun HPC ClusterTools Documentation	*Read Me First: Guide to Sun HPC ClusterTools Documentation*	817-0096-10
Sun HPC ClusterTools Software	*Sun HPC ClusterTools 5 Product Notes*	817-0081-10
	Sun HPC ClusterTools 5 Installation Guide	817-0082-10
	Sun HPC ClusterTools 5 Performance Guide	817-0090-10
	Sun HPC ClusterTools 5 Administrator's Guide	817-0083-10
	Sun HPC ClusterTools 5 User's Guide	817-0084-10
Sun MPI Programming	*Sun MPI 6.0 Programming and Reference Guide*	817-0085-10
Sun S3L	*Sun S3L 4.0 Programming Guide*	817-0086-10
	Sun S3L 4.0 Reference Manual	817-0087-10
Prism™ graphical programming environment	*Prism 7.0 Software User's Guide*	817-0088-10

Table listing other documents that are related to this book or product.

Accessing Sun Documentation

You can view, print, or purchase a broad selection of Sun documentation, including localized versions, at:

`http://www.sun.com/documentation`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (817-0089-10) of your document in the subject line of your email.

Command Reference

This reference manual gives, in alphabetical order, the syntax and reference description of every command in the Prism™ programming environment. This information is also available online:

- Choose the Commands Reference selection from the Prism Help menu to obtain reference information about all Prism commands.
- Type `help commands` on the Prism command line to obtain summary information about Prism commands.
- Issue a command of the form `help` *commandname* on the command line to display the reference description of the command.

TABLE 4 lists the commands discussed in this manual.

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 the `where` command into the file `where.output` in your current working directory within the Prism environment.

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

- `command` (abbreviated `com`). *commandname* `on command` sends output to the command window; it is the default.

- `dedicated` (abbreviated `ded`). *commandname* `on ded` sends output to a window dedicated to output for this command. If you subsequently issue the same command (no matter what its arguments) and specify that output is to be sent to the dedicated window, this window will be updated.
- `snapshot` (abbreviated `sna`). *commandname* `on snapshot` 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, the Prism environment 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. You can then issue a command using your window name, for example: *commandname* `on` *myname*. The name *myname* will appear in the title of the window.

Note – You cannot redirect the output of the commands `edit`, `make`, and `sh`.

Psets: Processes and Threads

When viewing multiprocess or multithreaded programs (including single-process programs with multiple threads), the Prism environment provides a method by which certain commands can take a set of processes or threads, or both, called a *pset*, as a qualifier. Note that psets are not available when viewing nonthreaded scalar programs.

Commands that take a pset qualifier are listed in TABLE 1. The format for commands taking a pset qualifier is

```
command pset [pset_name | pset_definition]
```

where *pset_definition* can include pset names (predefined or user-defined names), process numbers, thread numbers, expressions composed of combinations of such specifiers, and snapshots of all or part of such psets; see the `define pset` command for a discussion of how to define a pset. For a detailed description of psets, see the *Prism User's Guide*.

Place the `pset` qualifier after any arguments to the command, but before the optional `on` *window* syntax that specifies the window to which output is directed (see "Redirecting Output" on page 1). A command with a pset qualifier applies only to the processes (and threads) in the set. If you omit the qualifier, the command applies to the processes (and threads) in the current set.

The commands listed in TABLE 1 can take a pset qualifier.

TABLE 1 Commands Taking a Pset Qualifier

address`/`	`lwps`	`sync, syncs`
`assign`	`mpimsgs`	`thread, threads`
`call`	`next, nexti`	`trace, tracei`
`catch`	`print`	`wait`
`cont, contw`	`pstatus`	`whatis`
`display`	`return, stepout`	`where`
`ignore`	`step, stepi`	
`interrupt`	`stop, stopi`	

In summary, using the Prism environment, you can:

- Define and view groups of processes
- Define and view groups of threads within a single process
- Define and view groups of threads spanning processes

Prism documentation describes, primarily, the multiprocess (MP) mode of the Prism environment. The documentation distinguishes the MP mode from the scalar mode, which you can use to view nonthreaded scalar programs. The scalar mode does not support some features found in the MP mode, such as psets. For further information on the scalar mode, see the appendix in the *Prism User's Guide*.

Getting Information About Threads

The Prism environment includes several commands that provide information about threads in the currently loaded program. These commands are described in TABLE 2.

TABLE 2 Thread-Related Prism Commands

Command	Description
`thread`	Shows information about the last-stopped thread on each process with members in the current (or specified) pset
`threads`	Shows the current stopping point for all threads in processes that have a member in the current (or specified) pset
`lwps`	Shows all light-weight processes (LWPs) in the set of processes belonging to the current pset. Although Prism does not support debugging in terms of LWPs, it makes the mapping from thread identifier to LWP identifier available to you with the `lwps` command
`sync`	Shows information about a specified (by address) synchronization object (mutex lock)
`syncs`	Shows a list (with addresses) of all synchronization objects (mutex locks) for last-stopped threads in processes with members belonging to the current (or specified) pset

The states of threads and light-weight processes (LWPs) are described in TABLE 3.

TABLE 3 Thread and LWP States

Thread and LWP States	Description
`suspended`	Thread has been explicitly suspended
`runnable`	Thread is runnable and is waiting for an LWP as a computational resource
`zombie`	When a detached thread exits (`thr_exit()`), it is in a zombie state until it has rendezvoused through the use of `thr_join()`. `THR_DETACHED` is a flag specified at thread creation time (`thr_create()`). A nondetached thread that exits is in a zombie state until it has been reaped

TABLE 3 Thread and LWP States *(Continued)*

Thread and LWP States	Description
`asleep on syncobj`	Thread is blocked on the given synchronization object. Depending on what level of support `libthread` and `libthread_db` provide, `syncobj` might be as simple as a hexadecimal address or something with more information content
`active`	Thread is active on an LWP, but Prism cannot access the LWP
`unknown`	Prism cannot determine the state
`lwpstate`	A bound or active thread state is the state of the LWP associated with it
`running`	LWP was running but was interrupted
`syscall num`	LWP stopped on an entry into the given system call number
`syscall return num`	LWP stopped on an exit from the given system call number
`job control`	LWP stopped due to job control
`LWP suspended`	LWP is blocked in the kernel
`single stepped`	LWP has just completed a single step
`breakpoint`	LWP has just hit a breakpoint
`fault num`	LWP has incurred the given fault number
`signal name`	LWP has incurred the given signal
`process sync`	The process to which this LWP belongs has just started executing
`LWP death`	LWP is in the process of exiting

Prism Commands

TABLE 4 lists all the commands in the Prism environment in alphabetical order and provides brief descriptions. It is followed by the complete command reference, also in alphabetical order.

TABLE 4 Prism Commands

Command	Use
`/`*regexp*	Searches forward in the current file for the regular expression, *regexp*
`?`*regexp*	Searches backward in the current file for the regular expression, *regexp*
address`/`	Prints the contents of memory addresses
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 or job
`call`	Calls a procedure or function
`catch`	Tells Prism to catch the signal you specify
`cd`	Changes the current working directory
`cont`	Continues execution
`contw`	Continues execution and then waits for members of the current pset to finish execution (MP Prism environment only)
`core`	Associates a core file with an executable program (not available in MP Prism environment)
`cycle`	Makes the next member of the `cycle` pset the current set (MP Prism environment only)
`define pset`	Creates a named pset (MP Prism environment only)
`delete`	Removes one or more events from the event list
`delete pset`	Deletes a user-defined pset (MP Prism environment only)
`detach`	Detaches from a running process or job
`disable`	Disables an event
`display`	Displays the values of one or more expressions or variables
`down`	Moves the symbol-lookup context down one level

TABLE 4 Prism Commands *(Continued)*

Command	Use
`dump`	Prints the names and values of local variables
`edit`	Calls up an editor
`enable`	Enables a previously disabled event
`eval pset`	Updates the membership of a variable pset (MP Prism environment only)
`fg`	Runs the executable program in the foreground (MP Prism environment only)
`file`	Sets the source file to the specified file name
`func`	Sets the current function to the specified function name
`help`	Lists currently implemented commands
`hide`	Hides a pane of a split source window (not available in commands-only Prism)
`ignore`	Tells Prism to ignore the specified signal
`interrupt`	Interrupts execution of processes (MP Prism environment only)
`kill`	Kills a process or job running within Prism
`list`	Lists lines in the current source file
`load`	Loads a program
`log`	Creates a log file of your commands and Prism's responses
`lwps`	Lists all LWPs in the processes belonging to the current pset
`make`	Executes the `make` utility
`mpimsgs`	Displays the contents of MPI message queues
`next`	Executes one or more source lines, stepping over functions
`nexti`	Executes one or more instructions, stepping over functions
`print`	Displays the values of one or more expressions or variables
`printenv`	Displays currently set environment variables
`process`	Sets or displays the current process of the current pset (MP Prism environment only)
`pset`	Sets or displays the current pset (MP Prism environment only)
`pstatus`	Displays the execution status of processes (MP Prism environment only)
`pushbutton`	Adds a Prism command to the tear-off region (not available in commands-only Prism)
`pwd`	Displays the current working directory

TABLE 4 Prism Commands *(Continued)*

Command	Use
`quit`	Leaves the Prism environment
`reload`	Reloads the currently loaded program
`rerun`	Reruns the currently loaded program, using arguments previously passed to the program
`return`	Steps out to the caller of the current routine
`run`	Starts execution of a program
`select`	Chooses the master pane in a split source window
`set`	Defines an abbreviation for a variable or expression
`setenv`	Displays or sets environment variables
`sh`	Passes a command line to the shell for execution
`show`	Splits the source window (not available in commands-only Prism)
`show events`	Displays the event list
`show pset`	Displays the contents of a pset (MP Prism environment only)
`show psets`	Displays information about all psets (MP Prism environment only)
`source`	Reads commands from a file
`status`	Displays the event list
`step`	Executes one or more source lines
`stepi`	Executes one or more instructions
`stepout`	Steps out to the caller of the current routine
`stop`	Sets a breakpoint
`stopi`	Sets a breakpoint at an instruction
`sync`	Shows information about a specified (by address) synchronization object (mutex lock)
`syncs`	Lists all synchronization objects (mutex locks) for last-stopped threads in processes with members in the current (or specified) pset
`tearoff`	Adds a menu selection to the tear-off region (not available in commands-only Prism)
`thread`	Displays information about the last-stopped thread on each process with members in the current (or specified) pset
`threads`	Displays the current stopping point for all threads in processes that have a member in the current (or specified) pset
`trace`	Traces program execution
`tracei`	Traces instructions

TABLE 4 Prism Commands *(Continued)*

Command	Use
`type`	Specifies the data type of an S3L array handle, allowing Prism to display and visualize the S3L array
`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 (not available in commands-only Prism)
`up`	Moves the symbol-lookup context up one level
`use`	Adds a directory to the list to be searched for source files
`varsave`	Saves values of a variable or expression to a file
`wait`	Waits for a process or processes to stop execution (MP Prism environment only)
`whatis`	Displays the type of a variable
`when`	Sets a breakpoint
`where`	Displays a stack trace
`whereis`	Displays the list of all fully qualified names for an identifier
`which`	Displays the fully qualified name the Prism environment chooses for an identifier

/*regexp*, ?*regexp*

Searches forward or backward for a regular expression in the current source file.

SYNTAX

```
/regexp
?regexp
```

Note – *regexp* may be any regular expression, as described in the man page `regexp`(5).

DESCRIPTION

Use the `/` command to search forward in the current source file for the regular expression you specify. The `/` command searches from line *n+1* forward, wrapping after it passes the end of the file. If the expression is found, the source pointer moves to the line that contains the expression, and the line is echoed in the history region of the command window.

The `?` command works in the same way, except that it searches backward from line *n–1* in the source file, wrapping after it passes the beginning of the file.

Using `/` or `?` updates the current line, affecting subsequent executions of the `list` command. The `list` command resets the starting line for `/` and `?`. For further information, see “list” on page 48.

The `/` or `?` commands with no arguments search for the next (or previous) occurrence of the last-used regular expression. Both `/` and `?` wrap around if no match is found.

If the regular expression is not found, the Prism environment displays the message

```
No match.
```

in the history region of the command window.

