Prism 6.0 Reference Manual

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 Fax 650 969-9131

Part No: 805-6278-10
June 1999, Revision A

Contents

Preface ix

1. Commands Reference 1

Redirecting Output 1

Using Pset Qualifiers 2

Prism Commands 3

`/, ?` 8

address `/` 9

value=base 13

`alias` 13

`assign` 14

`attach` 15

`bsubargs` 16

`call` 16

`catch` 17

`cd` 17

`cont` 18

`contw` 18

`core` 19

`cycle` 20

define pset 20
delete 23
delete pset 23
detach 23
disable 24
display 24
down 28
dump 28
edit 29
enable 30
eval pset 30
fg 31
file 31
func 32
help 33
hide 34
ignore 34
interrupt 35
kill 36
list 36
load 37
log 38
make 38
mprunargs 38
next 39
nexti 39
print 40
printenv 43

process 44
pset 44
pstatus 45
pushbutton 46
pwd 47
quit 47
reload 47
rerun 48
return 48
run 49
select 49
set 50
setenv 52
sh 53
show 53
show events 54
show pset 54
show psets 55
source 56
status 57
step 57
stepi 58
stepout 58
stop 58
stopi 60
tearoff 61
tnfcollection 62
tnfdebug 63

tnfdisable 64
tnfenable 65
tnffile 66
tnflist 67
tnfview 68
trace 70
tracei 71
type 73
unalias 74
unset 74
unsetenv 75
untearoff 76
up 77
use 77
varsave 78
wait 79
whatis 79
when 81
where 82
whereis 82
which 83

A. Prism man Page 85

prism 85
SYNTAX 85
DESCRIPTION 85
Environment-Specific Prism Descriptions 86
OPTIONS 88
FILES 89

IDENTIFICATION 89

SEE ALSO 89

B. Debugger Command Comparison 91

Prism Equivalents for Common GDB and dbx Commands 91

Index 95

Preface

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

The manual is intended for application programmers developing serial or parallel programs that are to run on a Sun™ HPC System. We assume you know the basics of developing and debugging programs, as well as the basics of the system on which you will be using Prism software. Some familiarity with the UNIX® 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 may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

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

Typographic Conventions

TABLE P–1 Typographic Conventions

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

Shell Prompts

TABLE P–2 Shell Prompts

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

Related Documentation

TABLE P–3 Related Documentation

Application	Title	Part Number
All	*Sun HPC ClusterTools 3.0 System Administrator's Guide: With LSF*	805-6280-10
All	*Sun MPI 4.0 User's Guide: With LSF*	805-7230-10
All	*Sun HPC ClusterTools 3.0* System Administrator's Guide: With CRE	806-0295-10
All	*Sun MPI 4.0 User's Guide: With CRE*	806-0296-10
All	*Sun HPC ClusterTools 3.0 Product Notes*	805-6262-10
Sun MPI Programming	*Sun MPI 4.0 Programming and Reference Guide*	805-6269-10
Prism	*Prism 6.0 User's Guide*	805-6277-10
S3L	*Sun S3L 3.0 Programming and Reference Guide*	805-6275-10

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

docfeedback@sun.com

Please include the part number of your document in the subject line of your email.

CHAPTER 1

Commands Reference

This reference manual gives, in alphabetical order, the syntax and reference description of every Prism command. 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 1–2 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 a `where` command into the file `where.output` in your current working directory within Prism.

You can also redirect output of a command to a window by using the syntax *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, Prism creates a separate window for the new output. The time each window was created is shown in its title. Snapshot windows let you save and compare outputs.
- You can also make up your own name for the window. 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`.

Using Pset Qualifiers

In Message Passing Prism (MP Prism), certain commands can take a set of processors, called a *pset*, as a qualifier; they are listed in Table 1–1. The format is

command `pset` *set_name* | *set_definition*

where *set_name* is the name of a pset and *set_definition* is the definition of an unnamed pset; see the `define pset` command for a discussion of how to define a pset.

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

The commands listed in Table 1–1 can take a pset qualifier.

TABLE 1–1 Commands Taking a Pset Qualifier

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

Prism Commands

Table 1–2 lists all the Prism commands and provides brief descriptions in alphabetical order. It is followed by the complete command reference, also in alphabetical order.

TABLE 1–2 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.

TABLE 1–2 Prism Commands *(continued)*

Command	Use
`attach`	Attaches to a running process or task.
`bsubargs`	Specifies `bsub` options to use in executing multiprocess programs.
`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 only).
`core`	Associates a core file with an executable program (not available in MP Prism).
`cycle`	Makes the next member of the `cycle` pset the current set (MP Prism only).
`define pset`	Creates a named pset (MP Prism only).
`delete`	Removes one or more events from the event list.
`delete pset`	Deletes a user-defined pset (MP Prism only).
`detach`	Detaches from a running process or task.
`disable`	Disables an event.
`display`	Displays the values of one or more expressions or variables.
`down`	Moves the symbol-lookup context down one level.
`dump`	Prints the names and values of local variables.
`edit`	Calls up an editor.

TABLE 1–2 Prism Commands *(continued)*

Command	Use
`enable`	Enables a previously disabled event.
`eval pset`	Updates the membership of a variable pset (MP Prism only).
`fg`	Runs the executable program in the foreground (MP Prism 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 only).
`kill`	Kills a process or task 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.
`make`	Executes the make utility.
`mprunargs`	Specifies `mprun` options to use in executing multiprocess programs.
`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.

TABLE 1–2 Prism Commands *(continued)*

Command	Use
`process`	Sets or displays the current process of the current pset (MP Prism only).
`pset`	Sets or displays the current pset (MP Prism only).
`pstatus`	Displays the execution status of processes (MP Prism only).
`pushbutton`	Adds a Prism command to the tear-off region (not available in commands-only Prism).
`pwd`	Displays the current working directory.
`quit`	Leaves Prism.
`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 only).
`show psets`	Displays information about all psets. (MP Prism only.)

