TotalView
Reference Guide

version 8.8

TotalView
TECHNOLOGIES

Book Overview

Contents

About This Book

This document is the reference guide for TotalView® Debugger. Unlike the *TotalView Users Guide* which presented GUI and CLI information together, the chapters in this book either are devoted to one interface or contain information that pertains to both.

TotalView Family Differences

This manual describes the TotalView Enterprise, TotalView Team, and TotalView Individual debuggers. Each of these allows you to use the CLI debugger as well. In all cases, TotalView Enterprise and TotalView Team have the same features. They differ in the way they are licensed. However, TotalView Individual is slightly different.

The most fundamental differences between TotalView Team and TotalView Enterprise are the way resources are shared and used. When you purchase TotalView Team, you are purchasing "tokens." These tokens represent debugging capabilities. For example, if you have 64 tokens available, 64 programmers could be active, each debugging a one-process job; or 2 programmers, each debugging a 32 process job. In contrast, a TotalView Enterprise license is based on the number of users and the number of licensed processors. You'll find more precise information on our web site.

The basic differences are:

TotalView Team & Enterprise	TotalView Individual	Comment
Execute on any licensed computer of the same architecture	Node locked.	You can execute TotalView Individual only on the computer you install it on.
Number of users is determined by license	Only one user	The TotalView Enterprise license limits the number of users. TotalView team does not.
Number of processes limited by license. No limit on threads	No more than 16 processes and threads.	
Your license determines the number of processors upon which your program can run.	A program can execute on no more than two cores.	TotalView Enterprise licenses the full capabilities of all machines upon which it runs. TotalView Team can acquire part of your machine.
Processes can execute on any computes in the same network.	Remote processes are not allowed.	Processes must execute on the installed computer.
Remote X Server connections allowed.	No remote X Server connections are allowed.	Programers cannot remotely log into a computer and then execute TotalView Individual.
Memory debugging is bundled.	No memory debugging	

How To Use This Book

The information in this book is in three parts.

- **CLI Commands**

  This part contains descriptions of all the CLI commands, the variables that you can set using the CLI, and other CLI-related information.
- **Running TotalView**

  TotalView and the TotalView Debugger Server (**tvdsvr**) can accept many command-line options. This part describes these options.
- **Platforms and Operating Systems**

  Although the way in which you use TotalView is the same from system to system and from environment to environment, these systems and environments place some constraints on what you must do, and require that you compile programs differently on the various UNIX platforms. This part describes these differences.

Conventions

The following table describes the conventions used in this book:

Convention	Meaning
[ ]	Brackets are used when describing parts of a command that are optional.
arguments	In a command description, text in italics represents information you type. Elsewhere, italics is used for emphasis.
Dark text	In a command description, **dark text** represents keywords or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.
`Example text`	In program listings, this indicates that you are seeing a program or something you'd type in response to a shell or CLI prompt. If this text is in bold, it's indicating that what you're seeing is what you'll be typing. If you're viewing this information online, example text is in color.
📌	This graphic symbol indicates that the information that follows—*which is printed in italics*—is a note. This information is an important qualifier to what you just read.

TotalView Documentation

The following table describes other TotalView documentation:

Title	Contents	Online Help	HTML	PDF	Print
Evaluating TotalView	Brochure that leads you to basic TotalView features.			✓	✓
TotalView Users Guide	Describes how to use the TotalView GUI and the CLI; this is the most used of all the TotalView books.	✓	✓	✓	
TotalView New Features	Describes new features added to TotalView.	✓	✓	✓	

Title	Contents	Online Help	HTML	PDF	Print
Debugging Memory Using TotalView	Is a combined user and reference guide describing how to find your program's memory problems.	✓	✓	✓	
TotalView Reference Guide	Contains descriptions of CLI commands, how you run TotalView, and platform-specific information.	✓	✓	✓	
TotalView QuickView	Presents what you need to know to get started using TotalView.			✓	✓
CLI Command Summary	A reference card that contains a brief description of all CLI commands and their syntax.			✓	✓
TotalView Commands	Defines all TotalView GUI commands—this is the online Help.	✓	✓	✓	
TotalView Installation Guide	Contains the procedures to install TotalView and the FLEX*lm* license manager.	✓	✓	✓	
Platforms and System Requirements	Lists the platforms upon which TotalView runs and the compilers it supports.	✓	✓	✓	

Contacting Us

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Our Internet email address for support issues is:

support@totalviewtech.com

For documentation issues, the address is:

documentation@totalviewtech.com

Our phone numbers are:

1-800-856-3766 in the United States
(+1) 508-652-7700 worldwide

If you are reporting a problem, please include the following information:

- The *version* of TotalView and the *platform* on which you are running TotalView.
- An *example* that illustrates the problem.
- A *record* of the sequence of events that led to the problem.

Part I: CLI Commands

This part of the *TotalView Reference Guide* contains five chapters that describe the TotalView Command Line Interpreter (CLI).

Chapter 1: CLI Command Summary
: Summarizes all the CLI commands.

Chapter 2: CLI Commands
: Contains detailed descriptions of the CLI commands that are found in the CLI's unqualified (top-level) namespace. These are the commands that you use day-in and day-out, and those that are most often used interactively.

Chapter 3: CLI Namespace Commands
: Contains descriptions of commands found in the **TV::** namespace. These commands are seldom used interactively, as they are most often used in scripts.

Chapter 4: Batch Debugging Using tvscript
: Shows how you create batch scripts that run TotalView unattended.

Chapter 5: TotalView Variables
: Describes all TotalView variables, including those that you use to set GUI behaviors. These variables reside in three namespaces: unqualified (top-level), **TV::** and **TV::GUI**. For the most part, you set these variables to alter TotalView behaviors.

Chapter 6: Creating Type Transformations
: If you do not wish to see all the members of a class or structure or if you would like to alter the way TotalView displays these elements, you can call a CLI routine that tells TotalView how you want it to display information. This chapter tells how to create these CLI routines.

chapter

CLI Command Summary

This chapter contains a summary of all TotalView® Debugger CLI commands. The commands are described in Chapter 2, "CLI *Commands*," on page 13 and Chapter 3, "CLI *Namespace Commands*," on page 129.

actionpoint

Gets and sets action point properties

TV::actionpoint *action* [*object-id*] [*other-args*]

alias

Creates a new user-defined pseudonym for a command

alias *alias-name defn-body*

Views previously defined aliases

alias [*alias-name*]

capture

Returns a command's output as a string

capture [**–out** | **–err** | **–both**] [**–f** *filename*] *command*

dactions

Displays information about action points

dactions [*ap-id-list*] [**–at** *source-loc*]
[**–enabled** | **–disabled**]

Saves action points to a file

dactions –save [*filename*]

Loads previously saved action points

dactions –load [*filename*]

dassign

Changes the value of a scalar variable

dassign *target value*

dattach

Brings currently executing processes under CLI control

dattach [–g *gid*] [–r *hname*]
[–replay | –no_replay]
[–ask_attach_parallel | –no_attach_parallel]
[–c *corefile-name*] [–rank *num*]
[–e] *fname pid-list*
[–parallel_attach_subset *subset_specification*]

dbarrier

Creates a barrier breakpoint at a source location

dbarrier *breakpoint-expr* [–stop_when_hit { group | process | none }]
[–stop_when_done { group | process | none }] [–pending]

Creates a barrier breakpoint at an address

dbarrier –address *addr*
[–stop_when_hit { group | process | none }]
[–stop_when_done { group | process | none }] [–pending]

dbreak

Creates a breakpoint at a source location

dbreak *breakpoint-expr* [–p | –g | –t] [[–l *lang*] –e *expr*] [–pending]

Creates a breakpoint at an address

dbreak –address *addr* [–p | –g | –t] [[–l *lang*] –e *expr*] [–pending]

dcache

Clears the remote library cache

dcache –flush

dcheckpoint

Creates a checkpoint on SGI IRIX

dcheckpoint [*after_checkpointing*] [–by *process_set*] [–no_park]
[–ask_attach_parallel | –no_attach_parallel]
[–no_preserve_ids] [–force] *checkpoint-name*

Creates a checkpoint on IBM AIX

dcheckpoint [–delete | –halt]

dcont

Continues execution and waits for execution to stop

dcont

ddelete

Deletes some action points

ddelete *action-point-list*

Deletes all action points

ddelete –a

ddetach

Detaches from the processes

ddetach

ddisable

Disables some action points

ddisable *action-point-list* [**–block** *number-list*]

Disables all action points

ddisable –a

ddlopen

Loads a shared object library

ddlopen [**–now** | **–lazy**] [**–local** | **–global**] [**–mode** *int*] *filespec*

Displays information about shared object libraries

ddlopen [**–list** *dll-ids*...]

ddown

Moves down the call stack

ddown [*num-levels*]

dec2hex

Converts a decimal number into hexadecimal

TV::dec2hex *number*

denable

Enables some action points

denable *action-point-list*

Enables all disabled action points in the current focus

denable –a

dexamine

Display memory contents

dexamine [**–column_count** *cnt*] [**–count** *cnt*] [**–data_only**]
[**–show_chars**] [**–string_length** *len*] [**–format** *fmt*]
[**–memory_info**] [**–wordsize** *size*] *variable_or_expression*

dflush

Removes the top-most suspended expression evaluation

dflush

Removes all suspended **dprint** computations

dflush –all

Removes **dprint** computations preceding and including a suspended evaluation ID

dflush *susp-eval-id*

dfocus

Changes the target of future CLI commands to this P/T set

dfocus *p/t-set*

Executes a command in this P/T set

dfocus [*p/t-set command*]

dga

Displays global array variables

dga [**–lang** *lang_type*] [*handle_or_name*] [*slice*]

dgo

Resumes execution of target processes

dgo

dgroups

Adds members to thread and process groups

dgroups –add [**–g** *gid*] [*id-list*]