Note – Because the scope pointer may be modified by this command, subsequent expression evaluation uses the resulting scope pointer for symbol resolution.

address /

Prints to the screen the contents of the specified memory address.

SYNTAX

```
address, address/[mode] [pset pset_name | pset_definition]
address | register/[count] [mode]
```

DESCRIPTION

Use this command to print the contents of memory or of a register. If two addresses are separated by commas, the Prism environment prints the contents of memory starting at the first address and continuing to the second address. If you specify a *count*, the Prism environment prints *count* locations, starting from the address you specify.

If the address is `.` (period), the Prism environment prints the address that follows the most recently printed address.

Specify a symbolic address by preceding the name with an & (ampersand). For example,

```
&x/
```

prints the contents of memory for variable *x*.

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.

Specify a register by preceding its name with a dollar sign. For example,

```
$f0/
```

prints the contents of the `f0` register. See TABLE 6 for a list of supported registers. If you specify *count* with a register, that number of registers is printed, starting with the specified register.

The *mode* argument specifies how memory is to be printed; if it is omitted, the Prism environment uses the previous mode that you specified. The initial mode is X. Supported modes are listed below.

TABLE 5 Mode Arguments Supported by the Prism Environment

Mode	Description
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.

Supported UltraSPARC™ registers are listed below.

TABLE 6 Sun UltraSPARC Registers Supported by the Prism Environment

Name	Register
`$g0–$g7`	Global registers (64 bits)
`$o0–$o7`	Output registers (64 bits)
`$l0–$l7`	Local registers
`$i0–$i7`	Input registers
`$psr`	Processor state register
`$pc`	Program counter
`$npc`	Next program counter
`$y`	Y register
`$wim`	Window invalid mask
`$tbr`	Trap base register

TABLE 6 Sun UltraSPARC Registers Supported by the Prism Environment *(Continued)*

Name	Register
`$f0–$f31`	Floating-point registers
`$fsr`	Floating status register (64 bits)
`$f0f1–$f62f63`	Floating-point registers
`$xg0–$xg7`	Upper 32 bits of `$g0–$g7` (SPARC V8 plus only, or higher)
`$xo0–$xo7`	Upper 32 bits of `$o0–$o7` (SPARC V8 plus only, or higher)
`$xfsr`	Upper 32 bits of `$fsr` (SPARC V8 plus only, or higher)
`$fprs`	Floating-point registers state (SPARC V8 plus only, or higher)
`$tstate`	Trap state register (SPARC V8 plus only, or higher)
`$fp`	Frame pointer (synonym for `$i6`)
`$sp`	Stack pointer (synonym for `$o6`)

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

value*=*base

Converts a value to the specified base.

SYNTAX

```
value=base
```

DESCRIPTION

Use the *value*=*base* command to convert the value you specify to the base you specify. 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). The Prism environment prints to the screen the converted value in the command window.

EXAMPLES

```
0x100=D
 256

256=X
 0x100

0x100=O
 0400

0400=X
 0x100
```

alias

Sets up an alias for a command or string.

SYNTAX

```
alias
alias new-name command
alias new-name [(parameters)] "string"
```

DESCRIPTION

Use the `alias` command to set up an alias for a command or string. When commands are processed, the Prism environment first checks if the word is an alias for either a command or a string. If it is an alias, the Prism environment treats the input as though the corresponding string (with values substituted for any parameters) had been entered.

For example, to define an alias `rr` for the command `rerun`, issue the command:

```
alias rr rerun
```

To define an alias called `b` that sets a breakpoint at a particular line, issue the command:

```
alias b(x) "stop at x"
```

You can then issue the command `b(12)`, which the Prism environment expands to:

```
stop at 12
```

The Prism environment sets up some aliases for you automatically. Issue `alias` with no parameters to list the current set of aliases.

Issue the `unalias` command to remove an alias.

assign

Assigns the value of an expression to a variable or array.

SYNTAX

```
assign lval = expression [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `assign` command to assign the value of *expression* to *lval*. *lval* can be any value that can go on the left-hand side of a statement in the language you are using, such as a variable or a Fortran array section. The Prism environment performs the proper type coercions if the right-hand side does not have the same type as the left-hand side.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See "Psets: Processes and Threads" on page 2 for more information on pset qualifiers.

EXAMPLES

To assign the value 1 to `x`:

```
assign x = 1
```

If `x` is an array, 1 is assigned to each element.

To add 2 to each element of `array2` and assign these values to `array1`:

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

Note that `array2` and `array1` must be conformable.

attach

Attaches to a running process or job.

SYNTAX

```
attach pid | jid
```

DESCRIPTION

Use the `attach` command to attach to the running process with process ID *pid* or to the running job with job ID *jid*.

You can use the `attach` command to attach to an executable without issuing a prior `load` command. You can simply attach to the process ID or job ID. For example,

(`prism all`) **`attach`** *jid*

The `attach` command will clean up the current session before attaching to the *jid* specified in the command.

The `attach` command does not accept multiple job IDs.

However, if the job ID specified is a result of a `MPI_Comm_spawn_multiple()`, multiple Prism sessions will get created.

You can `attach` through the shell command line when you launch the Prism environment. To attach at startup, use the following syntax:

`%` **`prism –`** *pid* | *jid* | *jid_list*

where you use the dash (–) instead of the name of the executable and the name *jid_list* is a list of job IDs.

Use the `detach` command to detach a process running within the Prism environment.

call

Calls a procedure or function.

SYNTAX

```
call procedure (parameters) [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `call` command to call the specified procedure or function at the current stopping point in the program. The Prism environment executes the procedure as if the call to it had occurred from the current stopping point. Breakpoints within the procedure are ignored, however.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

catch

Tells the Prism environment to catch the specified Solaris™ signal.

SYNTAX

```
catch [number | signal_name] [pset pset_name | pset_definition]
```

DESCRIPTION

The Prism environment can intercept Solaris signals before they are sent to the program. Use the catch command to tell the Prism environment to catch the signal you specify. When the Prism environment receives the signal, execution stops, and the Prism environment prints a message. A subsequent cont from a naturally occurring signal that is caught causes the signal to be propagated to signal handlers in the program (if any); if there is no handler for the signal, the program terminates—in other words, the program proceeds as if the Prism environment were not present.

By default, the Prism environment catches all signals except SIGHUP, SIGEMT, SIGKILL, SIGALRM, SIGTSTP, SIGCONT, SIGCHLD, and SIGWINCH; use the ignore command to add other signals to this list.

Specify the signal by number or by name. Signal names are case-insensitive, and the SIG prefix is optional.

Issue catch without an argument to list the signals that the Prism environment is set to catch.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

cd

Changes the current working directory.

SYNTAX

```
cd [directory]
```

DESCRIPTION

Use the `cd` command to change your current working directory in the Prism environment to *directory*; with no arguments, `cd` makes your login directory the current working directory.

The `cd` command is identical to its Solaris counterpart. See your Solaris documentation for more information.

cont

Continues execution of a target program.

SYNTAX

```
cont [number | signal_name] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `cont` command to continue execution of the process from the point at which it stopped. If you specify a Solaris signal, either by name or by number, the process continues as though it received the signal. Otherwise, the process continues as though it had not been stopped.

You can use the default alias `c` for this command.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

contw

Continues execution and then waits for the members of the current pset to finish execution. The `contw` command is available only in the MP Prism environment.

SYNTAX

```
contw [number | signal_name] [pset pset_name | pset_definition]
```

DESCRIPTION

The `contw` command is an alias for

`cont; wait`

Issuing the command continues execution of the process from the point at which it stopped, then waits for the members of the current pset to finish execution. Most Prism commands are unavailable during this time.

If you specify a Solaris signal, either by name or by number, the process continues as though it received the signal. Otherwise, the process continues as though it had not been stopped.

This command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

core

Associates a core file with the loaded program.

SYNTAX

```
core corefile
```

DESCRIPTION

Use the `core` command to associate the specified core file with the program currently loaded in the Prism environment. The Prism environment reports the error that caused the core dump and sets the current line to the location at which the error occurred. You can then work with the program within the Prism environment—for example, you can print the values of variables. You cannot continue execution from the current line, however.

The `core` command is not available in the MP Prism environment. Instead, you must specify the name of the process core file on the shell command line, after the name of the program executable. For example,

```
% prism a.out core
```

See the *Prism User's Guide* for more information.

cycle

Makes the next member of the `cycle` pset the current set. The `cycle` command is available only in the MP Prism environment.

SYNTAX

```
cycle
```

DESCRIPTION

Use the `cycle` command in the MP Prism environment to cycle through the members of the `cycle` pset. The `cycle` pset is by default equivalent to the current set; you can set it to some other set via the `define pset` command.

In a nonthreaded program, issuing the `cycle` command sets the current process to the next one in the `current` pset. In threaded programs, it sets the current thread to the next valid thread in the current process, and steps to the next process when appropriate. This provides a convenient way of looking at each individual member within a pset.

EXAMPLE

This example defines a pset, makes it current, then cycles through its members, making each one the current set in turn:

```
(prism all) define pset foo 0:3
(prism all) pset foo
(prism foo) cycle
(prism 1) cycle
(prism 2) cycle
(prism 3) cycle
(prism 0)
```

define pset

Creates a named pset. The `define pset` command is available only in the MP Prism environment.

SYNTAX

```
define pset name definition
```

DESCRIPTION

Use the `define pset` command to create a pset with the membership you specify.

You can give a pset any name except the predefined names `all`, `running`, `error`, `interrupted`, `break`, `stopped`, `done`, `current`, and `cycle`. The name must begin with a letter; it may contain any alphanumeric character plus the dollar sign and underscore.

For the *definition*, specify any of the following, singly or in combination:

- *An individual process (or thread) number.*
- *The name of a pset*. The new pset will have the same definition as the existing set.
- *A list of process (or thread) numbers*. Separate the numbers with commas. Use a colon between two process (or thread) numbers to indicate a range. Use a second colon to indicate the stride to be used within this range.
- *A union, difference, or intersection of psets*. To specify the union, use the symbol `+`, `|`, or `||`. To specify the difference, use the minus sign (–). To specify the intersection, use the symbol &, &&, or *.The Prism environment evaluates these expressions from left to right. For a union, if a process returns `true` for the first part of the expression, it is not evaluated further. For an intersection, if a process returns `false` for the first part of the expression, it is not evaluated further.
- *A snapshot of a pset expression*. Use the `snapshot` (*pset_expression*) intrinsic (parentheses are required) to define a pset with a constant value (in a multithreaded program) which could otherwise change during program execution.
- *A condition to be met*. Put braces around an expression that evaluates to true or false on each process. Processes in which the expression is true are part of the set. This is referred to as a *variable* pset, since membership in it can vary depending on the current state of your program. Use the command `eval pset` to update the membership of a variable pset.

If a variable is not active in a process, the Prism environment prints an error message and does not execute the command. To ensure that the command is executed, use the intrinsic `isactive` in the pset definition. The expression `isactive`(*variable*) returns `true` if *variable* is on the stack for a process or is a global. If *variable* is not fully qualified, it must be within the scope of the current process.

If the Prism environment tries to evaluate a process that is running, the evaluation fails and the command is not executed. To avoid this, use the intersection of the predefined set `stopped` and the expression you want to evaluate. For example,

```
define pset xon stopped && {isactive(x) && (x .NE. 0)}
```

This command defines a pset `xon` consisting of processes that are stopped and in which `x` is active and not equal to 0.

You cannot use this command in an event action.

Use the command `delete pset` to delete a pset that you have created using `define pset`.

EXAMPLES

To create a pset `foo` containing the processes 0, 4, and 7:

```
define pset foo 0, 4, 7
```

To define a pset `odd` containing the odd-numbered processes between 1 and 31:

```
define pset odd 1:31:2
```

To define a pset `quux` that contains processes that are members of either pset `foo` or pset `bar`:

```
define pset quux foo | bar
```

To define a pset `noty` that consists of all processes that are stopped except those in which `y` is equal to 1:

```
define pset noty stopped – {y == 1}
```

To define a pset, `snap1`, containing every process and thread (at the time of the snapshot) in `all` except thread 1 of process 1:

```
(prism all) define pset snap1 snapshot (all - 1.1)
```

delete

Removes one or more events from the event list.

SYNTAX