TABLE 1–2 Prism Commands *(continued)*

Command	Use
`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.
`tearoff`	Adds a menu selection to the tear-off region (not available in commands-only Prism).
`tnfcollection`	Toggles the TNF collection process on or off.
`tnfdebug`	Directs probe information to `stderr` rather than the trace file.
`tnfdisable`	Turns off the tracing activity associated with the specified `TNF` probe.
`tnfenable`	Turns on the tracing activity associated with the specified `TNF` probe.
`tnffile`	Specifies the name of the final `TNF` output file.
`tnflist`	Lists the available `TNF` probes in the loaded program.
`tnfview`	Invokes the `TNF` viewer to display a trace file.
`trace`	Traces program execution.
`tracei`	Traces instructions.
`type`	Specifies the data type of an S3L array handle, allowing Prism to display and visualize the S3L array.

TABLE 1–2 Prism Commands *(continued)*

Command	Use
`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 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 Prism chooses for an identifier.

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 36.

`/` or `?` with no argument searches 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, Prism displays the message

```
No match.
```

in the history region.

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

Prints the contents of the specified memory address.

SYNTAX

address, *address*/[*mode*] [pset *set_name* | *set_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, Prism prints the contents of memory starting at the first address and continuing to the second address. If you specify a *count*, Prism prints *count* locations, starting from the address you specify.

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

Specify a symbolic address by preceding the name with an &. 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 1–4 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, Prism uses the previous mode that you specified. The initial mode is X. Supported modes are listed below.

TABLE 1–3 Mode Argument

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 1–4 Sun UltraSPARC Registers

Name	Register
`$g0--$g7`	Global registers (64 bits)
`$o0--$o7`	Output registers (64 bits)
`$l0--$l7`	Local registers

TABLE 1–4 Sun UltraSPARC Registers *(continued)*

Name	Register
`$i0--$i7`	Input registers
`$psr`	Processor state register
`$pc`	Program counter
`$npc`	Next program counter
`$y`	Y register
`$wim`	Window invalid mask
`$tbr`	Trap base register
`$f0--$f31`	Floating-point registers
`$fsr`	Floating status register (64 bits)
`$f0f1--$f62f63`	Floating-point registers
`$xg0--$xg7`	Upper 32 bits of `$g0--$g7` (SPARC V8 plus only, or higher)
`$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 MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

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). Prism prints the converted value in the command window.

EXAMPLES

`0x100=D` 256
`256=X` 0x100
`0x100=O` 0400
`0400=X` 0x100

Sets up an alias for a command or a 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 a string. When commands are processed, Prism first checks if the word is an alias for either a

command or a string. If it is an alias, Prism 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, you can issue the command

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

You could then issue the command `b(12)`, which Prism expands to

```
stop at 12
```

Prism 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.

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

SYNTAX

`assign` *lval* = *expression* [`pset` *set_name* | *set_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. Prism performs the proper type coercions if the right-hand side does not have the same type as the left-hand side.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

EXAMPLES

This example assigns the value 1 to `x`:

```
assign x = 1
```

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

This example adds 2 to each element of `array2` and assigns these values to `array1`:

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

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

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*. To work on a process or job within Prism, you must have previously loaded its executable program (the program’s object file).

`attach` takes control of the process or task and interrupts it. Once attached, you can issue Prism commands.

You can `attach` through the command line. For example, attaching to program `foo`, you would specify the program name and the process ID (pid):

% **`prism foo`** *pid*

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

Specifies `bsub` options to use when executing multiprocess programs.

SYNTAX

```
bsubargs [option | off]
```

DESCRIPTION

Use the `bsubargs` command to specify `bsub` options that are to be used in subsequently executing multiprocess programs within Prism. Options you specify via `bsubargs` supersede the entire list of options set via the Prism command line.

You must reset every one of your bsub options every time you issue the bsubargs command.

Use the `off` option to remove existing `bsub` options.

Issue `bsubargs` with no options to display the current `bsub` options.

Calls a procedure or function.

SYNTAX

`call` *procedure* (*parameters*) [`pset` *set_name* | *set_definition*]

DESCRIPTION

Use the `call` command to call the specified procedure or function at the current stopping point in the program. Prism 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 MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

Tells Prism to catch the specified Solaris signal.

SYNTAX

```
catch [number | signal_name] [pset set_name | set_definition]
```

DESCRIPTION

Prism can catch Solaris signals before they are sent to the program. Use the `catch` command to tell Prism to catch the signal you specify. When Prism receives the signal, execution stops, and Prism 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 Prism were not present.

By default, Prism catches all signals except `SIGHUP`, `SIGEMT`, `SIGKILL`, `SIGALRM`, `SIGTSTP`, `SIGCONT`, `SIGCHLD`, and `SIGWINCH`; you can 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 Prism is to catch.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

Changes the current working directory.

SYNTAX

```
cd [directory]
```

DESCRIPTION

Use the `cd` command to change your current working directory in Prism 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.

Continues execution.

SYNTAX

```
cont [number | signal_name] [pset set_name | set_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, by name or 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 MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

In MP Prism, continues execution and then waits for the members of the current pset to finish execution.

SYNTAX

`contw` [*number* | *signal_name*] [`pset` *set_name* | *set_definition*]

DESCRIPTION

The `contw` command is available only in MP Prism. It 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, by name or 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

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 Prism. Prism 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 Prism — 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 MP Prism. Instead, you must specify the name of the process core file from the Prism command line. For example,

```
prism a.out core
```

See the *Prism 6.0 User's Guide* for more information.

In MP Prism, makes the next member of the `cycle` pset the current set.

SYNTAX

```
cycle
```

DESCRIPTION

The `cycle` command is available only in MP Prism.

Use the `cycle` command in MP Prism 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. Issuing the `cycle` command advances the current process in the `cycle` pset to be the next member in the set and makes the current pset consist of only this process. This provides a convenient way of looking at each individual process 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)
```

In MP Prism, creates a named pset.

SYNTAX

```
define pset name definition
```

DESCRIPTION

The `define pset` command is available only in MP Prism.

Use the `define pset` command to create a pset with the name and definition you specify.

For the *name* argument, you can use any name except the predefined names `all`, `running`, `error`, `interrupted`, `break`, `stopped`, `done`, `current`, and `cycle`. The name must begin with a letter; it can contain any alphanumeric character, plus the dollar sign and underscore.