Deletes groups

dgroups –delete [**–g** *gid*]

Intersects a group with a list of processes and threads

dgroups –intersect [**–g** *gid*] [*id-list*]

Prints process and thread group information

dgroups [**–list**] [*pattern-list*]

Creates a new thread or process group

dgroups –new [*thread_or_process*] [**–g** *gid*] [*id-list*]

Removes members from thread or process groups

dgroups –remove [**–g** *gid*] [*id-list*]

dhalt

Suspends execution of processes

dhalt

dheap

Shows Memory Debugger state

dheap [**–status**]

Applies a saved configuration file

dheap –apply_config { **default** | *filename* }

Shows information about a backtrace

dheap –backtrace [*subcommands*]

Compares memory states

dheap –compare *subcommands* [*optional_subcommands*]
[*process* | *filename* [*process* | *filename*]]

Enables or disables the Memory Debugger

dheap { **–enable** | **–disable** }

Enables or disables event notification

dheap –event_filter *subcommands*

Writes memory information

dheap –export *subcommands*

Specifies which filters the Memory Debugger uses

dheap –filter *subcommands*

Writes guard blocks (memory before and after an allocation)

dheap –guard [*subcommands*]

Enables and disables the retaining (hoarding) of freed memory blocks

dheap –hoard [*subcommands*]

Displays Memory Debugger information

dheap –info [–backtrace] [*start_address* [*end_address*]]

Indicates whether an address is within a deallocated block

dheap –is_dangling *address*

Locates memory leaks

dheap –leaks [–check_interior]

Enables or disables Memory Debugger event notification

dheap –[no]notify

Paints memory with a distinct pattern

dheap –paint [*subcommands*]

Enables and disables the ability to catch bounds errors and use-after-free errors retaining freed memory blocks

dheap –red_zones [*subcommands*]

Enables and disables allocation and reallocation notification

dheap –tag_alloc *subcommand start_address* [*end_address*]

Displays the Memory Debugger's version number

dheap –version

dhistory

Displays information about the state of the program as it is being replayed. If you have received a timestamp, you can go back to the line that was executing at that time.

dhistory [–info] [–get_time] [–go_time *time*] [–go_live]
[–enable] [–disable]

dhold

Holds processes

dhold –process

Holds threads

dhold –thread

dkill

Terminates execution of target processes

dkill [–remove]

dlappend

Appends list elements to a TotalView variable

dlappend *variable-name value* [...]

dlist

Displays code relative to the current list location

dlist [–n *num-lines*]

Displays code relative to a named location

dlist *breakpoint-expr* [–n *num-lines*]

Displays code relative to the current execution location

dlist –e [–n *num-lines*]

dll
Manages shared libraries

TV::dll *action* [*dll-id-list*] [–all]

dload
Loads debugging information

dload [–g *gid*] [–mpi *starter_value*] [–r *hname*]
    [–replay | –noreplay]
    [–env *variable=value*] ... [–e] *executable*
    [–parallel_attach_subset *subset_specification*]

dmstat
Displays memory use information

dmstat

dnext
Steps source lines, stepping over subroutines

dnext [–back] [*num-steps*]

dnexti
Steps machine instructions, stepping over subroutines

dnexti [–back] [*num-steps*]

dout
Executes until just after the place that called the current routine

dout [–back] [*frame-count*]

dprint
Prints the value of a variable or expression

dprint [–nowait] [–slice *slice_expr*] *variable_or_expression*

dptsets
Shows the status of processes and threads in an array of P/T expressions

dptsets [*ptset_array*] ...

drerun
Restarts processes

drerun [*cmd_arguments*] [< *infile*]
    [> [>][&] *outfile*]
    [2> [>] *errfile*]

drestart
Restarts a checkpoint on AIX

drestart [–halt] [–g *gid*] [–r *host*] [–no_same_hosts]

Restarts a checkpoint on SGI

drestart [*process-state*] [–no_unpark] [–g *gid*] [–r *host*]
    [–ask_attach_parallel | –no_attach_parallel]
    [–no_preserve_ids] *checkpoint-name*

drun
Starts or restarts processes

drun [*cmd_arguments*] [< *infile*]
    [> [>][&] *outfile*]
    [2> [>] *errfile*]

dset

Creates or changes a CLI state variable

dset *debugger-var value*

Views current CLI state variables

dset [*debugger-var*]

Sets the default for a CLI state variable

dset -set_as_default *debugger-var value*

dstatus

Shows current status of processes and threads

dstatus

dstep

Steps lines, stepping into subfunctions

dstep [**–back**] [*num-steps*]

dstepi

Steps machine instructions, stepping into subfunctions

dstepi [**–back**] [*num-steps*]

dunhold

Releases a process

dunhold –process

Releases a thread

dunhold –thread

dunset

Restores a CLI variable to its default value

dunset *debugger-var*

Restores all CLI variables to their default values

dunset –all

duntil

Runs to a line

duntil [**–back**] *line-number*

Runs to an address

duntil [**–back**] **–address** *addr*

Runs into a function

duntil *proc-name*

dup

Moves up the call stack

dup [*num-levels*]

dwait

Blocks command input until the target processes stop

dwait

dwatch

Defines a watchpoint for a variable

dwatch *variable* [–length *byte-count*] [–p | –g | –t]
[[–l *lang*] –e *expr*] [–t *type*]

Defines a watchpoint for an address

dwatch –address *addr* –length *byte-count* [–p | –g | –t]
[[–l *lang*] –e *expr*] [–t *type*]

dwhat

Determines what a name refers to

dwhat *symbol-name*

dwhere

Displays locations in the call stack

dwhere [–level *level-num*] [*num-levels*] [–args] [–locals] [–registers]
[–noshow_pc][–noshow_fp][–show_image]

Displays all locations in the call stack

dwhere –all [–args] [–locals] [–registers]
[–noshow_pc][–noshow_fp][–show_image]

dworker

Adds or removes a thread from a workers group

dworker { *number* | *boolean* }

errorCodes

Returns a list of all error code tags

TV::errorCodes

Returns or raises error information

TV::errorCodes *number_or_tag* [–raise [*message*]]

exit

Terminates the debugging session

exit [–force]

expr

Manipulates values created by **dprint –nowait**

TV::expr *action* [*susp-eval-id*] [*other-args*]

focus_groups

Returns a list of groups in the current focus

TV::focus_groups

focus_processes

Returns a list of processes in the current focus

TV::focus_processes [–all | –group | –process | –thread]

focus_threads

Returns a list of threads in the current focus

TV::focus_threads [–all | –group | –process | –thread]

group

Gets and sets group properties

TV::group *action* [*object-id*] [*other-args*]

help

Displays help information

help [*topic*]

hex2dec

Converts to decimal

TV::hex2dec *number*

process

Gets and sets process properties

TV::process *action* [*object-id*] [*other-args*]

quit

Terminates the debugging session

quit [**–force**]

read_symbols

Reads symbols from libraries

TV::read_symbols –lib *lib-name-list*

Reads symbols from libraries associated with a stack frame

TV::read_symbols –frame [*number*]

Reads symbols for all frames in the backtrace

TV::read_symbols –stack

respond

Provides responses to commands

TV::respond *response command*

scope

Sets and gets internal scope properties

TV::scope *action* [*object-id*] [*other-args*]

source_process_startup

"Sources" a **.tvd** file when a process is loaded

TV::source_proccess_startup *process_id*

stty

Sets terminal properties

stty [*stty-args*]

symbol

Returns or sets internal TotalView symbol information

TV::symbol *action* [*object-id*] [*other-args*]

thread

Gets and sets thread properties

TV::thread *action* [*object-id*] [*other-args*]

type

Gets and sets type properties

TV::type *action* [*object-id*] [*other-args*]

type_transformation

Creates type transformations and examines properties

TV::type_transformation *action* [*object-id*] [*other-args*]

unalias

Removes an alias

unalias *alias-name*

Removes all aliases

unalias –all

chapter 2

CLI Commands

This chapter contains detailed descriptions of CLI commands.

Command Overview

This section lists all of the CLI commands. It also contains a short explanation of what each command does.

General CLI Commands

The CLI commands in this group provide information on the general CLI operating environment:

- **alias**: Creates or views pseudonym for commands and arguments.
- **capture**: Allows commands that print information to instead send their output to a variable.
- **dlappend**: Appends list elements to a TotalView variable.
- **dset**: Changes or views values of TotalView variables.
- **dunset**: Restores default settings of TotalView variables.
- **help**: Displays help information.
- **stty**: Sets terminal properties.
- **unalias**: Removes a previously defined alias.

CLI Initialization and Termination Commands

These commands initialize and terminate the CLI session, and add processes to CLI control:

- **dattach**: Brings one or more processes currently executing in the normal runtime environment (that is, outside TotalView) under TotalView control.
- **ddetach:** Detaches TotalView from a process.