```
delete all | ID [ID…]
```

DESCRIPTION

Use the `delete` command to remove the events corresponding to the specified ID numbers (obtained by issuing the `show events` command). Use the `all` argument to delete all existing events. Deleting the events also removes them from the event list in the Event Table.

You can use the default alias `d` for this command.

delete pset

Deletes a user-defined pset. The `delete pset` command is available only in the MP Prism environment.

SYNTAX

```
delete pset pset_name
```

DESCRIPTION

Use the `delete pset` command to delete the pset *pset_name*. If you have created events that apply to this pset, the events continue to exist. Their printed representation, however, is changed so that it shows the processes that were members of the pset at the time you deleted the set.

You cannot include the `delete pset` command in an event action.

Use the command `define pset` to create a pset.

detach

Detaches a process or job running within the Prism environment.

SYNTAX

```
detach
```

DESCRIPTION

Use the `detach` command to detach the process or job that is currently running within the Prism environment. The process or job must be stopped before it can be detached. Once detached, the process or job continues to run in the background, but it is no longer under the control of the Prism environment.

The `detach` command only applies to the Prism session where it is invoked. If you issue the `detach` command in a primary session, it is not propagated down to secondary sessions.

For information about debugging multiple sessions, sessions spawned using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, see the *Prism User's Guide*.

Use the `attach` command to attach to a running process or job.

Use the `kill` command to terminate the process or job to which the Prism environment is attached.

disable

Disables one or more events.

SYNTAX

```
disable event_ID [event_ID ...]
```

DESCRIPTION

Use the `disable` command to disable the events with the specified ID numbers (obtained by issuing the `show events` command). Disabled events are kept in the event list, but they no longer affect execution. Use the `enable` command to re-enable events. This can be more convenient than deleting events and then redefining them.

display

Displays the values of one or more variables or expressions.

SYNTAX

```
[where (expression)] display[/radix] expression [, expression ...]
[pset pset_name | pset_definition]
```

DESCRIPTION

Use the `display` command to display the value(s) of the specified variable(s) or expression(s). The `display` command prints the values to the screen immediately and creates a display event, so that the values are updated automatically each time the program stops execution.

The optional `where` expression provides a mask for the elements of the parallel variable or array being displayed. The mask can be any expression that evaluates to true or false for each element of the variable or array. Elements whose values evaluate to true are considered *active*; elements whose values evaluate to false are considered *inactive*. If values are displayed in the command window, values of inactive elements are not printed. If values are displayed graphically, the treatment of inactive elements depends on the type of representation you choose.

The optional /*radix* syntax specifies the radix to be used in displaying the value(s). Possible settings of /*radix* are described in TABLE 7.

TABLE 7 Radix Settings for the `display` Command

Symbol	Radix
`/b`	Binary
`/d`	Decimal
`/x`	Hexadecimal
`/o`	Octal

The default radix setting is decimal, unless you have overridden the default via the `set $radix` command.

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.

If you don't send output to the command window (the default), separate windows are created for each variable or expression that you display. Note that displaying to a window other than the command window creates a visualizer for the data.

Thus, the commands

```
display x on dedicated
display y on dedicated
```

create two dedicated windows, one for each variable; the two windows are updated separately.

Also, by specifying `as` *representation* with the `on` *window* option, you can select the visualizer representation shown. For example:

```
display x on dedicated as colormap
display y on dedicated as histogram
```

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

```
display $pc on dedicated
```

displays the contents of the program counter register.

Supported UltraSPARC registers are listed in TABLE 8.

TABLE 8 Sun UltraSPARC Registers Supported by the Prism Environment

Name	Register
`$g0–$g7`	Global registers (64 bits)
`$o0–$o7`	Output registers (64 bits)
`$l0–$l7`	Local registers
`$i0–$i7`	Input registers
`$psr`	Processor state register
`$pc`	Program counter
`$npc`	Next program counter
`$y`	Y register
`$wim`	Window invalid mask
`$tbr`	Trap base register
`$f0–$f31`	Floating-point registers, printable only as floats

TABLE 8 Sun UltraSPARC Registers Supported by the Prism Environment *(Continued)*

Name	Register
`$fsr`	Floating status register (64 bits)
`$f0f1–$f62f63`	Floating-point registers, printable only as doubles
`$xg0–$xg7`	Upper 32 bits of `$g0–$g7` (SPARC V8 plus only, or higher)
`$xo0–$xo7`	Upper 32 bits of `$o0–$o7` (SPARC V8 plus only, or higher)
`$xfsr`	Upper 32 bits of `$fsr` (SPARC V8 plus only, or higher)
`$fprs`	Floating-point registers state (SPARC V8 plus only, or higher)
`$tstate`	Trap state register (SPARC V8 plus only, or higher)
`$fp`	Frame pointer (synonym for `$i6`)
`$sp`	Stack pointer (synonym for `$o6`)

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

EXAMPLES

To display the sum of the elements of the array `foo`:

```
display sum(foo)
```

To display (in a dedicated window) the values of `foo` that are not equal to 0:

```
where (foo .ne. 0) display foo on dedicated as text
```

down

Moves the symbol lookup context down one level in the call stack.

SYNTAX

```
down [count]
```

DESCRIPTION

Use the `down` command to move the current function down the call stack (that is, toward the current stopping point in the program) *count* levels. If you omit *count*, the default is one level.

Issuing `down` repositions the source window at the new current function.

After a series of `down` commands, the Prism environment attempts to preserve the level when the current process changes.

dump

Prints the names and values of local variables.

SYNTAX

```
dump [function |…]
```

DESCRIPTION

Use the `dump` command to print the names and values of all the local variables in the function or procedure you specify. If you omit *function*, the Prism environment uses the current function. If you specify a period (`.`), `dump` follows all stack frames from the current one back to `main` and prints the names and values of all local variables in the functions in the stack.

Note – The `dump` command is not available in the MP Prism environment.

EXAMPLE

```
(prism) stop at 8
(1) stop at "dump.c":8
(prism) stop at 19
(2) stop at "dump.c":19
(prism) run
Running: /usr/users/tjl/dump.x
Debugee pid is 13302
stopped in procedure "main" at "dump.c":8
8      sub();
(prism) dump
# Print all local variables from main()
‘dump.x‘dump.c‘main‘z = 1.900000
‘dump.x‘dump.c‘main‘x = 9
‘dump.x‘dump.c‘main‘y = 19.190000
(prism) c
stopped in procedure "sub" at "dump.c":19
19      y = y + x;
(prism) where
# Show the active procedures on the call stack
sub(), line 19 in "dump.c"
main(), line 8 in "dump.c"
(prism) dump .
# Print all local variables in all active procedures
‘dump.x‘dump.c‘sub:19‘y = 100        # from nested for() { } block
‘dump.x‘dump.c‘sub‘z = -9.100000      # from sub()
‘dump.x‘dump.c‘sub‘x = 1              # from sub()
‘dump.x‘dump.c‘sub‘y = 91.910000      # from sub()
‘dump.x‘dump.c‘main‘z = 1.900000      # from main()
‘dump.x‘dump.c‘main‘x = 9             # from main()
‘dump.x‘dump.c‘main‘y = 19.190000     # from main()
```

edit

Invokes an editor.

SYNTAX

```
edit [filename | procedure]
```

DESCRIPTION

Use the `edit` command to invoke an editor. With no arguments, the editor is invoked on the current file. If you specify *filename*, it is invoked on that file. If you specify *procedure*, it is invoked on the file that contains that procedure or function, positioning the cursor at the start of the procedure.

The editor that is invoked depends on the setting of the Prism resource `Prism.editor`. If this resource is not set, the Prism environment uses the setting of the `EDITOR` environment variable. If neither is set, the default editor is `vi`.

You cannot redirect the output of this command.

You can use the default alias `e` for this command.

enable

Enables previously disabled events.

SYNTAX

```
enable event_ID [event_ID ...]
```

DESCRIPTION

Use the `enable` command to enable the event with specified ID numbers (obtained by issuing the `show events` command). Use the `disable` command to disable events. Disabled events are kept in the event list, but they no longer affect execution. Use the `enable` command to re-enable events. This can be more convenient than deleting events and then redefining them.

eval pset

Updates the membership of a variable pset. The `eval pset` command is available only in the MP Prism environment.

SYNTAX

```
eval pset pset_name
```

DESCRIPTION

Use the `eval pset` command to update the membership of the variable pset *set_name*. You create a variable pset by issuing the `define pset` command and specifying a condition to be met. For example, to define a pset `foo` that consists of all stopped processes in which `x` is active and is greater than zero:

```
define pset foo stopped && {isactive(x) && (x>0)}
```

The membership of such a set can change as a program executes. To update its membership, issue the command:

```
eval pset foo
```

If the evaluation fails (for example, because a process that was previously stopped is now running, and you didn't include the `stopped` && syntax in your pset definition), the membership of the pset does not update.

Note – The `isactive` intrinsic requires that its variable either must be fully qualified or it must be within the scope of the current process.

fg

Runs the executable program in the foreground. The `fg` command is available only in the commands-only version of the MP Prism environment, or if you are using the graphical interface of the Prism environment without an Xterm for I/O.

SYNTAX

```
fg
```

DESCRIPTION

Use the `fg` command to bring your executable program into the foreground. When executing a message-passing program in the commands-only interface of the MP Prism environment, the program starts up in the background. Bring the program into the foreground if it needs to read terminal input. You cannot execute Prism commands while the program is executing in the foreground.

To have the program run in the background again and regain the `(prism)` prompt, type Ctrl-Z.

file

Changes or displays the current source file.

SYNTAX

```
file [filename]
```

DESCRIPTION

Use the `file` command to set the current source file to *filename*. If you do not specify a file name, `file` prints the name of the current source file.

Note – The tilde (~) is valid syntax for all file names.

Changing the current file causes the new file to be displayed in the source window. The scope pointer (`–`) in the line-number region moves to the current file to indicate the beginning of the new scope that the Prism environment uses in identifying variables.

When `file` is invoked with an absolute file name, the Prism environment searches for *filename* as specified. When invoked with a relative file name, the Prism environment searches first in the directory where *filename* was compiled. Then, if *filename* is not found, the Prism environment attempts to locate *filename* using the current-use list. For further information, see “use” on page 107.

Note – Because the scope pointer may be modified by this command, subsequent expression evaluation uses the resulting scope pointer for symbol resolution.

---

func

Changes or displays the current procedure or function.

SYNTAX

```
func [function]
```

DESCRIPTION

Use the `func` command to set the current procedure or function to *function*. If you do not specify a procedure or function, `func` prints the name of the current function.

Changing the current function causes the file containing it to be displayed in the source window; this file becomes the current file. The scope pointer (`–`) in the line-number region moves to the current function to indicate the beginning of the new scope that the Prism environment uses in identifying variables.

Invoking `func` with an invalid function name leaves the scope pointer unchanged.

The `func` command causes the function frame to be set to the first instance of the specified function, if any, on the expression stack. For example, assume that the function on the top of the stack, function `bar`, is not optimized. All of `bar`'s local variables are accessible. Issuing the Prism command:

```
func foo
```

causes `foo` to become the first instance of `foo` on the stack. If `foo` is optimized, then the only accessible variables are global variables. No local variable of `foo` is accessible and none of the local variables of function `bar` are visible (because of scope change), so none of `bar`'s variables are accessible. In other words, variables that were previously accessible are no longer accessible after issuing the command:

```
func foo
```

Note – The set of accessible variables is a subset of the set of visible variables.

---

help

Gets help.

SYNTAX

```
help [commands | command_name]
```

DESCRIPTION

Use the `help` command to get help about Prism commands.

Use the `commands` option to display a list of Prism commands. Specify a command name to display reference information about that command.

Issuing `help` with no arguments displays a brief help message.

You can use the default alias `h` for this command.

hide

Removes a pane from a split source window.

SYNTAX

```
hide file_extension
```

DESCRIPTION

Use the `hide` command to remove one of the panes in a split source window. The pane that is removed contains the code specified by the file extension you supply as the argument to the command.

Use the `show` command to create a split source window. For more information about the `show` command, see "show" on page 77.

The `hide` command is not meaningful in the commands-only interface of the Prism environment.

EXAMPLES

To remove the pane containing the assembly code for the loaded program, issue this command:

```
hide .s
```

To remove the pane containing Fortran 77 source code, issue this command:

```
hide .f
```

ignore

Tells the Prism environment to ignore the specified Solaris signal.

SYNTAX

```
ignore [number | signal_name] [pset pset_name | pset_definition]
```

DESCRIPTION

The Prism environment can intercept Solaris signals before they are sent to the program. Use the `ignore` command to tell the Prism environment to ignore the specified signal. If the signal is ignored, the Prism environment sends it to the program and allows the program to continue running without interruption; the program can then react to the signal as though the Prism environment were not there. By default, the Prism environment catches all signals except `SIGHUP`, `SIGEMT`, `SIGKILL`, `SIGALRM`, `SIGTSTP`, `SIGCONT`, `SIGCHLD`, and `SIGWINCH`; use the `catch` command to catch these signals as well.

Specify the signal by number or by name. Signal names are case-insensitive, and the `SIG` prefix is optional.

Issue `ignore` with no arguments to list the signals that the Prism environment ignores.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

interrupt

Suspends execution on processes. The `interrupt` command is available only in the MP Prism environment.

SYNTAX

```
interrupt [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `interrupt` command to suspend execution on processes.