For the definition, you can specify any of the following:

- *An individual process number*.
- *The name of a pset*. The new pset will have the same definition as the existing set.
- *A list of process numbers*. Separate the numbers with commas. Use a colon between two process numbers to indicate a range. Use a second colon to indicate the stride to be used within this range.
- *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 *.

Prism 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 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 isn't active in a process, Prism 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 a process that Prism tries to evaluate is running, the evaluation fails and the command is not executed. To avoid this, you can 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

This command creates a pset `foo` containing the processes 0, 4, and 7:

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

This command defines a pset `odd` containing the odd-numbered processes between 1 and 31:

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

This command defines a pset `quux` that contains processes that are members of either pset `foo` or pset `bar`:

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

This command defines 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 }
```

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. You obtain these numbers 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.

In MP Prism, deletes a user-defined pset.

SYNTAX

```
delete pset set_name
```

DESCRIPTION

The `delete pset` command is available only in MP Prism.

Use the `delete pset` command to delete the pset *set_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.

Detaches a process or task running within Prism.

SYNTAX

```
detach
```

DESCRIPTION

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

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

Use the `kill` command to terminate the process or task to which prism is attached.

Disables one or more events.

SYNTAX

`disable` *event_ID [event_ID …]*

DESCRIPTION

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

Displays the values of one or more variables or expressions.

SYNTAX

`[where` *(expression*`)] display[`*/radix*`]` *expression* `[`, *expression* …`]`
`[pset` *set_name* | *set_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 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 1–5.

TABLE 1–5 Radix Settings

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.

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.

Displaying to a window other than the command window creates a visualizer for the data.

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 below.

TABLE 1–6 UltraSPARC Registers

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)

TABLE 1–6 UltraSPARC Registers *(continued)*

Name	Register
`$f0f1--$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 MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

EXAMPLES

```
display sum(foo)
```

displays the sum of the elements of the array `foo`.

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

displays (in a dedicated window) the values of `foo` that are not equal to 0.

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.

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*, Prism 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 MP Prism.

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
    # 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 .
    # 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()
```

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; Prism positions you 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, Prism uses the setting of your `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.

Enables previously disabled events.

SYNTAX

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

DESCRIPTION

Use the `enable` command to enable the event with corresponding ID numbers. Use the `show events` command to list events along with their IDs and to determine which events are disabled. Use the `disable` command to disable events. Disabled events are kept in the event list, but they no longer affect execution. Enabling events via the `enable` command allows them to affect execution once again.

In MP Prism, updates the membership of a variable pset.

SYNTAX

```
eval pset set_name
```

DESCRIPTION

The `eval pset` command is available only in MP Prism.

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,

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

defines a pset `foo` that consists of all stopped processes in which `x` is active and is greater than zero. 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 remains what it was before you issued the `eval pset` command.

In MP Prism, runs the executable program in the foreground.

SYNTAX

```
fg
```

DESCRIPTION

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

Use the `fg` command to bring your executable program into the foreground. When you execute a message-passing program in the commands-only version of MP Prism, the program starts up in the background. You would need to bring it 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.

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 Prism uses in identifying variables.

When `list` is invoked with an absolute file name, Prism searches for *filename* as specified. When invoked with a relative file name, Prism searches first in the directory where *filename* was compiled. Then, if *filename* is not found, Prism attempts to locate *filename* using the current-use list. For further information, see "`use`" on page 77.

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

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 Prism 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.

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.

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.

The `hide` command is not meaningful in commands-only Prism.

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
```

Tells Prism to ignore the specified Solaris signal.

SYNTAX

```
ignore [number | signal_name] [pset set_name | set_definition]
```

DESCRIPTION

Prism can catch Solaris signals before they are sent to the program. Use the `ignore` command to tell Prism to ignore the specified signal. If the signal is ignored, Prism sends it to the program and allows the program to continue running without interruption; the program can then do what it wants with the signal. By default, Prism catches all signals except `SIGHUP`, `SIGEMT`, `SIGKILL`, `SIGALRM`, `SIGTSTP`, `SIGCONT`, `SIGCHLD`, and `SIGWINCH`; you can 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 Prism ignores.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

In MP Prism, interrupts execution on processes.

SYNTAX

```
interrupt [pset set_name | set_definition]
```

DESCRIPTION

The `interrupt` command is available only in MP Prism.

Use the `interrupt` command to interrupt execution on processes. Without a pset qualifier, `interrupt` interrupts execution on the processes in the current pset. With a pset qualifier, `interrupt` interrupts execution on the processes in the set you specify.

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

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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

EXAMPLES

This command interrupts the execution of the members of the predefined pset `running`:

```
interrupt pset running
```

This command interrupts the execution of process 5:

```
interrupt pset 5
```

Kills a process or task running within Prism.

SYNOPSIS

```
kill
```

DESCRIPTION

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

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 Prism 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 commands-only Prism, `list` changes the current source line (but not the current execution line) to the last line displayed or updates the current line pointer in the source display. Subsequent list commands (or search commands, for further information, see "`/, ?`" on page 8) begin from the new current line.

In graphic Prism, 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` for this command.

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

Loads an executable program into Prism.

SYNTAX

`load` *filename*

DESCRIPTION

The `load` command loads the file specified by *filename* into Prism. 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 Prism.

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 Prism's responses.

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

Use `log off` to turn off logging.

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, Prism 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.

Specifies `mprun` options to use when executing multiprocess programs.

SYNTAX

```
mprunargs [options | off]
```

DESCRIPTION

Use the `mprunargs` command to specify `mprun` options that are to be used in subsequently executing multiprocess programs within Prism. Options you specify via `mprunargs` supersede the corresponding options set via the Prism command line.

Use the `off` option to remove existing `mprun` options.

Issue `mprunargs` with no options to display the current `mprun` options.

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

SYNTAX

```
next [n] [pset set_name | set_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.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

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

SYNTAX

```
nexti [n] [pset set_name | set_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.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

Prints the values of one or more variables or expressions.

SYNTAX