- **ddlopen**: Dynamically loads shared object libraries.
- **dgroups**: Manipulates and manages groups.
- **dkill:** Kills existing user processes, leaving debugging information in place.
- **dload:** Loads debugging information about the program into TotalView and prepares it for execution.
- **drerun:** Restarts a process.
- **drun:** Starts or restarts the execution of user processes under control of the CLI.
- **exit, quit:** Exits from TotalView, ending the debugging session.

Program Information Commands

The following commands provide information about a program's current execution location, and allow you to browse the program's source files:

- **ddown:** Navigates through the call stack by manipulating the current frame.
- **dexamine**: Display memory contents.
- **dflush**: Unwinds the stack from computations.
- **dga**: Displays global array variables.
- **dlist**: Browses source code relative to a particular file, procedure, or line.
- **dmstat**: Displays memory usage information.
- **dprint:** Evaluates an expression or program variable and displays the resulting value.
- **dptsets**: Shows the status of processes and threads in a P/T set.
- **dstatus:** Shows the status of processes and threads.
- **dup:** Navigates through the call stack by manipulating the current frame.
- **dwhat:** Determines what a name refers to.
- **dwhere:** Prints information about the thread's stack.

Execution Control Commands

The following commands control execution:

- **dcont:** Continues execution of processes and waits for them.
- **dfocus:** Changes the set of processes, threads, or groups upon which a CLI command acts.
- **dgo:** Resumes execution of processes (without blocking).
- **dhalt:** Suspends execution of processes.
- **dhistory**: Provides information for ReplayEngine and lets you work with timestamps.
- **dhold**: Holds threads or processes.
- **dnext:** Executes statements, stepping over subfunctions.
- **dnexti:** Executes machine instructions, stepping over subfunctions.
- **dout**: Runs out of current procedure.
- **dstep:** Executes statements, moving into subfunctions if required.
- **dstepi:** Executes machine instructions, moving into subfunctions if required.
- **dunhold:** Releases held threads.
- **duntil:** Executes statements until a statement is reached.

- **dwait:** Blocks command input until processes stop.
- **dworker:** Adds or removes threads from a workers group.

Action Points

The following action point commands define and manipulate the points at which the flow of program execution should stop so that you can examine debugger or program state:

- **dactions**: Views information on action point definitions and their current status; this command also saves and restores action points.
- **dbarrier**: Defines a process barrier breakpoint.
- **dbreak:** Defines a breakpoint.
- **ddelete**: Deletes an action point.
- **ddisable**: Temporarily disables an action point.
- **denable**: Re-enables an action point that has been disabled.
- **dwatch:** Defines a watchpoint.

Other Commands

The commands in this category do not fit into any of the other categories:

- **dassign:** Changes the value of a scalar variable.
- **dcache**: Clears the remote library cache.
- **dcheckpoint**: Creates a file that can later be used to restart a program.
- **dheap**: Displays information about the heap.
- **drestart**: Restarts a checkpoint.

alias

Creates or views pseudonyms for commands

Format: Creates a new user-defined pseudonym for a command

alias *alias-name defn-body*

Views previously defined aliases

alias [*alias-name*]

Arguments:

alias-name	The name of the command pseudonym being defined.
defn-body	The text that Tcl substitutes when it encounters *alias-name*.

Description: The **alias** command associates a name you specify with text that you define. This text can contain one or more commands. After you create an alias, you can use it in the same way as a native TotalView or Tcl command. In addition, you can include an alias as part of a definition of another alias.

If you do not enter an *alias-name* argument, the CLI displays the names and definitions of all aliases. If you only specify an *alias-name* argument, the CLI displays the definition of the alias.

Because the **alias** command can contain Tcl commands, you must ensure that *defn-body* complies with all Tcl expansion, substitution, and quoting rules.

The TotalView global startup file, **tvdinit.tvd**, defines a set of default aliases. All the common commands have one- or two-letter aliases. (You can obtain a list of these commands by typing **alias**—being sure not to use an argument—in the CLI window.)

You cannot use an alias to redefine the name of a CLI-defined command. You can, however, redefine a built-in CLI command by creating your own Tcl procedure. For example, the following procedure disables the built-in **dwatch** command. When a user types **dwatch**, the CLI executes this code instead of the built-in CLI code.

```
proc dwatch {} {
    puts “The dwatch command is disabled”
}
```

The CLI does not parse *defn-body* (the command’s definition) until it is used. Therefore, you can create aliases that are nonsensical or incorrect. The CLI only detects errors when it tries to execute your alias.

When you obtain help for a command, the help text includes information for TotalView predefined aliases.

Examples:

`alias nt dnext`
Defines a command called **nt** that executes the **dnext** command.

`alias nt` Displays the definition of the **nt** alias.

`alias` Displays the definitions of all aliases.

`alias m {dlist main}`
Defines an alias called **m** that lists the source code of function **main()**.

```
alias step2 {dstep; dstep}
```

Defines an alias called **step2** that does two **dstep** commands. This new command applies to the focus that exists when someone uses this alias.

```
alias step2 {s ; s}
```

Creates an alias that performs the same operations as the one in the previous example. It differs in that it uses the alias for **dstep**. You could also create the following alias which does the same thing:
alias step2 {s 2}.

```
alias step1 {f p1. dstep}
```

Defines an alias called **step1** that steps the first user thread in process 1. All other threads in the process run freely while TotalView steps the current line in your program.

capture

Returns a command's output as a string

Format: capture [–out | –err | –both] [–f *filename*] *command*

Arguments:

–out	Tells the CLI to only capture output that is sent to **stdout**. This option is the default.
–err	Tells the CLI to send the output it captures to **stderr**.
–both	Tells the CLI to send the output it captures to **stdout** and **stderr**.
–f *filename*	Tells the CLI to send the output it captures to *filename*.
command	The CLI command (or commands) whose output is being captured. If you are specifying more than one command, you must enclose them within braces (**{ }**).

Description: The **capture** command executes *command*, capturing all output that would normally go to the console into a string. After *command* completes, it returns the string. This command is analogous to the UNIX shell's back-tick feature (`` ` ``*command*`` ` ``). The **capture** command lets you obtain the printed output of any CLI command so that you can assign it to a variable or otherwise manipulate it.

Examples:

```
set save_stat [ capture st ]
```

Saves the current process status to a Tcl variable.

```
set arg [ capture p argc]
```

Saves the printed value of argc into a Tcl variable.

```
set vbl [ capture {foreach i {1 2 3 4} \
            {p int2_array($i )}} ]
```

Saves the printed output of four array elements into a Tcl variable. Here is sample output:

```
int2_array(1) = -8 (0xfff8)
int2_array(2) = -6 (0xfffa)
int2_array(3) = -4 (0xfffc)
int2_array(4) = -2 (0xfffe)
```

Because the **capture** command records all of the information sent to it by the commands in the **foreach** loop, you do not have to use a **dlist** command.

```
exec cat << [ capture help commands ] > cli_help.txt
```

Writes the help text for all CLI commands to the **cli_help.txt** file.

```
set ofile [open cli_help.txt w]
capture -f $ofile help commands
close $ofile
```

Also writes the help text for all CLI commands to the **cli_help.txt** file. This set of commands is more efficient than the previous command because the captured data is not buffered.

dactions

Displays information, and saves and reloads action points

Format: Displays information about action points

dactions [*ap-id-list*] [**–at** *source-loc*] [**–enabled** | **–disabled**]

Saves action points to a file.

dactions –save [*filename*]

Loads previously saved action points

dactions –load [*filename*]

Arguments:

ap-id-list	A list of action point identifiers. If you specify individual action points, the information that appears is limited to these points. Do not enclose this list within quotes or braces. See the examples at the end of this section for more information. If you omit this argument, the CLI displays summary information about all action points in the processes in the focus set. If you enter one ID, the CLI displays full information for it. If you enter more than one ID, the CLI just displays summary information for each.
–at *source-loc*	Displays the action points at *source-loc*.
–enabled	Shows only enabled action points.
–disabled	Shows only disabled action points.
–save	Writes information about action points to a file.
–load	Restores action point information previously saved in a file.
filename	The name of the file into which TotalView reads and writes action point information. If you omit this file name, TotalView writes action point information to a file named *program_name*.**TVD.v3breakpoints**, where *program_name* is the name of your program.

Description: The **dactions** command displays information about action points in the processes in the current focus. If you do not indicate a focus, the default focus is at the process level. The information is printed; it is not returned.

This command also lets you obtain the action point identifier. You will need to use this identifier when you delete, enable, and disable action points.

The identifier is returned when TotalView creates the action point. The CLI prints this ID when the thread stops at an action point.

You can include action point identifiers as arguments to the command when more detailed information is needed. The **–enabled** and **–disabled** options restrict output to action points in one of these states.

You cannot use the **dactions** command when you are debugging a core file or before TotalView loads executables.

The –**save** option tells TotalView to write action point information to a file so that either you or TotalView can restore your action points later. The –**load** option tells TotalView to immediately read the saved file. If you use the *file-name* argument with either of these options, TotalView either writes to or reads from this file. If you do not use this argument, TotalView uses a file named *program_name*.**TVD.v3breakpoints** where *program_name* is the name of your program. TotalView writes this file into the directory in which your program resides.