The interrupted processes become members of the predefined pset `interrupted`.

Without a pset qualifier, `interrupt` suspends execution on the processes in the current pset. With a pset qualifier, `interrupt` suspends execution on the processes in the set you specify. See "Psets: Processes and Threads" on page 2 for more information on pset qualifiers.

EXAMPLES

To interrupt the execution of the members of the predefined pset `running`:

```
interrupt pset running
```

To interrupt the execution of process 5:

```
interrupt pset 5
```

kill

Kills a process or job running within the Prism environment.

SYNTAX

```
kill
```

DESCRIPTION

Use the `kill` command to terminate the process or job that is currently running within the Prism environment.

If you issue a `kill` command in a primary Prism session, the command will propagate to the secondary Prism sessions. That is, the Prism environment will shut down the secondary Prism sessions and the debuggees.

For information about debugging multiple sessions, sessions spawned using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, see the *Prism User's Guide*.

list

Lists lines in the current source file or specified routine.

SYNTAX

```
list [source_line_number [, source_line_number]]
list routine
```

DESCRIPTION

Use the `list` command to list lines in the current file. The source window is repositioned. The command also affects the scope that the Prism environment uses for resolving names. By default, the lines are displayed in the command window.

With no arguments, `list` lists the next 10 lines starting with the current line.

If you specify line numbers, the lines are listed from the first line number through the second.

If you specify a procedure or function, `list` lists 10 lines starting with the first statement in the procedure or function.

In the commands-only interface of the Prism environment, `list` changes the current source line (but not the current execution line) to the last line displayed. Subsequent `list` commands (or search commands, for further information, see "/regexp, ?regexp" on page 10) begin from the new current line.

In the graphical mode of the Prism environment, the current source line is indicated by a dash (–) and the current execution line is indicated by an angle bracket (`>`). If the current source line is the same as the current execution line, that line is indicated by an asterisk (`*`).

You can use the default alias `l` (lowercase letter "L") for this command.

You can repeat this command by pressing Enter.

Note – Because the scope pointer may be modified by this command, subsequent expression evaluation uses the resulting scope pointer for symbol resolution.

---

load

Loads an executable program into the Prism environment.

SYNTAX

```
load filename
```

DESCRIPTION

The `load` command loads the file specified by *filename* into the Prism environment. The file must be an executable program compiled with the appropriate debugging switch.

When you execute `load`, the name of the program appears in the `Program` field of the main Prism window, and the source code that contains the main function of the program is displayed in the source window.

Use the `reload` command to reload the program currently loaded in the Prism environment.

log

Creates a log file.

SYNTAX

```
log @ filename
log @@ filename
log off
```

DESCRIPTION

Use the `log` command to create a log file, *filename*, of your commands and the Prism environment's responses.

Use the `@@` form of the command to append the log to an already existing file.

Use `log off` to turn off logging.

lwps

Lists all lightweight processes (LWPs) in the set of processes that belong to the current (or specified) pset.

SYNTAX

```
lwps [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `lwps` command to display a list of all lightweight processes belonging to the current (or specified) pset.

This command requires the MP Prism environment. If used with a pset qualifier, it applies to the processes (not threads) with members belonging to the pset you specify. If used without a pset qualifier, it applies to the processes with members belonging to the current pset.

For information about LWP states, see TABLE 3.

make

Executes the `make` utility.

SYNTAX

```
make [option…]
```

DESCRIPTION

Use the `make` command to execute the `make` utility to update and regenerate one or more programs. You can specify any arguments that are valid in the Solaris version of `make`.

By default, the Prism environment uses the standard Solaris `make`, `/bin/make`. You can change this by using the `Customize` utility or by changing the setting of the Prism resource `Prism.make`.

You cannot redirect the output of this command.

mpimsgs

Displays the contents of MPI message queues.

SYNTAX

```
mpimsgs [send | recv  | urecv] [verbose] [pset pset_name | pset_definition]
```

where

`send` – Allows you to examine the message queue for Posted Sends.

`recv` – Allows you to examine the message queue for Posted Receives.

`urecv` – Allows you to examine the message queue for Unexpected Receives.

`verbose` – Displays additional details about the communicator and dumps the contents of each message.

`pset` – Using the pset option, you can specify the message queues you wish to view by choosing a set of processes. If you do not use the pset option, the current pset is used by default. However, among the members of the specified pset, only the message queues of the processes that are stopped are displayed. For more information about psets, see “Psets: Processes and Threads” on page 2.

DESCRIPTION

Use the `mpimsgs` command to display message queues created by a Sun MPI program. The Prism environment displays the messages in the output window, sorting the messages by rank. The fields of each message are displayed, including message `size`, `tag`, `to` (or `from`), `comm` (communicator), `protocol`, and `data type`.

Specify the `verbose` option to display more details about the communicator and to display the contents of the message.

EXAMPLE

A typical message with the `verbose` option enabled,

```
Queues for Rank 0:

** 6 Posted Sends:
  #  0: size = 40
        tag = 101
        to = 0
        comm:
          name = MPI_COMM_WORLD
          fortran handle = 1
          topology = none
          size = 4
          ranks = 0:3
        protocol = loopback
        data type = int
        contents:

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

next

Executes one or more source lines, counting functions or procedures as single statements.

SYNTAX

```
next [n] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `next` command to execute the next *n* source lines, stepping over procedures and functions. If you do not specify a number, `next` executes the next source line.

You can use the default alias `n` for this command.

You can repeat this command by pressing Enter.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

nexti

Executes one or more machine instructions, stepping over procedure and function calls.

SYNTAX

```
nexti [n] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `nexti` command to execute the next *n* machine instructions, stepping over procedures and functions. If you do not specify a number, `nexti` executes the next machine instruction.

You can repeat this command by pressing Enter.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See "Psets: Processes and Threads" on page 2 for more information on pset qualifiers.

print

Prints the values of one or more variables or expressions.

SYNTAX

```
[where (expression)] print[/radix] expression [, expression ...]
[pset pset_name | pset_definition]
```

DESCRIPTION

Use the `print` command to print to the screen the values of the specified variable(s) or expression(s).

The optional `where` expression provides a mask for the elements of the parallel variable or array being printed. The mask can be any expression that evaluates to true or false for each element of the variable or array. Elements whose values evaluate to true are considered *active*; elements whose values evaluate to false are considered *inactive*. If values are printed in the command window, values of inactive elements are not printed. If values are printed graphically, the treatment of inactive elements depends on the type of representation you choose.

The optional */radix* syntax specifies the radix to be used in printing the value(s). Possible settings of */radix* are described in TABLE 9.

TABLE 9 Radix Settings for the `print` command

Symbol	Radix
`/b`	Binary
`/d`	Decimal
`/x`	Hexadecimal
`/o`	Octal

The default radix is decimal, unless you have overridden the default via the `set $radix` command.

Redirection of output to a window via the `on` *window* syntax works slightly differently for `print` and `display` from the way it works for other commands. If you don't send output to the command window (the default), separate windows are created for each variable or expression that you print. Note that printing to a window other than the command window creates a visualizer for the data.

Thus, the commands

```
print x on dedicated
print y on dedicated
```

create two dedicated windows, one for each variable; the two windows are updated separately.

Also, by specifying `as` *representation* when you use the `on` *window* option, you can select the visualizer representation shown. For example:

```
print x on dedicated as colormap
print y on dedicated as histogram
```

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

`print $pc on dedicated`

prints the contents of the program counter register.

Supported UltraSPARC registers are listed in the following table.

TABLE 10 Sun UltraSPARC Registers Supported by the Prism Environment

Name	Register
`$g0–$g7`	Global registers (64 bits)
`$o0–$o7`	Output registers (64 bits)
`$l0–$l7`	Local registers
`$i0–$i7`	Input registers
`$psr`	Processor state register
`$pc`	Program counter
`$npc`	Next program counter
`$y`	Y register
`$wim`	Window invalid mask
`$tbr`	Trap base register
`$f0–$f31`	Floating-point registers, printable only as floats
`$fsr`	Floating status register (64 bits)
`$f0f1–$f62f63`	Floating-point registers, printable only as doubles

TABLE 10 Sun UltraSPARC Registers Supported by the Prism Environment *(Continued)*

Name	Register
`$xg0–$xg7`	Upper 32 bits of `$g0–$g7` (SPARC V8 plus only, or higher)
`$xo0–$xo7`	Upper 32 bits of `$o0–$o7` (SPARC V8 plus only, or higher)
`$xfsr`	Upper 32 bits of `$fsr` (SPARC V8 plus only, or higher)
`$fprs`	Floating-point registers state (SPARC V8 plus only, or higher)
`$tstate`	Trap state register (SPARC V8 plus only, or higher)
`$fp`	Frame pointer (synonym for `$i6`)
`$sp`	Stack pointer (synonym for `$o6`)

You can use the default alias `p` for the `print` command.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

EXAMPLES

To print the maximum value of the array `a`:

```
print maxval(a)
```

To print in a dedicated window the values of `a` that are greater than 3:

```
where (a > 3) print a on dedicated as text
```

printenv

Displays currently set environment variables.

SYNTAX

```
printenv [variable]
```

DESCRIPTION

Use the `printenv` command to display the value of the specified environment variable. If you omit *variable*, the command prints the values of all environment variables that are currently set.

The Prism environment's `printenv` command is identical to its Solaris C shell counterpart. See your Solaris documentation for more information.

process

Sets or displays the current process (or thread) of the current pset. The `process` command is available only in the MP Prism environment.

SYNTAX

```
process [process_number]
```

DESCRIPTION

Use the `process` command to change the current process (or thread) of the current pset to *process_number*. If you omit the argument, `process` displays the current process of the current pset. By default, the lowest numbered process in the pset is the default process; in threaded programs, the lowest numbered thread in the lowest numbered process in the current pset is the current thread. (The current process, among other functions, determines the scope used in interpreting the names of variables.) If you omit the argument, process displays the current process of the current pset.

You cannot include this command in event actions.

EXAMPLE

To change the current thread from thread 4 to thread 3:

```
(prism 1.4) process 1.3
(prism 1.3)
```

In the following example, as a result of changing the current pset, the current thread changes from thread 3 of process 1 to thread 5 of process 2:

```
(prism 1.3) pset (2:7).(5,6)
(prism 2:7.(5.6))
```

Note that the current pset now includes threads 5 and 6 of processes 2 through 7.

pset

Sets or displays the current pset. Controls which threads are visible or hidden in the psets of multithreaded programs. The `pset` command is available only in the MP Prism environment.

SYNTAX

```
pset [pset_name | pset_definition][-hide | -unhide pset_expression]
```

DESCRIPTION

Use the `pset` command to change the current pset. You can either specify the name of a pset or the definition of a pset. See “define pset” on page 25 for an explanation of how to define a pset.

The `(prism)` prompt changes to reflect the new current set.

Use the –`hide` *pset_expression* argument to specify the set of threads to be hidden from view in the Prism environment. Hidden threads never appear in any pset. Debugging commands have no effect on hidden threads. By default, threads 2, 3, and 4 are hidden. These are auxiliary threads created by any program linked with `libthread.so`. They are rarely of interest to programmers.