```
[where (expression)] print[/radix] expression [, expression …]
[pset set_name | set_definition]
```

DESCRIPTION

Use the `print` command to print 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

TABLE 1–7 Radix Settings

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

described in Table 1–7.

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. 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.

Printing to a window other than the command window creates a visualizer for the data.

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 1–8 UltraSPARC Registers

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--$f62f63`	Floating-point registers, printable only as doubles
`$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)

TABLE 1–8 UltraSPARC Registers *(continued)*

Name	Register
`$fp`	Frame pointer (synonym for `$i6`)
`$sp`	Stack pointer (synonym for `$o6`)

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

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

EXAMPLES

```
print maxval(a)
```

prints the maximum value of the array `a`.

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

prints (in a dedicated window) the values of `a` that are greater than 3.

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.

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

In MP Prism, sets or displays the current process of the current pset.

SYNTAX

```
process [process_number]
```

DESCRIPTION

The `process` command is available only in MP Prism.

Use the `process` command to change the current process of the current pset to *process_number*. If you omit the argument, `process` displays the current process of the current pset.

You cannot include this command in event actions.

In MP Prism, sets or displays the current pset.

SYNTAX

```
pset [set_name | set_definition]
```

DESCRIPTION

The `pset` command is available only in MP Prism.

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` for an explanation of how to define a pset.

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

With no arguments, `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 0:12:2
(prism 0:12:2) pset
The current set is defined as '0:12:2'.
The set contains processes: 0,2,4,6,8,10,12.
(prism 0:12:2) pset foo
(prism foo) pset
The current set is defined as 'foo'.
The set contains processes: 0:3.
```

In MP Prism, displays the execution status of processes.

SYNTAX

```
pstatus [set_name | set_definition]
```

DESCRIPTION

The `pstatus` command is available only in MP Prism.

Use the `pstatus` command to display the execution status of the processes in pset *set_name* or which fit the pset definition *set_definition*. See the `define pset` command for a discussion of how to define a pset. If you issue `pstatus` with no

arguments, it displays the execution status of the processes in the current pset. In its output, `pstatus` aggregates processes that have the same status.

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
```

Adds a Prism command to the tear-off region.

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 Prism.

This command is not available in commands-only Prism.

EXAMPLE

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

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

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 Prism.

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

Leaves Prism.

SYNTAX

```
quit
```

DESCRIPTION

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

Reloads the currently loaded program.

SYNTAX

```
reload
```

DESCRIPTION

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

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 Prism. 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.

Steps out to the caller of the current function.

SYNTAX

`return` [*count*] [`pset` *set_name* | *set_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 MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

Executes the currently loaded program.

SYNTAX

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

DESCRIPTION

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

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

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`.

Prism 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.

In commands-only Prism, `select` determines the programming language that Prism uses in displaying messages, events, and so on.

EXAMPLES

```
select .s
```

makes the pane containing the loaded program's assembly code the master pane.

```
select .f
```

selects the pane containing the Fortran 77 source code to be the master pane.

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 Prism; 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 know about variables you set via 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 Prism. These variables begin with a `$` so that they will not conflict with the names of user-set variables. You can change the settings of these internal variables:

- `$d_precision`, `$f_precision` — Use these variables to specify the default number of significant digits Prism prints for doubles and floating-point variables, respectively. Prism'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
```

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

- `$history` — Prism stores the maximum number of lines in the history region in this variable. When the history region reaches the maximum, Prism 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
```

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

- `$fortran_string_length` — Prism uses this value as the length of a character string when the length is not explicitly specified. The default is 10.
- `$fortran_adjust_limit` — Prism uses this value as the limit of an adjustable array. The default is 10.
- `$page_size` — This value is used only in commands-only Prism. It specifies the number of output lines Prism displays before stopping and prompting with a `more?` message. Prism obtains its default from the size of your screen. If you specify 0, Prism never displays a `more?` message.
- `$print_width` — This value is used only in commands-only Prism. It specifies the number of items to be printed on a line. The default is 1.
- `$prompt_length` — This value is used only in MP Prism. It specifies the maximum number of characters to appear in the pset part of the `(prism)` prompt. The default is 25.
- `$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.

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

Issue the `unset` command to remove a setting (except a predefined one).

Displays or sets environment variables.

SYNTAX

```
setenv [VARIABLE [setting]
]
```

DESCRIPTION

Use the `setenv` command to set an environment variable within Prism. 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` are executed. The program to be debugged by Prism 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 Prism, 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 Prism, or Prism itself.

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

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, Prism 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.

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 commands-only Prism.

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
```

Displays the event list.

SYNTAX

```
show events
```

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 `n`, the `show events` command reports only for the process number specified. If `n` 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.

In MP Prism, displays the contents of a pset.

SYNTAX

`show pset` [*set_name* | *set_definition*]

DESCRIPTION

This command is available only in MP Prism.

Use the `show pset` command to display the contents of the pset with the name *set_name* or that which is defined by *set_definition*. (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

Issue this command to display the contents of the pset `stopped`:

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

In MP Prism, displays information about all psets.

SYNTAX

```
show psets
```

DESCRIPTION

This command is available only in MP Prism.

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 graphical MP Prism, or in commands-only MP Prism 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
```

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

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 an alias for the `show events` command.

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

Executes one or more source lines.

SYNTAX

`step` [*n*] [`pset` *set_name* | *set_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.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

Executes one or more machine instructions.

SYNTAX

```
stepi [n] [pset set_name | set_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.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

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`.

Sets a breakpoint.

SYNTAX

```
stop [var | at line | in func] [if expression] [{cmd; cmd …}] [after n]
[pset set_name | set_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 `Prism` 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. 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, Prism checks the condition first.

When issued in MP Prism, 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 " Using Pset Qualifiers" on page 2 for more information on pset qualifiers.

EXAMPLES

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

stops execution the tenth time in the function `foo`, prints `a`, and executes the `where` command.

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

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

```
stop a
```

stops execution whenever the value of `a` changes.

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

stops execution the third time `a` equals 5.

Sets a breakpoint at a machine instruction.

SYNTAX

```
stopi [var | at addr | in func] [if expression] [{cmd; cmd …}] [after n]
[pset set_name | set_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.

`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, Prism checks the condition first.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

EXAMPLES

```
stopi at 0x1000
```

stops execution at address 1000 (hex).