The information saved includes expression information associated with the action point and whether the action point is enabled or disabled. For example, if your program's name is **foo**, TotalView writes this information to **foo.TVD.v3breakpoints**.

TotalView does not save information about watchpoints.

If a file with the default name exists, TotalView can read this information when it starts your program. When TotalView exits, it can create the default. For more information, see the **File > Preference** Action Points Page information in the online Help.

Command alias:

Alias	Definition	Description
ac	dactions	Displays all action points

Examples:

`ac -at 81`

Displays information about the action points on line 81. (This example uses the alias instead of the full command name.) Here is the output from this command:

```
ac –at 81
1 shared action point for group 3:
   1 addr=0x10001544 [arrays.F#81] Enabled
      Share in group: true
      Stop when hit: group
```

`dactions 1 3`

Displays information about action points 1 and 3, as follows:

```
2 shared action points for process 1:
   1 addr=0x100012a8 [arrays.F#56] Enabled
   3 addr=0x100012c0 [arrays.F#57] Enabled
```

If you have saved a list of action points as a string or as a Tcl list, you can use the eval command to process the list's elements. For example:

```
d1.<> dactions
2 shared action points for group 3:
    3 [global_pointer_ref.cxx#52] Enabled
    4 [global_pointer_ref.cxx#53] Enabled
d1.<> set group1 "3 4"
3 4
d1.<> eval ddisable $group1
```

```
d1.<> ac
2 shared action points for group 3:
    3 [global_pointer_ref.cxx#52] Disabled
    4 [global_pointer_ref.cxx#53] Disabled
```

`dfocus p1 dactions`

Displays information about all action points defined in process 1.

`dfocus p1 dactions -enabled`

Displays information about all enabled action points in process 1.

dassign

Changes the value of a scalar variable

Format: **dassign** *target value*

Arguments:

target	The name of a scalar variable in your program.
value	A source-language expression that evaluates to a scalar value. This expression can use the name of another variable.

Description: The **dassign** command evaluates an expression and replaces the value of a variable with the evaluated result. The location can be a scalar variable, a dereferenced pointer variable, or an element in an array or structure.

The default focus for the **dassign** command is *thread*. If you do not change the focus, this command acts upon the *thread of interest*. If the current focus specifies a width that is wider than **t** (thread) and is not **d** (default), **dassign** iterates over the threads in the focus set and performs the assignment in each. In addition, if you use a list with the **dfocus** command, the **dassign** command iterates over each list member.

The CLI interprets each symbol name in the expression according to the current context. Because the value of a source variable might not have the same value across threads and processes, the value assigned can differ in your threads and processes. If the data type of the resulting value is incompatible with that of the target location, you must cast the value into the target's type. (*Casting* is described in Chapter 13 of the *TotalView Users Guide*.)

You need to know the following about assigning characters and strings:

- If you are assigning a character to a *target*, place the character value within single-quotation marks; for example, **'c'**.
- You can use the standard C language escape character sequences; for example, **\n and \t**. These escape sequences can also be in a character or string assignment.
- If you are assigning a string to a *target*, place the string within quotation marks. However, you must escape the quotation marks so they are not interpreted by Tcl; for example, **\"The quick brown fox\"**.

If *value* contains an expression, the TotalView expression system evaluates the expression

Command alias:

Alias	Definition	Description
as	dassign	Changes a scalar variable's value

Examples:

```
dassign scalar_y 102
```

Stores the value 102 in each occurrence of variable **scalar_y** for all processes and threads in the current set.

```
dassign i 10*10
```

Stores the value 100 in variable **i**.

`dassign i i*i` Does not work and the CLI displays an error message. If **i** is a simple scalar variable, you can use the following statements:

```
set x [lindex [capture dprint i] 2]
dassign i [expr $x * $x]
```

`f {p1 p2 p3} as scalar_y 102`
Stores the value 102 in each occurrence of variable **scalar_y** contained in processes 1, 2, and 3.

dattach

Brings currently executing processes under CLI control

Format:

```
dattach [ –g gid ] [ –r hname ]
        [ –ask_attach_parallel | –no_attach_parallel ]
        [ –replay | -no_replay ]
        [ –c corefile-name ] [ –rank num ]
        [ –e ] filename [ pid-list ]
        [ –parallel_attach_subset subset-specification ]
```

Arguments:

–g *gid*	Sets the control group for the processes being added to group *gid*. This group must already exist. (The CLI **GROUPS** variable contains a list of all groups. See **GROUPS** on page 188 for more information.)
–r *hname*	The host on which the process is running. The CLI launches a TotalView Server on the host machine if one is not already running there. See the *Setting Up Parallel Debugging Sessions* chapter of the *TotalView Users Guide* for information on the launch command used to start this server. Setting a host sets it for all PIDs attached to in this command. If you do not name a host machine, the CLI uses the local host.
–ask_attach_parallel	Asks whether TotalView should attach to parallel processes of a parallel job. The default is to automatically attach to processes. For additional information, see the Dynamic Libraries Page in the **File > Preferences** Dialog Box in the online Help.
–no_attach_parallel	Do not attach to any additional parallel processes in a parallel job. For additional information, see the Dynamic Libraries Page in the **File > Preferences** Dialog Box in the online Help.
–replay | –no_replay	These options enable and disable the ReplayEngine the next time the program is restarted.
-c *corefile-name*	Tells the CLI that it should load the core file named in the argument that follows. If you use this option, you must also specify a file name (*filename*).
-rank *num*	Specifies the rank associated with the executable being loaded. While this can be used independently, this option is best used with core files.
–e	Tells the CLI that the next argument is a file name. You need to use this argument if the file name begins with a dash (–) or only uses numeric characters.
filename	The name of the executable. Setting an executable here sets it for all PIDs being attached to in this command. If you do not include this argument, the CLI tries to determine the executable file from the process. Some architectures do not allow this to occur.

pid-list A list of system-level process identifiers (such as a UNIX PID) naming the processes that TotalView controls. All PIDs must reside on the same system, and they are placed in the same control group.

If you need to place the processes in different groups or attach to processes on more than one system, you must use multiple **dattach** commands.

parallel_attach_subset *subset_specification*

Defines a list of MPI ranks to be attached to when an MPI job is created or attached to. The list is space-separated; each element can have one of 3 forms:

rank: specifies that rank only

rank1-rank2: specifies all ranks between rank1 and rank2, inclusive

rank1-rank2:stride: specifies every strideth rank between rank1 and rank2

A rank specification can be either a number or "**$max**" which represents the last rank in the MPI job.

For example:

```
dattach -parallel_attach_subset {1 2 4-6
7-$max:2} mpirun
```

will attach to ranks 1, 2, 4, 5, 6, 7, 9, 11, 13,....

Description: The **dattach** command tells the CLI to attach to one or more processes, making it possible to continue process execution under CLI control.

TotalView Individual: You can only attach to processes running on the computer upon which you installed TotalView Individual.

This command returns the TotalView process ID (DPID) as a string. If you specify more than one process in a command, the **dattach** command returns a list of DPIDs instead of a single value.

TotalView places all processes to which it attaches in one **dattach** command in the same control group. This lets you place all processes in a multiprocess program executing on the same system in the same control group.

If a program has more than one executable, you must use a separate **dattach** command for each one.

If you have not loaded *filename* already, the CLI searches for it. The search includes all directories in the **EXECUTABLE_PATH** CLI variable.

The process identifiers specified in the *pid-list* must refer to existing processes in the runtime environment. TotalView attaches to the processes, regardless of their execution states.

Command alias:

Alias	Definition	Description
at	dattach	Brings the process under CLI control

Examples:

```
dattach mysys 10020
```

Loads debugging information for **mysys** and brings the process known to the run-time system by PID 10020 under CLI control.

```
dattach -e 123 10020
```

Loads file 123 and brings the process known to the run-time system by PID 10020 under CLI control.

```
dattach -g 4 -r Enterprise myfile 10020
```

Loads **myfile** that is executing on the host named **Enterprise** into group 4, and brings the process known to the run-time system by PID 10020 under CLI control. If a TotalView Server (**tvdsvr**) is not running on **Enterprise**, the CLI will start it.

```
dattach my_file 51172 52006
```

Loads debugging information for **my_file** and brings the processes corresponding to PIDs 51172 and 52006 under CLI control.

```
set new_pid [dattach -e mainprog 123]
dattach -r otherhost -g $CGROUP($new_pid) -e slave 456
```

Begins by attaching to **mainprog** running on the local host. It then attaches to **slave** running on the **otherhost** host and inserts them both in the same control group.

dbarrier

Defines a process or thread barrier breakpoint

Format: Creates a barrier breakpoint at a source location

dbarrier *breakpoint-expr* [**–stop_when_hit** *width*]
[**–stop_when_done** *width*] [**–pending**]

Creates a barrier breakpoint at an address

dbarrier –address *addr* [**–stop_when_hit** *width*]
[**–stop_when_done** *width*] [**–pending**]

Arguments:

breakpoint-expr

This argument can be entered in more than one way. In most cases, you will be typing a line number or a path-name containing a file name, function name, and line number, each separated by **#** characters (for example, **#file#line**). If you omit parts of this specification, the CLI creates them for you. For more information, see "*Qualifying Symbol Names*" in Chapter 13 of the *TotalView Users Guide*.