Use the –`hide` argument without *pset_expression* to show the set of currently hidden threads.

Use the –`unhide` *pset_expression* argument to specify the set of threads to be made visible from the set of currently hidden threads.

The –`hide` and –`unhide` arguments are valid only when debugging a multithreaded program.

Use the `snapshot` argument in *pset_definition* to set the current pset—which would otherwise change during program execution—to a constant value (in a multithreaded program). For further information about constant and unbounded psets, see the *Prism User’s Guide*.

With no arguments specified, `pset` displays the membership of the current process set.

You cannot include the `pset` command in an event action.

EXAMPLES

This example changes the current pset a couple of times and displays its membership:

```
(prism all) pset
The current set was created by evaluating the Pset
’all’ once at the time when it became the current set.
The set contains threads: 0:3.(1,5,6)
(prism all) pset -hide all.6
(prism all) pset
The current set was created by evaluating the Pset
’all’ once at the time when it became the current set.
The set contains threads: 0:3.(1,5).
(prism all) pset -hide
currently hiding the set: 0:3.(2:4,6)
(prism all) pset -unhide all.6
Processes 0:3.6: stopped in procedure "do_work" at
"mpmt_julia.cc":278
(prism all) pset
The current set was created by evaluating the Pset
’all’ once at the time when it became the current set.
The set contains threads: 0:3.(1,5,6).
```

This example sets the current pset to contain every process and thread (at the time of the snapshot) in `all` except process 1 and its number 1 thread:

```
(prism all) pset snapshot (all - 1.1)
```

Because you have used the `snapshot` argument, all threads except 1.1 become the current pset. Unless you explicitly change the current pset (for example, by issuing another `pset` command), the current pset will continue to have the same members, even though new threads have been created.

pstatus

Displays the execution status of pset members. The `pstatus` command is available only in the MP Prism environment.

SYNTAX

```
pstatus [pset_name | pset_definition]
```

DESCRIPTION

Use the `pstatus` command to display the execution status of the members of the pset you specify. See “define pset” on page 25 for a discussion of how to define a pset. If you issue `pstatus` with no arguments, it displays the execution status of the members of the current pset. Pset members that have the same status are grouped together.

EXAMPLE

```
(prism foo) pstatus
process 0: interrupted in procedure ″make_move″ at ″chess.c″:1261
process 1: running
processes 2,3: interrupted in procedure ″bishop_moves″ at
″chess.c″:478
processes 4,5: interrupted in procedure ″knight_moves″ at
″chess.c″:383
processes 6,7: interrupted in procedure ″generate_moves″ at
″chess.c″:883
```

pushbutton

Adds a Prism command to the tear-off region of the main window of the Prism graphic user interface.

SYNTAX

`pushbutton` *label command*

DESCRIPTION

Use the `pushbutton` command to create a customized button in the tear-off region. The button will have the label you specify; clicking on it will execute the command you specify. The label must be a single word. The command can be any valid Prism command, along with its arguments.

To remove a button created via the `pushbutton` command, either enter tear-off mode and click on the button, or issue the `untearoff` command, using *label* as its argument.

Changes you make to the tear-off region are saved when you leave the Prism environment.

This command is not available in the commands-only interface of the Prism environment.

EXAMPLE

This command creates a button labeled `printfoo` that executes the command `print foo on dedicated`:

```
pushbutton printfoo print foo on dedicated
```

pwd

Displays the path name of the current working directory.

SYNTAX

```
pwd
```

DESCRIPTION

Use the pwd command to display the path name of the current working directory in the Prism environment.

The Prism environment's pwd command is identical to its Solaris counterpart. See your Solaris documentation for more information.

quit

Leaves the Prism environment.

SYNTAX

```
quit [-all]
```

DESCRIPTION

Issue the `quit` command to immediately leave the Prism environment. Note that, unlike its menu equivalent, `quit` does not ask you if you are sure you want to quit.

When issued in the primary Prism session (of a multiple session), the `quit` command does not propagate down to the secondary sessions unless you issue the command with the `-all` option.

If the job was run by the primary Prism session, the command `quit -all` will kill the debuggees in the primary as well as the secondary Prism sessions and close all the Prism sessions.

If you attached to the job in the primary Prism session, then `quit -all` will leave the debuggees running and close all the Prism sessions.

The `-all` option is valid only in the primary Prism session.

The quit entry on the Prism File menu is the same as the Prism (command-line) `quit` command. To quit all Prism sessions, you must type

```
(prism all) quit -all
```

For information about debugging multiple sessions, sessions spawned using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, see the *Prism User's Guide*.

reload

Reloads the currently loaded program.

SYNTAX

```
reload
```

DESCRIPTION

Use the `reload` command to reload the program currently loaded in the Prism environment.

rerun

Reruns the currently loaded program, using arguments previously passed to the program.

SYNTAX

```
rerun [args] [< filename] [> filename]
```

DESCRIPTION

Use the `rerun` command to execute the program currently loaded in the Prism environment. If you do not specify *args*, `rerun` uses the argument list previously passed to the program. Otherwise, `rerun` is identical to the `run` command. You can specify any command-line arguments as *args*, and you can redirect input or output using < or > in the standard Solaris manner.

When you issue the `rerun` command in a primary Prism session, the Prism environment will clean up any the secondary Prism sessions spawned by that session. That is, the Prism environment will shut down the secondary Prism sessions and the debuggees.

For information about debugging multiple sessions, sessions spawned using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, see the *Prism User's Guide*.

return

Steps out to the caller of the current function.

SYNTAX

```
return [count] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `return` command to execute the current function, then return to its caller. If you specify an integer as an argument, `return` steps out the specified number of levels in the call stack.

`return` is a synonym for `stepout`.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

run

Executes the currently loaded program.

SYNTAX

```
run [args] [< filename] [> filename]
```

DESCRIPTION

Use the `run` command to execute the program currently loaded in the Prism environment. Specify any command-line arguments as *args*. You can also redirect input or output using < or > in the standard Solaris manner.

When you issue the `run` command in a primary Prism session, the Prism environment will clean up any the secondary Prism sessions spawned by that session. That is, the Prism environment will shut down the secondary Prism sessions and the debuggees.

For information about debugging multiple sessions, sessions spawned using calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, see the *Prism User's Guide*.

You can use the default alias `r` for this command.

select

Chooses the master pane in a split source window.

SYNTAX

```
select file_extension
```

DESCRIPTION

Use the `select` command to choose the “master pane” when the source window is split into more than one pane. The master pane will contain the code with the file extension you specify as the argument to `select`.

The Prism environment interprets unqualified line numbers in commands in terms of the source code in the master pane. It also uses the master pane to determine the source code and language to use in displaying messages, events, the call stack, and so on.

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.

When used in the commands-only interface of the Prism environment, `select` determines the programming language used to display messages, events, and so on.

EXAMPLES

To make the pane containing the loaded program’s assembly code the master pane:

```
select .s
```

To select the pane containing the Fortran 77 source code to be the master pane:

```
select .f
```

set

Defines abbreviations and sets values for variables.

SYNTAX

```
set variable = expression
```

DESCRIPTION

Use the `set` command to define other names (typically abbreviations) for variables and expressions. The names you choose cannot conflict with names in the program loaded in the Prism environment; they are expanded to the corresponding variable or expression within other commands. For example, if you issue this command:

```
set x = variable_with_a_long_name
```

then

```
print x
```

is equivalent to

```
print variable_with_a_long_name
```

In addition to `print` and `display`, the `whatis`, `whereis`, and `which` commands recognize variables set using the `set` command. For example, issuing the command `whatis x` after issuing the `set` command above produces this response:

```
user-set variable, x = variable_with_a_long_name
```

In addition, you can use the `set` command to set the value of certain internal variables used by the Prism environment. These variables begin with a `$` so that they will not conflict with the names of user-set variables. You may change the settings of these internal variables:

- `$d_precision`, `$f_precision`

  Use these variables to specify the default number of significant digits the Prism environment prints for doubles and floating-point variables, respectively. the Prism environment's defaults are 16 for doubles and 7 for floating-point variables; this is the maximum precision for these variables. The value you set applies to printing in both the command window and text visualizers. For example,

  ```
  set $f_precision = 5
  ```

  This causes the Prism environment to print five significant digits for floating-point values.

- `$history`

  The Prism environment stores the maximum number of lines in the history region in this variable. When the history region reaches the maximum, the Prism environment starts throwing away the earliest lines in the history. The default number of lines in the history region is 10,000. To specify an infinite length for the history region, use any negative number. For example,

  ```
  set $history = -1
  ```

  Maintaining a large history region uses up memory. A smaller history region, improves performance and can prevent running out of memory.

- `$fortran_string_length`

  The Prism environment uses this value as the length of a character string when the length is not explicitly specified. The default is 10.

- `$fortran_adjust_limit`

  Prism uses this value as the limit of an adjustable array. The default is 10.

- `$page_size`

  This value is used only in the commands-only interface of the Prism environment. It specifies the number of output lines the Prism environment displays before stopping and prompting with a `more?` message. The Prism environment obtains its default from the size of your screen. If you specify 0, the Prism environment never displays a `more?` message.

- `$print_width`

  This value is used only in the commands-only interface of the Prism environment. It specifies the number of items to be printed on a line. The default is 1.

- `$prompt_length`

  This value is used only in the MP Prism environment. It specifies the maximum number of characters to appear in the pset part of the `(prism)` prompt. The default is 25.

- `$radix`

  This value specifies the radix to be used for printing the values of variables. Possible settings are 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal). The default is 10.

- `$viz`

  This value specifies the default visualizer representation to be used for the `print` or `display` commands. Possible settings are “Text“, “Histogram”, “Dither”, “Threshold”, “ColorMap”, “Graph”, “Surface”, and “Vector“ (quotation marks are required).

Issue the `set` command with no arguments to display your current settings.

Issue the `unset` command to remove a user-defined setting.

setenv

Displays or sets an environment variable.

SYNTAX

```
setenv [variable [setting]]
```

DESCRIPTION

Use the `setenv` command to set an environment variable within the Prism environment. With no arguments, `setenv` displays all current settings.

Environment variables become defined or undefined in the Prism environment at the moment that `setenv` or `unsetenv` is executed. The program to be debugged inherits the Prism environment at the moment that the target program is executed. For this reason, changes to the Prism environment by `setenv` and `unsetenv` do not affect any other processes that are already running.

Although the Prism environment, and any programs executed within it, inherits its environment from the shell that created it, the `setenv` and `unsetenv` commands do not affect the shell that started the Prism environment, or the Prism executable itself.

The Prism environment's `setenv` command is identical to its Solaris C shell counterpart. See your Solaris documentation for more information.

sh

Passes a command line to the shell for execution.

SYNTAX

```
sh [command_line]
```

DESCRIPTION

Use the sh command to execute a Solaris command line from a shell; the response is displayed in the history region. If you don’t specify a command line, the Prism environment invokes an interactive shell in a separate window. The setting of your SHELL environment variable determines which shell is used; if it isn’t set, the C shell is used.

You cannot redirect the output of this command.

show

Splits the source window to display the file with the specified extension.

SYNTAX

```
show file_extension
```

DESCRIPTION

Use the `show` command to split the source window and display the assembly code, or the version of the source code with the specified extension, in the new pane.

The `show` command is not meaningful in the commands-only interface of the Prism environment.

Use the `hide` command to cancel the display of the assembly code or source-code version and return to a single source window.

EXAMPLE

To display the assembly code for the loaded program, issue this command:

```
show .s
```

show events

Displays the event list.

SYNTAX

```
show events [processnumber] [on windowname]
```

DESCRIPTION

Use the `show events` command to print the event list. The list includes an ID for each command; you use this ID when issuing the `delete` command to delete an event from the event list. You can use the `enable` and `disable` commands to control whether specified events in the event list affect execution. See the `enable`, `delete`, and `disable` commands for further information.

`show events on ded` brings up the Event Table window, just as though you selected the Event Table option from the Events menu.

If you use the optional argument *processnumber*, the `show events` command reports only for the process number specified. If *processnumber* is not specified, all events are displayed.

Note – The `show events` command does not accept a pset qualifier.

You can use the default alias `j` for this command.

show pset

Displays the contents of a pset. This command is available only in the MP Prism environment.

SYNTAX