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

stops execution at address 500 (hex) if `a` is equal to 0.

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 Prism window. 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 Prism.

This command is not available in commands-only Prism.

EXAMPLES

```
tearoff ''file''
```

puts the `File` selection in the tear-off region.

```
tearoff ''print (events)''
```

puts the Print selection from the Events menu in the tear-off region.

Turns the collection of TNF trace data on or off.

SYNTAX

```
tnfcollection [on|off]
```

DESCRIPTION

Use the `tnfcollection` command to begin or halt the collection of TNF trace data. The `tnfcollection on` command:

- Adds `/opt/SUNWhpc/lib/tnf` to your `LD_LIBRARY_PATH.`
- Establishes a default file name for the TNF data.
- Sets the minimum size for data collection buffers (128 Kbytes).
- Enables all probes.
- Turns on TNF data collection.

Note - If you prefer to control the naming of TNF data files (or to specify a larger buffer size), you can define your own TNF data file name with the `tnffile` command before issuing the Prism `run` command. However, if you specify a file name that already exists, Prism issues an error message "file already exists" and ignores the `tnffile` command.

EXAMPLE

Use the `tnfcollection` command to start or stop the collection of probe data from the loaded program. You can issue the command to activate probes located throughout your program or you can issue the command as an event action specifier, activating the collection of probe data between breakpoints. For example:

```
(prism all) tnfenable mpi_api
(prism all) stop at foo {tnfcollection on}
(prism all) stop at bar {tnfcollection off}
(prism all) cont
```

Directs probe information to `stderr` rather than the trace file.

SYNTAX

```
tnfdebug [probes …]| probe_group
```

DESCRIPTION

Use `tnfdebug` to direct probe information to `stderr`. For example, if you want to see probe information about a single probe while your program is running, you can direct Prism to display that probe's information without waiting for the final trace file to be created when your program ends.

You select probes by:

- Probe name – Defined in the TNF-instrumented Sun MPI library. You can specify multiple probes by using a list of space-separated probe names. You can also use probe names created in your own C or C++ program.
- A wildcard expression using shell pattern matching notation.
- A group name defined in the TNF-instrumented version of the Sun MPI library. Probes can belong to multiple probe groups. If you specify a probe group, `tnfdebug` directs the probe information to `stderr` from all probes belonging to that probe group. You can also use group names created in your own C or C++ program. You can specify only one probe group per `tnfdebug` command.

For further information about creating TNF probes and groups, see the `TNF_PROBE(3X)` man page. For further information about the shell pattern matching format accepted by `tnfdebug`, see the `fnmatch(5)` man page.

EXAMPLES

To direct probe information from `MPI_Barrier_start` to `stderr:`

```
(prism all) tnfdebug MPI_Barrier_start
```

Turns off the tracing activity associated with the specified TNF probe.

SYNTAX

```
tnfdisable [probes ...]| probe_group
```

DESCRIPTION

You control TNF probe tracing activity by switching the probes on or off. By default, probes start in the off state. Once turned on, the specified probes can be turned off using the `tnfdisable` command.

You can disable probes by:

- Probe name – Defined in the TNF-instrumented Sun MPI library. Specify probes by using a list of space-separated probe names. You can also use probe names created in your own C or C++ program.
- A wildcard expression using shell pattern matching notation.
- A group name defined in the TNF-instrumented version of the Sun MPI library. Probes can belong to multiple *probe_groups*. If you specify a *probe_group*, all probes belonging to that *probe_group* are disabled. You can also use group names created in your own C or C++ program. You can specify only one *probe_group* per `tnfdisable` command.

For further information about creating TNF probes and groups, see the `TNF_PROBE`(3X) man page. For further information about the shell pattern matching format accepted by `tnfdisable`, see the `fnmatch(5)` man page.

EXAMPLES

To disable all `MPI_Send`* and `MPI_Recv`* probes:

(prism all) **`tnfdisable *Send* *Recv*`**

To disable all probes:

(prism all) **`tnfdisable *`**

Turns on the tracing activity associated with the specified TNF probe.

SYNTAX

`tnfenable [`*probes* ...`]|` *probe_group*

DESCRIPTION

You control TNF probe tracing activity by switching the probes on or off. By default, probes start in the off state. Turn probes on using the `tnfenable` command.

You can enable probes by:

- Probe name – Defined in the TNF-instrumented Sun MPI library. Specify probes by using a list of space-separated probe names. You can also use probe names created in your own C or C++ program.

- A wildcard expression using shell pattern matching notation.
- A group name defined in the TNF-instrumented version of the Sun MPI library. Probes can belong to multiple *probe_groups*. If you specify a *probe_group*, all probes belonging to that *probe_group* are enabled. You can also use group names created in your own C or C++ program. You can specify only one *probe_group* per `tnfenable` command.

For further information about creating TNF probes and groups, see the `TNF_PROBE`(3X) man page. For further information about the shell pattern matching format accepted by `tnfenable`, see the `fnmatch(5)` man page.

Note - To generate trace records, `tnfcollection` must be `on`.

EXAMPLES

To enable all `MPI_Send`* and `MPI_Recv`* probes:

```
(prism all) tnfenable *Send* *Recv*
```

To enable all probes:

```
(prism all) tnfenable *
```

You can use the `tnfenable` command in conjunction with the `tnfcollection on` command to restrict the set of enabled probes. For example,

```
(prism all) tnfcollection on; tnfenable probe_group
```

accepts all of the defaults set by `tnfcollection on`, but enables only the probes in `probe_group`.

Specifies the name of the final TNF output file.

SYNTAX

```
tnffile filename [size]
```

DESCRIPTION

Use the `tnffile` command to define a target file for the trace data generated by TNF probes. By default, the `tnfcollection on` command creates a trace file with an internally generated file name and sets the trace collection data files to the minimum size (128 Kbytes). Use the `tnffile` command to override those defaults.

The *filename* argument refers to the permanent file that Prism fills with the merged data taken from each process's (temporary) output trace file. If you specify a file name that already exists, an error message "file already exists" is issued and the `tnffile` command is ignored.