For more information on breakpoint expressions, see "*Breakpoint Expressions*" on page 32.

–address *addr*

The barrier breakpoint location as an absolute address in the address space of the program.

–stop_when_hit *width*

Tells the CLI what else to stop when it stops the thread that arrives at a barrier point.

If you do not use this option, the value of the **BARRIER_STOP_ALL** variable indicates what TotalView stops.

This command's *width* argument indicates what else TotalView stops. You can enter one of the following three values:

group	Stops all processes in the control group when the execution reaches the barrier point.
process	Stops the process that hit the barrier.
none	Stops the thread that hit the barrier; that is, the thread is held and all other threads continue running. If you apply this width to a process barrier breakpoint, TotalView stops the process that hit the breakpoint.

–stop_when_done *width*

After all processes or threads reach the barrier, the CLI releases all processes and threads held at the barrier. (*Released* means that these threads and processes can run.) Setting this option tells the CLI to stop additional threads contained in the same **group** or **process**.

If you do not use this option, the value of the **BARRIER_STOP_WHEN_DONE** variable indicates what else TotalView stops.

The *width* argument indicates what else is stopped. You can enter one of the following three values:

group	Stops the entire control group when the barrier is satisfied.
process	Stops the processes that contain threads in the satisfaction set when the barrier is satisfied.
none	Stops the satisfaction set. For process barriers, **process** and **none** have the same effect. This is the default if the **BARRIER_STOP_WHEN_DONE** variable is **none**.

–pending If TotalView cannot find a location to set the barrier, adding this option tells TotalView to create the barrier anyway. As shared libraries are read, TotalView checks to see if it can be set in the newly loaded library. For more information on this option, see **dbreak** on page 31.

Description: The **dbarrier** command sets a process or thread barrier breakpoint that triggers when execution arrives at a location. This command returns the ID of the newly created breakpoint.

You most often use the **dbarrier** command to synchronize a set of threads. The P/T set defines which threads the barrier affects. When a thread reaches a barrier, it stops, just as it does for a breakpoint. The difference is that TotalView prevents—that is, holds—each thread that reaches the barrier from responding to resume commands (for example, **dstep**, **dnext**, and **dgo**) until all threads in the affected set arrive at the barrier. When all threads reach the barrier, TotalView considers the barrier to be *satisfied* and releases these threads. They are just *released*; they are not continued. That is, TotalView leaves them stopped at the barrier. If you continue the process, those threads stopped at the barrier also run along with any other threads that were not participating with the barrier. After the threads are released, they can respond to resume commands.

If the process is stopped and then continued, the held threads, including the ones waiting on an unsatisifed barrier, do not run. Only unheld threads run.

The satisfaction set for the barrier is determined by the current focus. If the focus group is a thread group, TotalView creates a thread barrier:

- When a thread hits a process barrier, TotalView holds the thread's process.
- When a thread hits a thread barrier, TotalView holds the thread; TotalView might also stop the thread's process or control group. While they are stopped, neither is held.

TotalView determines the default focus width based on the setting of the **SHARE_ACTION_POINT** variable. If it is set to true, the default is group. Otherwise, it is process.

TotalView determines what processes and threads are part of the satisfaction set by taking the intersection of the share group with the focus set. (Barriers cannot extend beyond a share group.)

The CLI displays an error message if you use an inconsistent focus list.

Barriers can create deadlocks. For example, if two threads participate in two different barriers, each could be left waiting at different barriers, barriers that can never be satisfied. A deadlock can also occur if a barrier is set in a procedure that is never invoked by a thread in the affected set. If a deadlock occurs, use the **ddelete** *command to remove the barrier, since deleting the barrier also releases any threads held at the barrier.*

The –**stop_when_hit** option tells TotalView what other threads to stop when a thread arrives at a barrier.

The –**stop_when_done** option controls the set of additional threads that TotalView stops when the barrier is finally satisfied. That is, you can also stop an additional collection of threads after the last expected thread arrives and all the threads held at the barrier are released. Normally, you want to stop the threads contained in the control group.

If you omit a *stop* option, TotalView sets the default behavior by using the **BARRIER_STOP_ALL** and **BARRIER_STOP_WHEN_DONE** variables. For more information, see the **dset** command.

The **none** argument for these options tells the CLI not to stop additional threads.

- If –**stop_when_hit** is **none** when a thread hits a thread barrier, TotalView stops only that thread; it does not stop other threads.
- If –**stop_when_done** is **none**, TotalView does not stop additional threads, aside from the ones that are already stopped at the barrier.

TotalView places the barrier point in the processes or groups specified in the current focus, as follows:

- If the current focus does not indicate an explicit group, the CLI creates a process barrier across the share group.
- If the current focus indicates a process group, the CLI creates a process barrier that is satisfied when all members of that group reach the barrier.
- If the current focus indicates a thread group, TotalView creates a thread barrier that is satisfied when all members of the group arrive at the barrier.

The following example illustrates these differences. If you set a barrier with the focus set to a control group (the default), TotalView creates a process barrier. This means that the –**stop_when_hit** value is set to **process** even though you specified **thread**.

```
d1.<> dbarrier 580 -stop_when_hit thread
2
d1.<> ac 2
1 shared action point for group 3:
  2 addr=0x120005598 [../regress/fork_loop.cxx#580]
Enabled (barrier)
   Share in group: true
   Stop when hit: process
   Stop when done: process
   process barrier; satisfaction set = group 1
```

However, if you create the barrier with a specific workers focus, the stop when hit property remains set to **thread**:

```
1.<> baw 580 -stop_when_hit thread
1
d1.<> ac 1
1 unshared action point for process 1:
  1 addr=0x120005598 [../regress/fork_loop.cxx#580]
                    Enabled (barrier)
   Share in group: false
   Stop when hit: thread
   Stop when done: process
   thread barrier; satisfaction set = group 2
```

Command alias:

Alias	Definition	Description
ba	dbarrier	Defines a barrier.
baw	{dfocus pW dbarrier –stop_when_done process}	Creates a thread barrier across the worker threads in the process of interest. TotalView sets the set of threads stopped when the barrier is satisfied to the process that contains the satisfaction set.
BAW	{dfocus gW dbarrier –stop_when_done group}	Creates a thread barrier across the worker threads in the share group of interest. The set of threads stopped when the barrier is satisfied is the entire control group.

Examples:

`dbarrier 123`
Stops each process in the control group when it arrives at line 123. After all processes arrive, the barrier is satisfied and TotalView releases all processes.

`dfocus {p1 p2 p3} dbarrier my_proc`
Holds each thread in processes 1, 2, and 3 as it arrives at the first executable line in procedure **my_proc**. After all threads arrive, the barrier is satisfied and TotalView releases all processes.

`dfocus gW dbarrier 642 -stop_when_hit none`
Sets a thread barrier at line 642 in the workers group. The process is continued automatically as each thread arrives at the barrier. That is, threads that are not at this line continue running.

dbreak

Defines a breakpoint

Format: Creates a breakpoint at a source location

dbreak *breakpoint-expr* [–p | –g | –t] [[–l *lang*] –e *expr*]
[–pending]

Creates a breakpoint at an address

dbreak –address *addr* [–p | –g | –t] [[–l *lang*] –e *expr*]
[–pending]

Arguments:

breakpoint-expr
: This argument can be entered in more than one way. In most cases, you will be typing a line number or a pathname containing a file name, function name, and line number, each separated by **#** characters (for example, **#file#line**). If you omit parts of this specification, the CLI creates them for you. For more information, see "*Qualifying Symbol Names*" in Chapter 13 of the *TotalView Users Guide*.

  Breakpoint expressions are discussed later in this section.

–address *addr*
: The breakpoint location specified as an absolute address in the address space of the program.

–p
: Tells TotalView to stop the process that hit this breakpoint. You can set this option as the default by setting the **STOP_ALL** variable to **process**. See **dset** on page 97 for more information.

–g
: Tells TotalView to stop all processes in the process's control group when execution reaches the breakpoint. You can set this option as the default by setting the **STOP_ALL** variable to **group**. See **dset** on page 97 for more information.

–t
: Tells TotalView to stop the thread that hit this breakpoint. You can set this option as the default by setting the **STOP_ALL** variable to **thread**. See **dset** on page 97 for more information.

–l *lang*
: Sets the programming language used when you are entering expression *expr*. The languages you can enter are **c**, **c++**, **f7**, **f9**, and **asm** (for C, C++, FORTRAN 77, Fortran 9x, and assembler, respectively). If you do not specify a language, TotalView assumes that you wrote the expression in the same language as the routine at the breakpoint.

–e *expr*
: When the breakpoint is hit, TotalView evaluates expression *expr* in the context of the thread that hit the breakpoint. The language statements and operators you can use are described in Chapter 16 of the *TotalView Users Guide*.

–pending
: If TotalView cannot find a location to set the breakpoint, adding this option tells TotalView to create the

breakpoint anyway. As shared libraries are read, TotalView checks to see if it can be set in the newly loaded library.

Description: The **dbreak** command defines a breakpoint or evaluation point that TotalView triggers when execution arrives at the specified location. The breakpoint tells each thread that it should stop when it arrives at a breakpoint. This command returns the ID of the new breakpoint. The CLI might not be able to set a breakpoint at the line you specify. This occurs when a line does not contain an executable statement.