```
show pset [pset_name | pset_definition]
```

DESCRIPTION

Use the `show pset` command to display the contents of the pset you specify. (See the `define pset` command for a discussion of how to define a pset.) With no arguments, `show pset` displays the contents of the current pset.

EXAMPLE

To display the contents of the pset `stopped`:

```
show pset stopped
The set contains the following processes: 0:3.
```

show psets

Displays information about all psets. This command is available only in the MP Prism environment.

SYNTAX

```
show psets
```

DESCRIPTION

Use the `show psets` command to display information about all currently defined psets. The output includes each set's definition, members, and current process. The sets listed include user-named sets, predefined sets, and sets that the user has defined but not named.

In either the graphical interface, or in the commands-only interface of the Prism environment started with the `–CX` option, issuing the command `show psets on dedicated` displays the Psets window.

EXAMPLE

Here is sample output from a show psets command:

```
(prism foo) show psets
foo:
 definition = 0:7
  members = 0:7
  current process = 0
break:
 definition = break
  members = nil
  current process = (none)
done:
 definition = done
  members = nil
  current process = (none)
interrupted:
 definition = interrupted
  members = 0:31
  current process = 0
error:
 definition = error
  members = nil
  current process = (none)
running:
 definition = running
  members = nil
  current process = (none)
stopped:
 definition = stopped
  members = 0:31
  current process = 0
current:
 definition = foo
  members = 0:7
  current process = 0
cycle:
 definition = foo
  members = 0:7
  current process = 0
all:
  definition = all
  members = 0:31
  current process = 0
```

source

Reads commands from a file.

SYNTAX

```
source filename
```

DESCRIPTION

Use the `source` command to read in and execute Prism commands from *filename*. This is useful if, for example, you have redirected the output of a `show events` command to a file, thereby saving all events from a previous session.

In the file, the Prism environment interprets lines beginning with # as comments. If \ is the final character on a line, the Prism environment interprets it as a continuation character.

status

Displays the event list.

SYNTAX

```
status
```

DESCRIPTION

Use the `status` command to display the event list. The list includes an ID for each command; you use this ID when issuing the `delete` command to delete an event. You can use the `enable` and `disable` commands to control whether specified events in the event list affect execution. See the `enable`, `delete`, and `disable` commands for further information.

`status` is a synonym for the `show events` command.

You can use the default alias `j` for this command.

step

Executes one or more source lines.

SYNTAX

```
step [n] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `step` command to execute the next *n* source lines, stepping into procedures and functions. If you do not specify a number, `step` executes the next source line.

You can use the default alias `s` for this command.

You can repeat this command by pressing Enter.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

stepi

Executes one or more machine instructions.

SYNTAX

```
stepi [n] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `stepi` command to execute the next *n* machine instructions, stepping into procedures and functions. If you do not specify a number, `stepi` executes the next machine instruction.

You can repeat this command by pressing Enter.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

stepout

Steps out to the caller of the current function.

SYNTAX

```
stepout [count]
```

DESCRIPTION

Use the `stepout` command to execute the current function, then return to its caller. If you specify an integer as an argument, `stepout` steps out the specified number of levels in the call stack.

`return` is a synonym for `stepout`.

stop

Sets a breakpoint.

SYNTAX

```
stop [var | at line | in func] [if expression] [{cmd; cmd …}] [after n]
[silent | disabled] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `stop` command to set a breakpoint at which the program is to stop execution. You can abbreviate this command to `st`.

The first option listed in the synopsis (*var* | `at` *line* | `in` *func*) must come first on the command line; you can specify the other options, if you include them, in any order.

var is the name of a variable. Execution stops whenever the value of the variable changes. If the variable is an array or a parallel variable, execution stops when the value of any element changes. This form of the command slows execution considerably. You cannot specify both a variable and a location.

`at` *line* stops execution when the specified line is reached. If the line is not in the current file, use the form `"`*filename*`":`*line_number*, using quotation marks around the file name.

`in` *func* stops execution when the specified procedure or function is reached. Note that the Prism environment uniformly treats `main` (the program's entry point) and `MAIN` (the main subroutine of the Fortran program) as separate and distinct entities. `stop in MAIN` will consistently give you different results than `stop in main`.

`if` *expression* specifies the logical condition, if any, under which execution is to stop. The logical condition can be any expression that evaluates to true or false. Unless combined with the `at` *line* syntax, this form of `stop` slows execution considerably.

`{`*cmd*`;` *cmd* …`}` specifies the actions, if any, that are to accompany the breakpoint. Put the actions in braces. The actions can be any Prism commands; if you include multiple commands, separate them with semicolons.

`after` *n* specifies how many times a trigger condition (for example, reaching a program location) is to occur before the breakpoint occurs. The default is 1. If you specify both a condition and an after count, the Prism environment checks the condition first.

silent allows you to create the event and gives the event the same attribute as if you had specified y in the silent field of the Event Table of the Prism graphic interface. disabled allows you to create the event, but the event is disabled as if you had specified n in the enabled field of the Event Table of the Prism graphic interface.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

EXAMPLES

To stop execution the tenth time in the function foo, print a, and execute the where command:

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

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

```
stop at ”bar”:17 if a == 0
```

To stop execution whenever the value of a changes:

```
stop a
```

To stop execution the third time a equals 5:

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

stopi

Sets a breakpoint at a machine instruction.

SYNTAX

```
stopi [var | at addr | in func] [if expression] [{cmd; cmd …}] [after n]
[silent | disabled] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `stopi` command to set a breakpoint at a machine instruction.

The first option listed in the synopsis (*var* | `at` *addr* | `in` *func*) must come first on the command line; you can specify the other options, if you include them, in any order.

var is the name of a variable. Execution stops whenever the value of the variable changes. If the variable is an array or a parallel variable, execution stops when the value of any element changes. This form of the command slows execution considerably. You cannot specify both a variable and a location.

`at` *addr* stops execution when the specified address is reached.

`in` *func* stops execution when the specified procedure or function is reached. Note that the Prism environment uniformly treats `main` (the program's entry point) and `MAIN` (the main subroutine of the Fortran program) as separate and distinct entities. `stop in MAIN` will consistently give you different results than `stop in main`.

`if` *expression* specifies the logical condition, if any, under which execution is to stop. The logical condition can be any expression that evaluates to true or false. Unless combined with the `at` *addr* syntax, this form of `stopi` slows execution considerably.

`{`*cmd*`;` *cmd* ...`}` specifies the actions, if any, that are to accompany the breakpoint. The actions can be any Prism commands; if you include multiple commands, separate them with semicolons.

`after` *n* specifies how many times a trigger condition (for example, reaching a program location) is to occur before the breakpoint occurs. The default is 1. If you specify both a condition and an after count, the Prism environment checks the condition first.

silent allows you to create the event and gives the event the same attribute as if you had specified y in the silent field of the Event Table of the Prism graphic interface. disabled allows you to create the event, but the event is disabled as if you had specified n in the enabled field of the Event Table of the Prism graphic interface.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

EXAMPLES

To stop execution at address 1000 (hex):

```
stopi at 0x1000
```

To stop execution at address 500 (hex) if a is equal to 0:

```
stopi at 0x500 if a == 0
```

sync

Shows information about a specified (by address) synchronization object (mutex lock).

SYNTAX

```
sync –info [addr] [pset pset_name | pset_definition]
```

DESCRIPTION

Shows information about the specified (by address) synchronization object (mutex lock), such as which thread it blocks or which thread owns the locks.

This command requires the MP Prism environment. If used with a pset qualifier, it applies to the threads in each of the processes with members belonging to the pset you specify. If used without a pset qualifier, it applies to the threads in each of the processes with members belonging to the current pset.

syncs

Lists all synchronization objects (mutex locks) for the last-stopped thread in processes with members in the current (or specified) pset.

SYNTAX

```
syncs [pset pset_name | pset_definition]
```

DESCRIPTION

Lists all synchronization objects (and their addresses) known to `libthread`.

This command requires the MP Prism environment. If used with a pset qualifier, it applies to the threads in each of the processes with members belonging to the pset you specify. If used without a pset qualifier, it applies to the threads in each of the processes with members belonging to the current pset.

tearoff

Places a menu selection in the tear-off region.

SYNTAX

```
tearoff ”selection”
```

DESCRIPTION

Use the `tearoff` command to add a menu selection to the tear-off region of the main window of the Prism environment. Put the selection name in quotation marks. Case and blank spaces don’t matter, and you can omit the three dots that indicate that choosing the selection displays a dialog box. If the selection name is available in more than one menu, put the name of the menu you want in parentheses after the selection name.

Use the `untearoff` command to remove a menu selection from the tear-off region.

Changes you make to the tear-off region are saved when you leave the Prism environment.

This command is not available in the commands-only interface of the Prism environment.

EXAMPLES

To put the `File` selection in the tear-off region:

```
tearoff ”file”
```

To put the Print selection from the Events menu in the tear-off region:

```
tearoff ”print (events)”
```

thread

Displays information about the last-stopped thread on each process with members in the current (or specified) pset.

SYNTAX

```
thread [–option] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `thread` command to view information about the last-stopped thread.

Options

If you omit the pset specification, `thread` displays the ID of the last-stopped thread.

- `info` – Display everything known about the last-stopped thread.
- `blocks` – List all locks held by the last-stopped thread.
- `blockedby` – Show which synchronization object (if any) blocks the last-stopped thread.

For information about thread states, see TABLE 3.

This command requires the MP Prism environment. If used with a pset qualifier, it applies to the last-stopped thread in each of the processes with members belonging to the pset you specify. If used without a pset qualifier, it applies to the last-stopped thread in each of the processes with members in the current pset.

threads

Displays a list of threads belonging to the processes in the current pset.

Syntax

```
threads [-all] [-mode all | filter] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `threads` command to view a list of threads belonging to the processes in the current pset.

Options

The options of the `threads` command are:

- `all` – Display the list of all known threads.
- `mode all|filter` – Controls whether the `threads` command displays all threads (the `all` option) or filters them by default. The `filter` option filters out all threads that have called `thr_exit()` but otherwise remain in the threads list (*zombie threads*).
- `mode` – Show which synchronization object blocks the given thread, if any.

This command requires the MP Prism environment. If used with a pset qualifier, it applies to the threads in each of the processes with members belonging to the pset you specify. If used without a pset qualifier, it applies to the threads in each of the processes with members belonging to the current pset.

trace

Traces program execution.

SYNTAX

```
trace [var | at line | in func] [if expression] [{cmd; cmd ...}] [after n]
[silent | disabled] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `trace` command to print tracing information when the program is executed. In a trace, the Prism environment prints a message in the command window when a program location is reached, a value changes, or a condition becomes true; it then continues execution.

The first option listed in the synopsis (*var* | `at` *line* | `in` *func*) must come first on the command line; you can specify the other options, if you include them, in any order.

var is the name of a variable. The value of the variable is displayed whenever it changes. If the variable is an array or a parallel variable, values are displayed if the value of any element changes. This form of the command slows execution considerably. You cannot specify both a variable and a location.

`at` *line* specifies that the line is to be printed immediately prior to its execution. If the line is not in the current file, use the form "*filename*"`:`*line_number,* placing the file name between quotation marks. You can also specify a line number without the `at`; the Prism environment will interpret it as a line number rather than a variable.

`in` *func* causes tracing information to be printed only while executing inside the specified procedure or function.

`if` *expression* specifies the logical condition, if any, under which tracing is to occur. The logical condition can be any expression that evaluates to true or false. Unless combined with the `at` *line* syntax, this form of `trace` slows execution considerably.

`{`*cmd*`;` *cmd* ...`}` specifies the actions, if any, that are to accompany the trace. Put the actions in braces. The actions can be any Prism commands; if you include multiple commands, separate them with semicolons.

`after` *n* specifies how many times a trigger condition (for example, reaching a program location) is to occur before the trace occurs. The default is 1. If you specify both a condition and an after count, the Prism environment checks the condition first.

When tracing source lines, the Prism environment steps into procedure calls if they have source associated with them. It “nexts” over them if they do not have source. See “next” on page 55 for more information.