When collecting TNF data, Prism creates a trace file for every process. Use the optional *size* argument to specify the size (in kilobytes) of the trace files. The default *size* is 128 Kbytes. The trace files are circular buffers—once a file has been filled, more recent trace events overwrite the oldest ones. Once the trace data collection process is complete, Prism merges all of the trace files into the output file *filename*, which can be as large as the number of processes * *size*. You can view *filename* with the `tnfview` command, or by selecting Display TNF Data from Prism's Performance menu.

Prism's MPI Performance Analysis generates large volumes of data, particularly for long-running programs or programs with high process counts. As a result the `tnfview` command may fail if insufficient disk space is available for storing TNF output data in `/usr/tmp`. To work around this restriction, limit the collection interval using the `tnfcollection` command or limit the types of events collected using the `tnfenable` command. See Chapter 6 in the *Prism 6.0 User's Guide* for more information about Prism's MPI Performance Analysis.

EXAMPLE

To create a TNF output file, `myfile.tnf`, using trace data collection files of *size* 8 Mbytes (8192 Kbytes):

```
(prism all) tnffile myfile.tnf 8192
```

Lists the available `TNF` probes in the loaded program. Requires that you issue the Prism `run` command as a precondition.

SYNTAX

```
tnflist [probes ...]| probe_group
```

DESCRIPTION

Since TNF probes can be very numerous, it can be helpful to browse the inventory of probes in the program loaded in Prism. You can list all, or a subset of the TNF probes in your program using the `tnflist` command.

You select probes by:

- Probe name – Defined in the TNF-instrumented Sun MPI library. Specify probes by using a list of space-separated probe names. You can also use probe names created in your own C or C++ program.
- A wildcard expression using shell pattern matching notation.
- A group name defined in the TNF-instrumented version of the Sun MPI library. Probes can belong to multiple *probe_groups*. If you specify a *probe_group*, all probes belonging to that *probe_group* are listed. You can also use group names created in your own C or C++ program. You can specify only one *probe_group* per `tnflist` command.

For further information about creating TNF probes and groups, see the `TNF_PROBE(3X)` man page. For further information about the shell pattern matching format accepted by `tnflist`, see the `fnmatch(5)` man page.

EXAMPLES

To list all point-to-point routine probes in the Sun MPI library:

```
(prism all) tnflist mpi_pt2pt
```

To list all probes:

```
(prism all) tnflist *
```

Invokes the TNF performance analysis program, `tnfview`, to display a trace file.

SYNTAX

```
tnfview filename
```

OPTIONS

filename – Specify a merged tracefile to load.

DESCRIPTION

`tnfview` is a CDE/Motif-based tool that allows you to load a TNF trace file, view it, and manipulate it to see what was going on in the program that recorded that trace file.

`tnfview` displays a timeline view of TNF data in the `tnfview` main window and displays scatter plot, table, and histogram views in the `tnfview` plot window.

The main window of `tnfview` displays a large timeline window showing colored glyphs for different events, arrayed on different horizontal lines that represent process ranks. Using the timeline window, you can:

- Select events with the mouse.
- Display event details in the table below the timeline graph.
- Adjust the vertical and horizontal axes, zooming and scrolling them independently for better viewing.
- Print the timeline graph.

`tnfview` displays a small panner window in the lower left corner, showing where the main window is currently focused. Using the panner window, you can:

- Select an area in the panner window to re-orient the timeline.
- Having selected an area, you can drag the area frame to another part of the panner window.

Clicking on the graph button in the main window opens the `tnfview` plot window. Using the plot window, you can:

- Select and view data derived from pairs of events called *intervals*, and groups of events (or intervals) called *datasets*.
- Manipulate the display of datasets in scatter plots, tables, and histograms.
- Print the displayed graphs.

For more information about TNF probes and how to use them in Prism, see the *Prism 6.0 User's Guide*. For more information about the TNF-instrumented Sun MPI library, see the *Sun MPI 4.0 User's Guide*.

For background information about TNF tracing, see the Solaris 2.6 Programming Utilities Guide, and the man pages `prex(1)`, `tnfdump(1)`, `tnfxtract(1)`, `TNF_DECLARE_RECORD(3X)`, `TNF_PROBE(3X)`, `libtnfctl(3X)`, `tnf_process_disable(3X)`, `tracing(3X)`, `tnf_kernel_probes(4)`, and `attributes(5)`.

Traces program execution.

SYNTAX

```
trace [var | at line | in func] [if expression] [{cmd; cmd …}] [after n]
[pset set_name | set_definition]
```

DESCRIPTION

Use the `trace` command to print tracing information when the program is executed. In a trace, Prism 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`; Prism 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 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, Prism checks the condition first.

When tracing source lines, Prism steps into procedure calls if they have source associated with them. It “nexts” over them if they do not have source.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

Note - In scalar Prism, when you issue the `trace` command Prism prints a status line followed by the source code for each source line traced. In MP Prism, the `trace` command prints only status lines.

EXAMPLES

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

does a trace, prints the value of `a`, and executes the `where` command at every source line.

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

traces line 17 if `a` is greater than 10.

```
trace ''bar'':20
```

traces line 20 of file `bar`.

Traces machine instructions.

SYNTAX

```
tracei [var | at addr | in func] [if expression] [{cmd; cmd …}]
[after n] [pset set_name | set_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 printed immediately prior to the execution of the specified address.

`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` *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 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, Prism checks the condition first.

When tracing instructions, Prism follows all procedure calls down.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

EXAMPLES

```
tracei 0x1000 after 3
```

traces the instruction at address 1000 (hex) the third time it is reached.

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

traces the instruction at address 500 (hex) if a is equal to 0.

Specifies the data type of an S3L array handle, allowing Prism to display and visualize the S3L array.

SYNTAX

type *datatype variable*

DESCRIPTION

Use the type command to notify Prism that a specified program variable is an S3L array descriptor, and that the S3L array has a specific basic data type. Basic data types are int, float, double, complex8, and complex16. Before using the type command, Prism 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, Prism 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**

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.

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 (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`.

Unsets an environment variable.

SYNTAX

```
unsetenv VARIABLE
```

DESCRIPTION

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

Environment variables become defined or undefined in the Prism environment at the moment that `setenv` or `unsetenv` are executed. The program to be debugged by Prism 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 Prism, 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 Prism, or Prism itself.

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

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 Prism window. 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 choosing 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 Prism.

This command is not available in commands-only Prism.

EXAMPLES

```
untearoff ''load''
```

removes the Load button from the tear-off region.