Specifying a procedure name without a line number tells the CLI to set an action point at the beginning of the procedure. If you do not name a file, the default is the file associated with the current source location.

If you try to set a breakpoint at a line at which TotalView cannot stop execution, it sets one at the nearest following line where it can halt execution.

If, after evaluating the breakpoint expression, TotalView determines the location represented by the expression does not exist, it can still set a breakpoint if you use the **–pending** option. This option tells TotalView that it should check shared libraries that it subsequently loads to see if it can set the breakpoint. If a location is found, it is set. Stated in a different way, TotalView normally creates and sets a breakpoint at the same time. The option tells it to separate these two actions.

When the CLI displays information on a breakpoint's status, it displays the location where execution actually stops.

If the CLI encounters a *stop group* breakpoint, it suspends each process in the group as well as the process that contains the triggering thread. The CLI then shows the identifier of the triggering thread, the breakpoint location, and the action point identifier.

TotalView determines the default focus width based on the setting of the **SHARE_ACTION_POINT** variable. If it is set to **true**, the default is group. Otherwise, it is process.

One possibly confusing aspect of using expressions is that their syntax differs from that of Tcl. This is because you need to embed code written in Fortran, C, or assembler in Tcl commands. In addition, your expressions often include TotalView built-in functions. For example, if you want to use the TotalView **$tid** built-in function, you need to type it as **\$tid**.

Breakpoint Expressions

A breakpoint expression can evaluate to more than one source line. If the expression evaluates to a function that has multiple overloaded implementations, TotalView sets a barrier on each of the overloaded functions. The expression can begin with one of the following options:

class *class_name* Tells TotalView that the breakpoint expression is a class name and it should set a barrier on all methods of that class.

virtual *class*::*signature*

Tells TotalView to set a a barrier on virtual function that override the signature in the class specified by the breakpoint expression.

Set a barrier at the line specified by *breakpoint-expr* or the absolute address *addr*. You can enter a breakpoint expression that are sets of addresses at which the barrier is placed, and are as follows:

- [[##image#]filename#]line_numbe r

  Indicates all addresses at this line number.
- A function signature; this can be a partial signature

  Indicates all addresses that are the addresses of functions matching *signature*. If parts of a function signature are missing, this expression can match more than one signature. For example, "**f**" matches "**f(void)**" and "**A::f(int)**". You cannot specify a return type in a signature.
- class *class_name*

  Names a set containing the addresses of all member functions of class *class_name*.
- virtual *class*::*signature*

  Names the set of addresses of all virtual member functions that match *signature*, and which are in the classes or derived from the class.

Command alias:

Alias	Definition	Description
b	break	Sets a breakpoint
bt	{dbreak t}	Sets a breakpoint only on the thread of interest

Examples: For all examples, assume that the current process set is **d2.<** when the breakpoint is defined.

`dbreak 12`
Suspends process 2 when it reaches line 12. However, if the **STOP_ALL** variable is set to **group**, all other processes in the group are stopped. In addition, if you set the **SHARE_ACTION_POINT** variable to **true**, the breakpoint is placed in every process in the group.

`dbreak -address 0x1000764`
Suspends process 2 when execution reaches address 0x1000764.

`b 12 -g`
Suspends all processes in the current control group when execution reaches line 12.

`dbreak 57 -l f9 -e {goto $63}`
Causes the thread that reaches the breakpoint to transfer to line 63. The host language for this statement is Fortran 90 or Fortran 95.

`dfocus p3 b 57 -e {goto $63}`
In process 3, sets the same evaluation point as the previous example.

dcache

Clears the remote library cache

Format:	dcache –flush	
Arguments:	–flush	Delete all files from the library cache that are not currently being used.

Description: The **dcache –flush** command tells TotalView to remove the library files that it places in your cache. This cache is located in the **.totalview/lib_cache** subdirectory contained in your home directory.

When you are debugging programs on remote systems that use libraries that either do not exist on the host or whose version differ, TotalView copies the library files into your cache. This cache can become large.

TotalView automatically deletes cached library files that it hasn't used in the last week. If you need to reclaim additional space, this command removes files not currently being used and that are not quite old enough for TotalView to automatically delete.

dcheckpoint

Creates a checkpoint image of processes (IBM and SGI)

Format: Creates a checkpoint on SGI IRIX

dcheckpoint [*after_checkpointing*] [**–by** *process_set*] [**–no_park**]
[**–ask_attach_parallel** | **–no_attach_parallel**]
[**–no_preserve_ids**] [**–force**] *checkpoint-name*

Creates a checkpoint on IBM AIX

dcheckpoint [**–by** *process_set*] [**–delete** | **–halt**]

Arguments:

after_checkpointing Defines the state of the process both before and after the checkpoint. Use one of the following options:

–delete	Processes exit after the checkpoint occurs.
–detach	(SGI only) Processes continue running after the checkpoint occurs. In addition, TotalView detaches from them.
–go	(SGI only) Processes continue running after the checkpoint occurs.
–halt	Processes halt after the checkpoint occurs.

–by *process_set* Indicates the set of processes to checkpoint and a group of related processes. If you do not use a *process_set* option, TotalView only checkpoints the focus process. (On AIX, the checkpoint is the scope of the Parallel Environment job.) Your options are:

ash	Checkpoints the array session.
pgid	Checkpoints the entire UNIX process group.
sid	Checkpoints the entire process session.

–no_park (SGI only) Tells TotalView not to *park* all processes before it begins checkpointing them. If you use this option, you must use the **drestart** command's **–no_unpark** option. If you restart the checkpoint from a shell, you must use this option.

–ask_attach_parallel
(SGI only) Asks whether TotalView reattaches to parallel processes of a parallel job. (Some systems automatically detach you from processes being checkpointed.)

–no_attach_parallel
(SGI only) Tells TotalView not to reattach to processes from which the checkpointing processes detached. (Some systems automatically detach you from processes being checkpointed.)

–no_preserve_ids (SGI only) Lets TotalView assign new IDs when it restarts a checkpoint. If you do not use this option, TotalView uses the same IDs.

–force (SGI only) Tells TotalView to overwrite an existing checkpoint.

checkpoint-name (SGI only) Specifies the name being assigned to the checkpoint. TotalView ignores this name if you are creating a checkpoint on an IBM/RS6000 machine.

Description: The **dcheckpoint** command saves program and process information to the *checkpoint-name* file. This information includes process and group IDs. Later, you use the **drestart** command to restart the program.

This command does not save TotalView breakpoint information. If you need to save tis information, use the **dactions** *command.*

The following restrictions exist when you are trying to checkpoint IRIX processes:

- IRIX does not checkpoint a process that is running remotely and which communicates using sockets. Since the TotalView Server (**tvdsvr**) uses sockets to redirect **stdin**, **stdout**, and **stderr**, you need to use the **drun** command to modify the way your processes send information to a **tty** before you create a checkpoint.
- If you are using SGI MPI, you need to use the **-cpr** command-line option. Use the ASH option with MPI checkpoints

The *after_checkpointing* options let you specify what happens after the checkpoint operation concludes. If you do not specify an option, the CLI tells the checkpointed processes to stop. This lets you investigate a program's state at the checkpoint position. In contrast, the –**go** option tells the CLI to let the processes continue to run. You use the –**detach** and –**halt** options less frequently. The –**detach** option shuts down the CLI and leaves the processes running. This command's –**halt** option is similar to the –**detach** option, differing only in that processes started by the CLI and TotalView also terminate.

The *process_set* options tell TotalView which processes to checkpoint. Although the focus set can only contain one process, you can also include processes in the same process group, process session, process hierarchy, or array session in the same checkpoint. If you do not use one of the –**by** options, TotalView only checkpoints the focus process.

If the focus group contains more than one process, the CLI displays an error message.

Before TotalView begins checkpointing your program, it temporarily stops (that is, *parks*) the processes that are being checkpointed. Parking ensures that the processes do not run freely after a **dcheckpoint** or **drestart** operation. (If they did, your code would begin running before you get control of it.) If you plan to restart the checkpoint file outside of TotalView, you must use the –**no_park** option.

When you create the checkpoint, the CLI detaches from processes before they are checkpointed. By default, the CLI automatically reattaches to them. If you do not want this to occur, use the –**no_attach_parallel** option to tell the CLI not to reattach, or use the –**ask_attach_parallel** option to tell the CLI to ask you whether to reattach.

Examples:

`dcheckpoint check1`

Checkpoints the current process. TotalView writes the checkpoint information to the *check*1 file. These processes stop.

`f3 dcheckpoint check1`

Checkpoints process 3. Process 3 stops. TotalView writes the checkpoint information to the *check*1 file.

`f3 dcheckpoint -go check1`

Checkpoints process 3. Process 3 continues to run. TotalView writes the checkpoint information to the *check*1 file.

`f3 dcheckpoint -by pgid -detach check1`

Checkpoints process 3 and all other processes in the same UNIX process group. All of the checkpointed processes continue running but they run detached from the CLI. TotalView writes the checkpoint information to the *check*1 file.

dcont

Continues execution and waits for execution to stop

Format: dcont

Arguments: This command has no arguments

Description: The **dcont** command continues all processes and threads in the current focus, and then waits for all of them to stop.