`silent` allows you to create the event and gives the event the same attribute as if you had specified `y` in the silent field of the Event Table of the Prism graphic interface. `disabled` allows you to create the event, but the event is disabled as if you had specified `n` in the enabled field of the Event Table of the Prism graphic interface.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

Note – In the scalar mode of the Prism environment, issuing the `trace` command prints a status line followed by the source code for each source line traced. In the MP Prism environment, the `trace` command prints only status lines.

EXAMPLES

To do a trace, print the value of `a`, and execute the `where` command at every source line:

```
trace {print a; where}
```

To trace line 17 if `a` is greater than 10:

```
trace at 17 if a .gt. 10
```

To trace line 20 of file `bar`:

```
trace ”bar”:20
```

tracei

Traces machine instructions.

SYNTAX

```
tracei [var | at addr | in func] [if expression] [{cmd; cmd ...}]
[after n] [silent | disabled] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `tracei` command to trace machine instructions when the program is executed.

The first option listed in the synopsis (*var* | `at` *addr* | `in` *func*) must come first on the command line; you can specify the other options, if you include them, in any order.

var is the name of a variable. The value of the variable is displayed whenever it changes. If the variable is an array or a parallel variable, values are displayed if the value of any element changes. This form of the command slows execution considerably. You cannot specify both a variable and a location.

`at` *addr* causes a message to be displayed immediately prior to the execution of the specified address.

`in` *func* causes tracing information to be displayed only while executing inside the specified procedure or function.

`if` *expression* specifies the logical condition, if any, under which tracing is to occur. The logical condition can be any expression that evaluates to true or false. Unless combined with the `at` *addr* syntax, this form of `tracei` slows execution considerably.

`{`*cmd*`;` *cmd* ...`}` specifies the actions, if any, that are to accompany the trace. Put the actions in braces. The actions can be any Prism commands; if you include multiple commands, separate them with semicolons.

`after` *n* specifies how many times a trigger condition (for example, reaching a program location) is to occur before the trace occurs. The default is 1. If you specify both a condition and an after count, the Prism environment checks the condition first.

When tracing instructions, the Prism environment follows all procedure calls down.

`silent` allows you to create the event and gives the event the same attribute as if you had specified `y` in the silent field of the Event Table of the Prism graphic interface. `disabled` allows you to create the event, but the event is disabled as if you had specified `n` in the enabled field of the Event Table of the Prism graphic interface.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See "Psets: Processes and Threads" on page 2 for more information on pset qualifiers.

EXAMPLES

To trace the instruction at address 1000 (hex) the third time it is reached:

```
tracei 0x1000 after 3
```

To trace the instruction at address 500 (hex) if `a` is equal to 0:

```
tracei 0x500 if a == 0
```

type

Specifies the data type of a Sun™ Scalable Scientific Subroutine Library (Sun S3L) array handle, allowing the Prism environment to display and visualize S3L arrays of as many as seven dimensions.

SYNTAX

```
type datatype variable
```

DESCRIPTION

Use the `type` command to notify the Prism environment that a specified program variable is an S3L array descriptor, and to specify the specific basic data type of the S3L array. Basic data types are int, float, double, complex8, and complex16. Before using the `type` command, the Prism environment recognizes the array handle as a simple variable. In Fortran 77 and Fortran 90, the array handle is a variable of type integer*8. In C, the array handle is type S3L_array_t.

The basic type used in the `type` command must match the basic type of the S3L array in the program.

Once you have specified the correct data type, the Prism environment can display the S3L array using the `print` command.

EXAMPLE

```
(prism all) whatis a
integer*8  a
(prism all) type float a
"a" defined as "float a"
(prism all) whatis a
(Parallel) $float a(0:19,0:33)
(prism all) print a(0:3,0:4)
a(0:3,0:4) =
(0:3,0) 0.4861192     0.8060876     0.4792756     0.4549360
(0:3,1) 0.05794585    0.1046422     0.05787051    0.1529560
(0:3,2) 0.4907097     0.02554476    0.4807888     0.6942390
(0:3,3) 0.5493287     0.2982326     0.8591906     0.3039416
(0:3,4) 0.01880360    0.3234419     0.2168089     0.1593620
```

To visualize `a` with one of the Prism visualizers:

```
(prism all) print a on dedicated
```

To gather information on where the elements of `a` are distributed:

```
(prism all) print layout(a) on dedicated
```

To assign a value to one or more elements of `a`:

```
(prism all) assign a(2,3) = 5.0
```

unalias

Removes an alias.

SYNTAX

```
unalias name
```

DESCRIPTION

Use the `unalias` command to remove the alias with the specified name. Issue the `alias` command with no arguments to obtain a list of your current aliases.

unset

Deletes a user-set name.

SYNTAX

```
unset name
```

DESCRIPTION

Use the `unset` command to delete the setting associated with *name*. See the `set` command for a discussion of setting names for variables and expressions.

Do not use the `unset` command to unset any of the Prism internal variables (variable names beginning with `$`).

EXAMPLE

If you use the `set` command to set this abbreviation for a variable name:

```
set fred = frederick_bartholomew
```

then you can unset it as follows:

```
unset fred
```

In this example, after issuing the `unset` command, you can no longer use `fred` as an abbreviation for `frederick_bartholomew`.

unsetenv

Unsets an environment variable.

SYNTAX

```
unsetenv variable
```

DESCRIPTION

Use the `unsetenv` command to remove the specified environment variable.

Environment variables become defined or undefined in the Prism environment at the moment that `setenv` or `unsetenv` is executed. The program to be debugged inherits the Prism environment at the moment that the target program is executed. For this reason, changes to the Prism environment by `setenv` and `unsetenv` do not affect any processes that are already running.

Although the Prism environment, and any programs executed within it, inherits its environment from the shell that created it, the `setenv` and `unsetenv` commands do not affect the shell that started the Prism environment, or the Prism environment itself.

The Prism environment's `unsetenv` command is identical to its Solaris C shell counterpart. See your Solaris documentation for more information.

untearoff

Removes a button from the tear-off region.

SYNTAX

```
untearoff ”label”
```

DESCRIPTION

Use the `untearoff` command to remove a button from the tear-off region of the main window of the Prism environment. Put the button’s label in quotation marks. Case and blank spaces don’t matter, and you can omit the three dots that indicate that clicking the button displays a dialog box. If the tear-off region includes more than one button with the same label, include the name of the selection’s menu in parentheses after the label.

Changes you make to the tear-off region are saved when you leave the Prism environment.

This command is not available in the commands-only interface of the Prism environment.

EXAMPLES

To remove the Load button from the tear-off region:

```
untearoff ”load”
```

To remove the button that executes the Print selection from the Events menu:

```
untearoff ”print (events)”
```

up

Moves the symbol lookup context up one level in the call stack.

SYNTAX

```
up [count]
```

DESCRIPTION

Use the up command to move the current function up the call stack *count* levels (that is, away from the current stopping point in the program toward the main procedure). If you omit *count*, the default is one level.

Issuing up repositions the source window at the new current function.

After a series of up commands, the Prism environment attempts to preserve the level when the current process changes.

use

Adds a directory to the list of directories to be searched when looking for source files.

SYNTAX

```
use [directory]
```

DESCRIPTION

Issue the `use` command to add *directory* to the front of the list of directories the Prism environment is to search when looking for source files. This is useful if you have moved a source file since compiling the program, or if for some other reason the Prism environment can’t find a file. If you do not specify a directory, `use` prints the current list.

No matter what the contents of the directory list is, the Prism environment always searches first in the directory in which the program was compiled.

varsave

Save the value of a variable or expression to a file.

SYNTAX

```
varsave "filename" expression
```

DESCRIPTION

Use the `varsave` command to save the value of the variable or expression specified by *expression* to the file *filename*. You can subsequently restore the values in *filename* via the `varfile` intrinsic (except in the MP Prism environment) and compare them with another version of the variable or expression via the Diff or Diff With selection from a visualizer's Options menu.

EXAMPLES

To save the value of the variable `alpha` in the file `alpha.data` (in your current working directory within the Prism environment):

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

To save the results of the expression `alpha*2` in the file with the path name `/u/kathy/alpha2.data`:

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

wait

Waits for a process or processes to stop execution. The `wait` command is available only in the MP Prism environment.

SYNTAX

```
wait [every | any] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `wait` command to tell the Prism environment to wait for the specified process or processes to stop execution before accepting commands that affect other processes (for example, commands that start or stop execution). A process is considered to have stopped if it has entered the `done`, `break`, `interrupted`, or `error` state.

This command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

Use the form `wait` or `wait every` to wait for every process in the pset to stop execution. The default is `wait every`.

Use the form `wait any` to wait for any running process in the pset to stop execution.

You can end the wait by doing one of the following:

- Type Ctrl-C; this does not affect processes that are running.
- Choose the Interrupt selection from the Execute menu (in the graphical interface of the Prism environment); this stops processes that are running, as well as ending the wait.

  You cannot use an unbounded (dynamic) pset as the context for a `wait every` command. For information about unbounded psets, see the *Prism User’s Guide.*

whatis

Displays the declaration of a name.

SYNTAX

```
whatis [struct | class | enum | union] name
[pset pset_name | pset_definition]
```

DESCRIPTION

Use the `whatis` command to display information about a specified name in the program.

The Prism environment displays type information using the syntax of the source language (the language of the definition, not the declaration). In programs written in a mixture of Fortran and C, the Prism environment displays each declaration in the appropriate language.

When a keyword (`struct`, `class`, `enum`, or `union`) is present, the Prism environment treats *name* as a type name. The keyword resolves ambiguities where there are types and variables with the same name.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

EXAMPLE

To display information about `Name` (by default, the declaration is assumed to be the declaration of a variable, not a type):

```
(prism) whatis Name
Name *Name;
```

Use the struct keyword to ask about a type. In this example there are two types spelled Name. One Name is a typedef:

```
(prism) whatis struct Name
More than one identifier ’Name’.
Select one of the following names:
0) Cancel
1) ‘a.out‘whatis.c‘struct Name
2) ‘a.out‘whatis.c‘Name
> 2
typedef struct Name  Name;
```

The other Name is a struct:

```
(prism) whatis struct Name
More than one identifier ’Name’.
Select one of the following names:
0) Cancel
1) ‘a.out‘whatis.c‘struct Name
2) ‘a.out‘whatis.c‘Name
> 1
struct Name {
char last[50];
char first[40];
char middle;
struct Name *next;
};
```

when

Sets a breakpoint. The `when` command is similar to the `stop` command.

SYNTAX

```
when [var | at line | in func | stopped] [if expr] [{cmd [; cmd …]}]
[after n]
```

DESCRIPTION

Use the `when` command to set a breakpoint at which the program is to stop execution.

The first option listed in the synopsis (*var* | `at` *line* | `in` *func* | `stopped`) must come first on the command line; you can specify the other options, if you include them, in any order.

var is the name of a variable. Execution stops whenever the value of the variable changes. If the variable is an array or a parallel variable, execution stops when the value of any element changes. This form of the command slows execution considerably. You cannot specify both a variable and a location.

`at` *line* stops execution when the specified line is reached. If the line is not in the current file, use the form `"`*filename*`":`*line_number*, placing the file name between quotation marks.

`in` *func* stops execution when the specified procedure or function is reached.

`stopped` specifies that the actions associated with the command occur every time the program stops execution.

`if` *expr* specifies the logical condition, if any, under which execution is to stop. The logical condition can be any expression that evaluates to true or false. Unless combined with the `at` *line* syntax, this form of `when` slows execution considerably.

{*cmd*`;` *cmd* …} specifies the actions, if any, that are to accompany the breakpoint. Put the actions in curly braces. The actions can be any Prism commands; if you include multiple commands, separate them with semicolons.

`after` *n* specifies how many times a location is to be reached before the breakpoint occurs. The default is 1. If you specify both a condition and an after count, the Prism environment checks the condition first.

EXAMPLE

To print the value of `a` in a dedicated window whenever execution stops:

```
when stopped {print a on dedicated}
```

where

Displays the call stack.

SYNTAX

```
where [count] [pset pset_name | pset_definition]
```

DESCRIPTION

Use the `where` command to print out a list of the active procedures and functions on the call stack. With no argument, `where` displays the entire list. If you specify *count*, `where` displays the specified number of functions.