```
untearoff ''print (events)''
```

removes the button that executes the Print selection in the Events menu.

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 (that is, away from the current stopping point in the program toward the main procedure) *count* levels. If you omit *count*, the default is one level.

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

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 Prism 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 Prism can't find the file. If you do not specify a directory, use prints the current list.

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

Saves the values of a variable or expression to a file.

SYNTAX

varsave ''*filename*'' *expression*

DESCRIPTION

Use the varsave command to save the values 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 MP Prism) 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

```
varsave ''alpha.data'' alpha
```

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

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

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

In MP Prism, waits for a process or processes to stop execution.

SYNTAX

```
wait [every | any] [pset set_name | set_definition]
```

DESCRIPTION

The `wait` command is available only in MP Prism.

Use the `wait` command to tell Prism 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 “ Using Pset Qualifiers” 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. `wait every` is the default.

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

You can end the wait by doing one of the following

- Typing Ctrl-c; this does not affect processes that are running.
- Choosing (in graphical Prism) the Interrupt selection from the Execute menu; this stops processes that are running, as well as ending the wait.

Displays the declaration of a name.

SYNTAX

```
whatis [struct] name [pset set_name | set_definition]
```

DESCRIPTION

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

Prism 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, Prism displays each declaration in the appropriate language.

When the `struct` keyword is present, Prism treats *name* as a type name. The `struct` keyword resolves ambiguities where there are types and variables with the same name.

When issued in MP Prism, 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 " Using Pset Qualifiers" on page 2 for more information on pset qualifiers.

EXAMPLE

The default attribute is variable:

```
(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;
};
```

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, as described below.

`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 quotes.

`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 condition, if any, under which execution is to stop. The 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 braces. The actions can be any valid 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, Prism checks the condition first.

EXAMPLE

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

prints the value of `a` in a dedicated window whenever execution stops.

Displays the call stack.

SYNTAX

`where` [*count*] [pset *set_name* | *set_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 graphical MP Prism, the command `where on dedicated` displays a `Where` graph, a dynamic call graph of the program.

When issued in MP Prism, 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 “ Using Pset Qualifiers” on page 2 for more information on pset qualifiers.

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`, `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
```

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 possibly several) variables or procedures by the name *identifier* Prism 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 6.0 User's Guide*.

APPENDIX **A**

Prism `man` Page

`prism`

Enter the Prism programming environment.

To run Prism in the LSF environment:

```
prism [program-name] [–C
| –CX] [–n | –np nprocs] [–W]
[Xoption ...] [core-file | pid | jid] [< infile] [> outfile]
[-bsubargs "option [option...
]"] [–q queue]
```

To run Prism in the CRE environment:

```
prism [program-name] [–C
| –CX] [–n | –np nprocs] [–W]
[Xoption ...] [core-file | pid | jid] [< infile] [> outfile]
[-mprunargs "option [option...]"] [-c cluster] [-p partition]
```

`prism` enters Prism, an X-based graphical programming environment within which you can develop, execute, debug, and visualize data in serial and parallel programs.

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

If issued without the `-n` (or –`np`) option `prism` starts scalar Prism, the version of Prism designed for use on serial programs.

If issued with the `--n` (or `--np`) option, `prism` starts MP Prism, the version of Prism designed for use on message-passing or other multiprocess programs. The `--c` and `--p` options for Prism in the CRE environment also start MP Prism. Message-passing programs must be written in the SPMD style (that is, each process must run the same executable program).

If you specify `--n` (or `--np`), you can also include other options to specify where the processes are to run. These control where the processes are run unless overridden by a Prism option. If you don"t specify `--n` (or –`np`), `--bsubargs`, or `--mprunargs`, one copy of the program runs on your login node.

If issued without the name of an executable program, `prism` displays the main window of the Prism environment, with no program loaded. If you issue it with the name of an executable program, `prism` loads that program into Prism upon startup.

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

If you specify *pid*, `prism` loads the running process with that process ID into Prism. The process is interrupted, and you can then work with the program in Prism as you normally would. When attaching to a running serial process in this manner, Prism must be started on the same node on which the process is running. In MP Prism, you can specify *jid*, the job ID for a multiprocess program running in a Sun HPC system. To load a multiprocess program by its job ID, you must also use one of the arguments that specifies the message-passing version of Prism, MP Prism, such as the –`n` or –`np` arguments (in the CRE environment, you can also use the –`c` and –`p` arguments to specify the message-passing version of Prism).

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

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, `prism` looks for it in your home directory. If it isn"t in either place, `prism` starts up without executing a `.prisminit` file.

You can run Prism in either of the LSF or CRE environments. To determine which environment is in effect, execute the script `hpc_rte` from a shell prompt.

To Run Prism in the LSF Environment:

If you are running Prism in the LSF environment, you can specify `bsub` options that you want to apply to your message-passing program on the Prism command line as a quoted string following the –`bsubargs` option. Once you have entered Prism you can issue the `bsubargs` command on the Prism command line to specify `bsub` options. Prism stores these options, then applies them when you start up a multiprocess program. Specifying the setting of an option via the `bsubargs` command supersedes the setting of the entire list of options you have established via the `prism` command line. You must reset every one of your bsub options every time you issue the bsubargs command.

The string given to `bsubargs` should not contain the –`I`, –`Ip`, or –`n` flags, because Prism internally generates values for them, and the results will be undefined. The same considerations apply using the –`bsubargs` switch to Prism.

To remove any existing `bsub` options you have specified, issue the command `bsubargs off` This removes options you have set via the `prism` command line and the `bsubargs` command. Issuing the `bsubargs` command with no options shows the current `bsub` options.

The `bsubargs` command and the –`bsubargs` option differ in one respect. Since the –`bsubargs` option is issued at a shell prompt, you must refer to the documentation for your shell program for the specific syntax for handling any quoted string supplied as an argument to the –`bsubargs` option. The `bsubargs` command does not interact with a shell, thus the `bsubargs` command requires no additional string quoting syntax.