This command is a Tcl macro, with the following definition:

```
proc dcont {args} {uplevel dgo; "dwait $args" }
```

You often want this behavior in scripts. You seldom want to do interactively.

You can interrupt this action by typing Ctrl+C. This tells TotalView to stop executing these processes.

A **dcont** command completes when all threads in the focus set of processes stop executing. If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
co	dcont	Resume
CO	{dfocus g dcont}	Resume at group-level

Examples:

`dcont` Resumes execution of all stopped threads that are not held and which belong to processes in the current focus. (This command does not affect threads that TotalView is holding held at barriers.) The command blocks further input until all threads in all target processes stop. After the CLI displays its prompt, you can enter additional commands.

`dfocus p1 dcont`
Resumes execution of all stopped threads that are not held and which belong to process 1. The CLI does not accept additional commands until the process stops.

`dfocus {p1 p2 p3} co`
Resumes execution of all stopped threads that are not held and which belong to processes 1, 2, and 3.

`CO` Resumes execution of all stopped threads that are not held and which belong to the current group.

ddelete

Deletes action points

Format:

Deletes the specified action points

ddelete *action-point-list*

Deletes all action points

ddelete –a

Arguments:

action-point-list	A list of the action points to delete.
–a	Tells TotalView to delete all action points in the current focus.

Description:

The **ddelete** command permanently removes one or more action points. The argument to this command tells the CLI which action points to delete. The –**a** option tells the CLI to delete all action points.

If you delete a barrier point, the CLI releases the processes and threads held at it.

If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
de	ddelete	Deletes action points

Examples:

`ddelete 1 2 3` Deletes action points 1, 2, and 3.

`ddelete -a` Deletes all action points associated with processes in the current focus.

`dfocus {p1 p2 p3 p4} ddelete -a`

Deletes all the breakpoints associated with processes 1 through 4. Breakpoints associated with other threads are not affected.

`dfocus a de -a`

Deletes all action points known to the CLI.

ddetach

Detaches from processes

Format: **ddetach**

Arguments: This command has no arguments

Description: The **ddetach** command detaches the CLI from all processes in the current focus. This *undoes* the effects of attaching the CLI to a running process; that is, the CLI releases all control over the process, eliminates all debugger state information related to it (including action points), and allows the process to continue executing in the normal run-time environment.

You can detach any process controlled by the CLI; the process being detached does not have to be originally loaded with a **dattach** command.

After this command executes, you are no longer able to access program variables, source location, action point settings, or other information related to the detached process.

If a single thread serves as the set, the CLI detaches the process that contains the thread. If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
det	ddetach	Detaches from processes

Examples:

`ddetach` Detaches the process or processes that are in the current focus.

`dfocus {p4 p5 p6} det`
Detaches processes 4, 5, and 6.

`dfocus g2 det` Detaches all processes in the control group associated with process 2.

ddisable

Temporarily disables action points

Format: Disables the specified action points

ddisable *action-point-list* [**–block** *number-list*]

Disables all action points

ddisable –a

Arguments:

action-point-list	A list of the action points to disable.
–block *number-list*	If you set a breakpoint on a line that is ambiguous, use this option to say which instances to disable. You can obtain a list of these numbers if you use the **dactions** command.
–a	Tells TotalView to disable all action points.

Description: The **ddisable** command temporarily deactivates action points. This command does not, however, delete them.

You can explicitly name the IDs of the action points to disable or you can disable all action points.

If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
di	ddisable	Temporarily disables action points

Examples:

`ddisable 3 7` Disables the action points whose IDs are 3 and 7.

`di -a` Disables all action points in the current focus.

`dfocus {p1 p2 p3 p4} ddisable -a`

Disables all action points associated with processes 1 through 4. Action points associated with other processes are not affected.

`di 1 -block 3 4`

Disables the action points associated with blocks 3 and 4. That is, one logical action point can map to more than one actual action point if you set the action point at an ambiguous location.

`ddisable 1 2 -block 3 4`

Disables the action points associated with blocks 3 and 4 in action points 1 and 2.

ddlopen

Dynamically loads shared object libraries

Format: Dynamically loads a shared object library

ddlopen [–now | –lazy] [–local | –global] [–mode *int* **]** *filespec*

Displays information about shared object libraries

ddlopen [–list { *dll-ids*... **]**

Arguments:

–now	Includes **RTLD_NOW** in the **dlopen** command's mode argument. (Now means immediately resolve all undefined symbols.)
–lazy	Includes **RTLD_LAZY** in the **dlopen** command's mode argument. (Lazy means that the **dlopen()** function does not try to resolve unresolved symbols. Instead, symbols are resolved as code is executed.)
–local	Includes **RTLD_GLOBAL** in the **dlopen** command's mode argument. (Local means that library symbols are not available to libraries that the program subsequently loads.) This argument is the default.
–global	Includes **RTLD_LOCAL** in the **dlopen** command's mode argument. (Global means that library symbols are available to libraries that the program subsequently loads.)
–mode *int*	The integer arguments are ORed into the other mode flags passed to the **dlopen()** function. (See your operating system's documentation for information on these flags.)
filespec	The shared library to load.
–list	Displays information about the listed DLL IDs. If you omit this option or use the **–list** without a DLL id list, TotalView displays information about all DLL IDs.
dll-ids	A list of one or more DLL IDs.

Description: The **ddlopen** command dynamically loads shared object libraries, or lists the shared object libraries that you loaded using this or the **Tools > Dynamic Libraries** command.

If use a *filespec* argument, TotalView performs a **dlopen** operation on this file in each process in the current P/T set. If you are running on the IBM AIX operating system, you can add a parenthesized library module name to the end of the *filespec* argument.

dlopen(3), **dlerror**(3), *and other related routines are not part of the default runtime libraries on* AIX, *Solaris, and* Red Hat Linux. *Instead, they are in the* **libdl** *system library. Consequently, you must link your program using the* **–ldl** *option if you want to use the* **ddlopen** *command*.

The **–now** and **–lazy** options indicate whether **dlopen** immediately resolves unresolved symbol references or defer resolving them until the target program references them. If you don't use either option, TotalView uses your operating system's default. (Not all platforms support both alternatives. For example, AIX treats **RTLD_LAZY** the same as **RTLD_NOW**).

The –**local** and –**global** options determine if symbols from the newly loaded library are available to resolve references. If you don't use either option, TotalView uses the target operating system's default. (HP Tru 64 UNIX doesn't support either alternative; its operation is equivalent to using the –**global** option. IRIX, Solaris, and Linux only support the –**global** option; if you don't specify an option, the default is the –**local** option.)

After you enter this command, the CLI waits until all **dlopen** calls complete across the current focus. The CLI then returns a unique *dll-id* and displays its prompt, which means that you can enter additional CLI commands. However, if an event occurs (for example, a **$stop,** a breakpoint in user function called by static object constructors, a SEGV, and so on), the **ddlopen** command throws an exception that describes the event. The first exception subcode in the **errorCode** variable is the DLL ID for the suspended **dlopen()** function call.

If an error occurs while executing the **dlopen()** function, TotalView calls the **dlerror()** function in the target process, and then prints the returned string.

A DLL ID describes a shareable object that was dynamically loaded by the **ddlopen** command. You can use the **TV:dll** command to obtain information about and delete these objects. If all **dlopen()** calls return immediately, the **ddlopen** command returns a unique DLL ID that you can also use with the **TV::dll** command.

Every DLL ID is also a valid breakpoint ID, representing the expressions used to load and unload DLLs; you can manipulate these breakpoints using the **TV::expr** command.

If you do not use a *filespec* argument or if you use the –**list** option without using a DLL ID argument, TotalView prints information about objects loaded using **ddlopen**. If you do use a DLL ID argument, TotalView prints information about DLLs loaded into all processes in the focus set; otherwise, TotalView prints information about just those DLLs. The **ddlopen** command prints its output directly to the console.

The **ddlopen** command calls the **dlopen()** function and it can change the string returned by the **dlerror()** function. I t can also change the values returned to the application by any subsequent **dlerror()** call.

Examples:

```
ddlopen "mpistat.so"
```

Loads **mpistat.so** library file. The returned argument lists the process into which TotalView loaded the library.

```
dfocus g ddlopen "mpistat.so(mpistat.o)"
```

Loads the module **mpistat.o** in the AIX DLL library **mpistat.so** into all members of the current process's control group:

`ddlopen -lazy -global "mpistat.so"`	Loads **mpistat.so** into process 1, and does not resolve outstanding application symbol requests to point to **mpistat**. However, TotalView uses the symbols in this library if it needs them.
`ddlopen`	Prints the list of shared objects dynamically loaded by the **ddlopen** command.

ddown

Moves down the call stack

Format: **ddown** [*num-levels*]

Arguments:

num-levels	Number of levels to move down. The default is 1.

Description: The **ddown** command moves the selected stack frame down one or more levels. It also prints the new frame's number and function name.

Call stack movements are all relative, so using the **ddown** command effectively moves down in the call stack. (If up is in the direction of the **main()** function, then down is back to where you were before you moved through stack frames.)

Frame 0 is the most recent—that is, the currently executing—frame in the call stack, frame 1 corresponds to the procedure that invoked the currently executing frame, and so on. The call stack's depth is increased by one each time a procedure is entered, and decreased by one when it is exited.