The `where` command reports all active stack frames that have a stack pointer. The `where` command does not report routines that have no frame pointer and routines that have been inlined.

You can use the default alias `t` for this command.

In the graphical mode of the Prism environment, the command `where on dedicated` displays a `Where` graph, a dynamic call graph of the program.

When issued in the MP Prism environment, this command can take a pset qualifier. If used with a qualifier, it applies to the pset you specify. If used without a qualifier, it applies to the current pset. See “Psets: Processes and Threads” on page 2 for more information on pset qualifiers.

whereis

Displays the full qualification of all the symbols matching a given identifier.

SYNTAX

```
whereis identifier
```

DESCRIPTION

Use the `whereis` command to display a list of the fully qualified names of all symbols whose name matches *identifier*. The symbol class (for example, `procedure` or `variable`) is also listed.

Use the `whatis` command on the fully qualified names to determine their types.

EXAMPLE

Issuing this command:

```
whereis x
```

might produce this response:

```
variable: ‘a.out‘foo.c‘foo‘x
```

which

Displays the fully qualified name of an identifier.

SYNTAX

```
which identifier
```

DESCRIPTION

Use the `which` command to display the fully qualified name of *identifier*. This indicates which (of several possible) variables or procedures by the name *identifier* the Prism environment would use at this point in the program (for example, in an expression). The fully qualified name includes the file name or function name with which the identifier is associated.

Use the `whatis` command on the fully qualified names to determine their types.

For more information on fully qualified names, see the *Prism User's Guide*.

APPENDIX **A**

Prism `man` Page

`prism`

Enter the Prism environment.

Syntax

To start in multi-process program mode:

```
mprun [mprun-options] prism
[ –C | –CX ]
[ Xoption ...]
[ < infile ] [ > outfile ]
[ –install ] [ –threads ] [ –nothreads ]
[ program-name ]
```

To attach to a running multi-process job:

```
prism [ program-name | - jid | - jid-list ]
[ –C | –CX ]
[ Xoption ...]
[ < infile ] [ > outfile ]
[ –install ] [ –threads ] [ –nothreads ]
```

To start in serial program mode:

```
prism program-name
[ –C | –CX ]
[ Xoption ...]
[ < infile ] [ > outfile ]
[ –install ] [ –threads ] [ –nothreads ]
```

To attach to a running serial program:

```
prism [ program-name | - pid ]
[ –C | –CX ]
[ Xoption ...] [ core-file ]
[ < infile ] [ > outfile ]
[ –install ] [ –threads ] [ –nothreads ]
```

Ways To Start Prism

The Prism environment is an X-based graphical environment that enables you to develop, execute, debug, and visualize data for both serial and parallel programs. Youcan start the Prism environment independently and then run a set of programs through it, or you can attach it to a running pro gram. As a result, you can start the Prism environment in four different ways:

- “Starting In Multi-Process Program Mode” on page 118
- “Attaching To A Running Multi-Process Program” on page 119
- “Starting in Serial Program Mode” on page 119
- “Attaching To A Running Serial Program” on page 119

Starting In Multi-Process Program Mode

When you start the Prism environment in parallel program mode, you are essentially starting it along with a parallel processing environment. That parallel processing environment uses the number of processes, resource manager, and any other characteristics specified by the **mprun**(1) command. The syntax is:

mprun *mprun-options* **prism** *prism-options*

For example:

```
% mprun -np 4 -x lsf prism
```

The syntax shown above starts a multi-processing environment with four processes under the control of the LSF resource manager, a host prism process, and four node prism processes, one for each of the four processes reserved by `mprun`(1).

Attaching To A Running Multi-Process Program

You can attach the Prism environment to a parallel program that is already running. One instance of the prism environment is created for each process in the program, plus one for the host. All you need to provide is the job name or job names to which you want the Prism environment attached. The syntax for attaching prism to a running parallel program is:

```
prism [ program-name | -jid | -jid-list ] prism-options
```

The Prism environment loads the jobs you specify into the Prism environment. The processes are interrupted, and you can then work with the program in the Prism environment as you normally would.

Starting in Serial Program Mode

You can start the Prism environment for use with serial (scalar) programs, by using this syntax:

```
prism program-name prism-options
```

Attaching To A Running Serial Program

You can attach the Prism environment to a running serial program by using this syntax:

```
prism [ program-name | - pid ] prism-options
```

When attaching to a running serial program in this manner, the Prism environment must be started on the same node on which the process is running.

Integrating the Prism Environment With Resource Managers

As described in mprun(1), you can start a parallel job from within several different resource managers, either interactively or through a script. Using either method, you must first enter the resource manager environment before you can start the Prism environment.

Once you have entered the resource manager environment, start the Prism environment either within mprun or after, using this syntax:

```
mprun mprun-options prism prism-options
```

The mprun command launches a node Prism process for each rank, and a host prism process on the same node as rank 0. The Prism processes running on the ranks are under the control of the resource manager, but the host Prism process is not. Following are examples of the Prism environment being used by the resource managers to debug the program a.out in interactive mode. For more information, see the appropriate resource manager manpage:

An interactive LSF session - see the **lsf_cre**(1) manpage:

```
% bsub -Is -n 16 qshort csh
LSF csh> mprun -x lsf prism -C a.out
prism> run
prism> quit
LSF csh>
```

An interactive PBS session - see the **pbs_cre**(1) manpage:

```
% qsub -l nodes=16:ppn=1 -I
PBS csh> mprun -x pbs prism -C a.out
prism> run
prism> quit
PBS csh>
```

An interactive SGE session - see the **sge_cre**(1) manpage:

```
% qsh -pe cre 16
SGE csh> mprun -x sge prism a.out
prism> run
prism> quit
SGE csh>
```

You cannot attach the Prism environment to an individual job through the batch system because no resources would be available for it. Instead, attach it using this syntax:

```
% prism a.out sge.100
```

The example above attaches the Prism environment to the program a.out with jobid sge.100. To find a program’s jobid, use mpps(1).

Notes

You must execute the **prism** command from a terminal or workstation running the X Window System (unless you specify the **-C** option).

If started in GUI mode and issued without the name of an executable program, **prism** displays the main window of the Prism environment, with no program loaded.

If you specify *core-file*, the Prism environment associates that core file with the program you load. Within the Prism environment, you can then examine the stack and display the values of variables at the point at which core was dumped.

If you specify *infile*, **prism** reads and executes commands from the specified file upon startup. Note that specifying an infile redirects standard input (stdin), blocking subsequent user input to the Prism environment. If you specify *outfile,* the Prism environment logs all its input and output to this file. This includes commands from *infile* and commands typed on the command line within the Prism environment.

If you specify **-install**, **prism** uses a private colormap at startup. If the **-install** option is not used, **prism** uses the default colormap and might run out of color resources.

If you specify **-threads**, **prism** operates on programs that have not been linked to the **libmpi_mt** library as threaded programs. For example, you might want to use this option if your program uses threads in its I/O or graphic user interface.

If you specify **-nothreads**, **prism** treats multithreaded programs as though they are unthreaded. This allows you to debug multithreaded programs using only the main thread. For example, you might want to use this option if your program generates threads automatically (by making library calls that have threaded implementations).

If there is a **.prisminit** file in your current working directory, **prism** executes the commands in it upon startup. If **.prisminit** isn't in your current working directory, the Prism environment looks for it in your home directory. If it isn't in either place, the Prism environment starts up without executing a **.prisminit** file.

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

Options

`-C`
Start the Prism environment for commands-only execution. The Prism environment displays a prompt from which you can issue any Prism commands. If you use this option, you do not need an X terminal or workstation.

`-CX`
Start the mode of the Prism environment that uses commands-only execution (like **-C**), but in which the output of certain Prism commands can be sent to X windows.

`-install`
Use a private colormap at startup.

`-threads`

View programs that have not been linked with **libmpi_mt** as threaded programs.

`-nothreads`

View multithreaded programs as though they were unthreaded.

Xoption

Apply the X toolkit option. The **prism** command accepts all standard X toolkit options. However, the **-font**, **-title**, and **-rv** options have no effect, and the **-bg** option is overridden in part by the setting of the Prism.textBgColor resource. X toolkit options are meaningless, of course, if you use **-C** to run Prism in commands-only mode.

Passing Command Line Options to Secondary Sessions

When debugging programs that make calls to `MPI_Comm_spawn()` or `MPI_Comm_spawn_multiple()`, the Prism environment creates special node Prism processes to debug the processes created by the spawn calls. These special node Prism processes are sometimes called *secondary* Prism sessions.

Secondary Prism sessions acquire some, but not all, options that you have set when you launch the primary Prism session. The acquisition status of Prism command line options is described below:

Command Option Set In Primary Prism Session	**Acquired by Secondary Prism Session**
`-C	-CX`	Yes
X*option*	Yes
-pid | *-jid* | *-jid-list*	Yes
core-file | *pid* | *jid-list*	No
< *infile*	No
> *outfile*	No
`-install`	Yes
`-threads	-nothreads`	Yes
-	No

Files

`.prisminit`	Prism initialization file
`.prism_defaults`	Prism defaults file

Identification

Prism Version 7.0.

See Also

`mprun`(1), `mpps`(1), `sge_cre`(1), `lsf_cre`(1), `pbs_cre`(1)
Prism User's Guide, Prism Reference Manual

APPENDIX **B**

Debugger Command Comparison

Prism Equivalents for Common GDB and dbx Commands

The following tables list approximately equivalent Prism commands for some common dbx and GNU Debugging (GDB) commands.

TABLE B-1 Breakpoint and Watchpoint Commands

GDB	dbx	Prism
`break` *line*	`stop at` *line*	`stop at` *line*
`break` *func*	`stop in` *func*	`stop in` *func*
`break *`*addr*	`stopi at` *addr*	`stopi {`*addr*`}`
`break ... if` *expr*	`stop ... -if` *expr*	`stop ... if` *expr*
`cond` *n*	`stop ... -if` *expr*	`stop ... if` *expr*
`watch` *expr*	`stop` *expr*	`stop` *expr*
`info break`	`status`	`status`
`info watch`	`status`	`status`
`clear` *fun*	`delete` *n*	`delete` *n*
`delete`	`delete all`	`delete all`
`disable` *n*	`handler -disable` *n*	`disable` *n*

TABLE B-1 Breakpoint and Watchpoint Commands *(Continued)*

GDB	dbx	Prism
`enable` *n*	`handler -enable` *n*	`enable` *n*
`ignore` *n cnt*	`handler -count` *n cnt*	`ignore`
`commands` *n*	`when ... {` *cmds*`; }`	`when ... {` *cmds*`; }`

TABLE B-2 Program Stack Commands

GDB	dbx	Prism
`backtrace` *n*	`where` *n*	`where` *n*
`info reg` *reg*	`print $`*reg*	`print $`*reg*

TABLE B-3 Execution Control Commands

GDB	dbx	Prism
`finish`	`step up`	`stepout`
`signal` *num*	`cont sig` *num*	`cont` *num*
`set` *var=expr*	`assign` *var=expr*	`assign` *var=expr*

TABLE B-4 Display Address Commands

GDB	dbx	Prism
`x/fmt` *addr*	`x` *addr*/*fmt*	*addr*`/[mode]`
`disassem` *addr*	`dis` *addr*	*addr/i*

TABLE B-5 Shell Commands

GDB	dbx	Prism
`shell` *cmd*	`sh` *cmd*	`sh` *cmd*

TABLE B-6 Signal Commands

GDB	dbx	Prism
`handle` *sig*	`stop sig` *sig*	`catch` *sig*

TABLE B-7 Debugging Target Commands

GDB	dbx	Prism
`attach` *pid*	`debug -` *pid*	`attach` *pid*
`attach` *pid*	`debug` *a.out pid*	`attach` *pid*
`exec` *file*	`debug` *file*	`load` *file*
`core` *file*	`debug` *a.out corefile*	`load` *a.out*`; core` *corefile*

TABLE B-8 Debugger Environment Commands

GDB	dbx	Prism
`dir` *name*	`pathmap` *name*	`use` *name*
`show dir`	`pathmap`	`use`

TABLE B-9 Source File Commands

GDB	dbx	Prism
`forw` *regexp*	`search` *regexp*	`/`*regexp*
`rev` *regexp*	`bsearch` *regexp*	`?`*regexp*

Index