To Run Prism in the CRE Environment:

If you are running Prism in the CRE environment, you can specify `mprun` options that you want to apply to your message-passing program on the Prism command line as a quoted string following the –`mprunargs` option. Once you have entered Prism you can issue the `mprunargs` command on the Prism command line to specify `mprun` options. Prism stores these options, then applies them when you start up a multiprocess program. Specifying the setting of an option via the `mprunargs` command overrides the setting of the same option you have established via the `prism` command line. If it is an option that has otherwise not been specified, it is added to the existing settings.

The string given to `mprunargs` should not contain the –`I`, –`Ip`, or –`n` flags, because Prism internally generates values for them, and the results will be undefined. The same considerations apply using the –`mprunargs` switch to Prism.

To remove any existing `mprun` options you have specified, issue the command `mprunargs off` This removes options you have set via the `prism` command line and the `mprunargs` command. Issuing the `mprunargs` command with no options shows the current `mprun` options.

The mprunargs command and the –mprunargs option differ in one respect. Since the –mprunargs option is issued at a shell prompt, you must refer to the documentation for your shell program for the specific syntax for handling any quoted string supplied as an argument to the –mprunargs option. The mprunargs command does not interact with a shell, thus the mprunargs command requires no additional string quoting syntax.

Use the –c and –p options to specify a CRE cluster and partition. These options override the CRE environment variables SUNHPC_CLUSTER and SUNHPC_PART. You can use these options only when launching Prism in the CRE environment.

--C – *Commands-only execution*. Prism displays a prompt from which you can issue any Prism commands. If you use this option, you don"t need an X terminal or workstation.

--CX – *Commands-only execution with output.* Start a version of Prism that uses commands-only execution (like --C), but in which the output of certain Prism commands can be sent to X windows.

--n [or –np] *nprocs – Start* nprocs *processes of the executable program*. Without this argument, prism starts a single process. Specify 0 (zero) to indicate that you want to start one process on each available node.

--W – *Start as many processes as the* --n *argument specifies, even when the number of processes exceeds the number of processors.* The default is to launch one process per processor.

Xoption – *Apply 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.

Options for Prism in the LSF Environment:

--bsubargs "*option* [*option*]..." – *Start the executable program using the specified* bsub *options.* Using the --bsubargs option implies -n and starts MP Prism. If the bsub option itself uses quotation marks, refer to the documentation for your shell program for the syntax for handling quotes.

--q *queue – Start the executable program in the specified queue*. Without this argument, Prism starts the program in the default queue. Using the --q option implies MP Prism.

Options for Prism in the CRE Environment:

–mprunargs "*option* [*option*]..." –
Start the executable program, using the specified mprun options. –mprunargs implies –n, and starts MP Prism. If the mprun option itself uses quotation marks, refer to the documentation for your shell program for the syntax for handling quotes.

–c *cluster – Start the executable program on the specified cluster*. Using this option implies -n, and starts MP Prism. The *cluster* overrides the value of the CRE SUNHPC_CLUSTER environment vari able.

–p *partition – Start the executable program on the specified partition*. Using this option implies –n, and starts MP Prism. The *partition* overrides the value of the CRE SUNHPC_PART environment variable.

`.prisminit` – Prism initialization file.

`.prism_defaults` – Prism defaults file.

Prism Version 6.0.

`bsub`(1), `mprun`(1), `hpc_rte`(1) *Prism 6.0 User"s Guide, Prism 6.0 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 Breakpoints and Watchpoints

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`

TABLE B–1 Breakpoints and Watchpoints *(continued)*

GDB	dbx	Prism
`info watch`	`status`	`status`
`clear` *fun*	`delete` *n*	`delete` *n*
`delete`	`delete all`	`delete all`
`disable n`	`handler -disable` *n*	`disable` *n*
`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

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

TABLE B–3 Execution Control

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

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 Signals

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

TABLE B–7 Debugging Targets

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 Controlling the Debugger

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

TABLE B–9 Source Files

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

Index

Special Characters

/ command, 9
? command, 9
@, 1

A

address/ command, 10
alias command, 13
assign command, 14
attach command, 15

C

call command, 16
catch command, 17
cd command, 18
cont command, 18
contw command, 19
core command, 19
cycle command, 20

D

dedicated window, 2
define pset command, 21
delete command, 23
delete pset command, 23
detach command, 24
disable command, 24
display command, 25
down command, 28
dump command, 28

E

edit command, 29
enable command, 30
eval pset command, 30

F

fg command, 31
file command, 32
func command, 32

H

help command, 33
hide command, 34

I

ignore command, 35
interrupt command, 35

K

kill command,

L

list command, 37
load command, 37
log command, 38

M

make command, 38

mprunargs command,

N

next command, 39
nexti command, 40

O

output
    redirecting, 1

P

print command, 40
printenv command, 44
process command, 44
pset command, 45
pset qualifiers, 2, 13, 15, 17 to 19, 27, 35, 36, 39, 40, 43, 49, 59, 61, 58, 71, 72, 79, 80, 82
pstatus command, 45
pushbutton command, 46
pwd command, 47

Q

quit command, 47

R

reload command, 48
rerun command, 48
return command, 48
run command, 49

S

S3L array descriptor,
S3L array handle,
select command, 49
set command, 50
setenv command, 52
sh command, 53
show command, 53
show events command, 54
show pset command, 55
show psets command, 55
snapshot window, 2
source command, 56
status command, 57
step command, 57
stepi command, 58
stepout command, 58
stop command, 59
stopi command, 60

T

tearoff command, 62
tnfcollection command,
tnfdebug command,
tnfdisable command,
tnfenable command,
tnffile command,
tnflist command,
tnfview command,
trace command, 70
tracei command, 72
type command,

U

unset command, 75
untearoff command, 46, 76
up command, 77
use command, 78

V

value=base command, 13
varfile intrinsic, 78
varsave command,

W

wait command, 79
whatis command, 80
when command, 81
where command, 82
whereis command, 83
which command, 84