The command affects each thread in the focus. That is, if the current width is process, the **ddown** command acts on each thread in the process. You can specify any collection of processes and threads as the target set.

In addition, the **ddown** command modifies the current list location to be the current execution location for the new frame; this means that a **dlist** command displays the code that surrounds this new location.

The context and scope changes made by this command remain in effect until the CLI executes a command that modifies the current execution location (for example, the **dstep** command), or until you enter either a **dup** or **ddown** command.

If you tell the CLI to move down more levels than exist, the CLI simply moves down to the lowest level in the stack, which was the place where you began moving through the stack frames.

Command alias:

Alias	Definition	Description
d	ddown	Moves down the call stack

Examples:

`ddown` Moves down one level in the call stack. As a result, for example, **dlist** commands that follow refers to the procedure that invoked this one. The following example shows what prints after you enter this command:

```
0 check_fortran_arrays_ PC=0x10001254,
   FP=0x7fff2ed0 [arrays.F#48]
```

`d 5` Moves the current frame down five levels in the call stack.

denable

Enables action points

Format:

Enables some action points

denable *action-point-list* [**–block** *number-list*]

Enables all disabled action points in the current focus

denable –a

Arguments:

action-point-list	The identifiers of the action points being enabled.
–a	Tells TotalView to enable all action points.
–block *number-list*	If you set a breakpoint on a line that is ambiguous, this option names which instances to enable. You can obtain a list of these numbers if you use the **dactions** command.

Description:

The **denable** command reactivates action points that you previously disabled with the **ddisable** command. The **–a** option tells the CLI to enable all action points in the current focus.

If you did not save the ID values of disabled action points, you can use the **dactions** command to obtain a list of this information.

If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
en	denable	Enables action points

Examples:

`denable 3 4` Enables two previously identified action points.

`dfocus {p1 p2} denable -a`
Enables all action points associated with processes 1 and 2. This command does not affect settings associated with other processes.

`en -a` Enables all action points associated with the current focus.

`f a en -a` Enables all actions points in all processes.

`en 1 -block 3 4`
Enables the action points associated with blocks 3 and 4. That is, one logical action point can map to more than one actual action point if you set the action point at an ambiguous location.

`denable 1 2 -block 3 4`
Enables the action points associated with blocks 3 and 4 in action points 1 and 2.

dexamine

Displays memory contents

Format: **dexamine** [**–column_count** *cnt*] [**–count** *cnt*] [**–data_only**] [**–show_chars**] [**–string_length** *len*] [**–format** *fmt*] [**–memory_info**] [**–wordsize** *size*] *variable_or_expression*

Arguments:

–cols *cnt*
–column_count *cnt*
Specifies the number of columns to display. If you omit this option, the CLI determines this number of columns based on the data's word size and format.

–c *cnt*
–count *cnt*
Specifies the number of elements to examine. If you omit this option, the CLI displays the entire object. This number is determined by the object's datatype. If no type is available, the default value for *cnt* is 1 element.

–d
–data_only
Do not display memory values with a prefixed *address*: field or address annotations. This option is incompatible with –**memory_info**.

–f *fmt*
–format *fmt*
Specifies the format to use when displaying memory. The default format is **hex**. You can abbreviate each of these to the first character in the format's name.

address	Interpret memory as addresses; the word size is always the size of a pointer
binary	Binary; this can also be abbreviated to **t**
char	Unsigned character
dec	Signed decimal value of size 1, 2, 4, or 8 bytes
float	Signed float value either 4 or 8 byte word size
hex	Unsigned hexadecimal value of size 1, 2, 4, or 8 bytes
instruction	Sequence of instructions
oct	Unsigned octal value of size 1, 2, 4, or 8 bytes
string	String

–m
–memory_info
Show information about the type of memory associated with the address. If you omit the option, the CLI does not display this information. This argument is incompatible with –**data_only**. When you use this option, the CLI annotates address each line in the dump as follows:

[d]: .data
[t]: .text
[p]: .plt
[b]: .bss
[?]: Another type of memory (such as stack address)

If you have enabled memory debugging, the following annotations can also appear:

[A]: Allocated block of memory
[D]: Deallocated block of memory
[G]: Address is a guard region
[C]: Address is a corrupted guard region

If the address being examined is within an allocated block, this option tells the Memory Debugger to automatically to include the pre-guard region if the user specified guards in the memory debugging configuration.

–sc
–show_chars
: Show a trailing character dump for each line. If you do not use this option, the CLI does not show the trailing characters.

–sl *len*
–string_length *len*
: Specifies the maximum size string to display. If you do not use this option, the length is all characters up to the first null character.

–w *size*
–wordsize *size*
: Specifies the "word size" to apply to the format. The default word size is '1' for most formats. For 'address' format, the word size is always the size of a target pointer. The values can be 1, 2, 4, 8 or one of the following: **b** (byte), **h** (half word), **w** (word), or **g** (giant).

variable_or_expression
: A variable or an expression that can be resolved into a memory address.

Description: Examine memory at the address of the specified variable or the address resulting from the evaluation of an expression. If you specify an expression, the result of the evaluation must be an lvalue.

In most cases, you will enclose the expression in {} symbols.

Instead of using the listed **dexamine** *options, you can instead use the gdb examine command syntax.*

Command alias:

Alias	Definition	Description
x	dexamine	Examine (dumps) memory

dflush

Unwinds stack from suspended computations

Format:

Removes the top-most suspended expression evaluation

dflush

Removes the computation indicated by a suspended evaluation ID and all those that precede it

dflush *susp-eval-id*

Removes all suspended computations

dflush –all

Arguments:

susp-eval-id	The ID returned or thrown by the **dprint** command or which is printed by the **dwhere** command.
–all	Flushes all suspended evaluations in the current focus.

Description:

The **dflush** command unwinds the stack to eliminate frames generated by suspended computations. Typically, these frames can occur if you used the **dprint** –**nowait** command. However, this situation can occur, for example, if an error occurred in a function call in an eval point, in an expression in a **Tools** > **Evaluate** window, or if you use a **$stop** function.

You can use this command in the following ways:

- If you don't use an argument, the CLI unwinds the top-most suspended evaluation in all threads in the current focus.
- If you use a *susp-eval-id*, the CLI unwinds each stack of all threads in the current focus, flushing all pending computations up to and including the frame associated with the ID.
- If you use the –**all** option, the CLI flushes all suspended evaluations in all threads in the current focus.

If no evaluations are suspended, the CLI ignores this command. If you do not indicate a focus, the default focus is the Thread of Interest.

Examples:

The following example uses the **dprint** command to place five suspended routines on the stack. It then uses the **dflush** command to remove them. This example uses the **dflush** command in three different ways.

```
#
# Create 5 suspended functions
#
d1.<> dprint -nowait nothing2(7)
7
Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
d1.<> dprint -nowait nothing2(8)
8
Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
d1.<> dprint -nowait nothing2(9)
9
Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
```

```
d1.<> dprint -nowait nothing2(10)
10
Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
d1.<> dprint -nowait nothing2(11)
11
Thread 1.1 hit breakpoint 4 at line 310 in "nothing2(int)"
...

#
# The top of the call stack looks like:
#

d1.<> dwhere
   0 nothing2    PC=0x00012520, FP=0xffbef130 [fork.cxx#310]
   1 ***** Eval Function Call (11) ****************
   2 nothing2    PC=0x00012520, FP=0xffbef220 [fork.cxx#310]
   3 ***** Eval Function Call (10) ****************
   4 nothing2    PC=0x00012520, FP=0xffbef310 [fork.cxx#310]
   5 ***** Eval Function Call (9) ****************
   6 nothing2    PC=0x00012520, FP=0xffbef400 [fork.cxx#310]
   7 ***** Eval Function Call (8) ****************
   8 nothing2    PC=0x00012520, FP=0xffbef4f0 [fork.cxx#310]
   9 ***** Eval Function Call (7) ****************
  10 forker      PC=0x00013fd8, FP=0xffbef648 [fork.cxx#1120]
  11 fork_wrap   PC=0x00014780, FP=0xffbef6c8 [fork.cxx#1278]
    ...

#
# Use the dflush commandto remove the last item pushed
# onto the stack. Notice the frame associated with “11”
# is no longer there.
#

d1.<> dflush
d1.<> dwhere
   0 nothing2    PC=0x00012520, FP=0xffbef220 [fork.cxx#310]
   1 ***** Eval Function Call (10) ****************
   2 nothing2    PC=0x00012520, FP=0xffbef310 [fork.cxx#310]
   3 ***** Eval Function Call (9) ****************
   4 nothing2    PC=0x00012520, FP=0xffbef400 [fork.cxx#310]
   5 ***** Eval Function Call (8) ****************
   6 nothing2    PC=0x00012520, FP=0xffbef4f0 [fork.cxx#310]
   7 ***** Eval Function Call (7) ****************
   8 forker      PC=0x00013fd8, FP=0xffbef648 [fork.cxx#1120]
   9 fork_wrap   PC=0x00014780, FP=0xffbef6c8 [fork.cxx#1278]

#
# Use the dflush command with a suspened ID argument to remove
# all frames up to and including the one associated with
# suspended ID 9. This means that IDs 7 and 8 remain.
#
```

```
d1.<> dflush 9
# Top of call stack after dflush 9
d1.<> dwhere
   0 nothing2    PC=0x00012520, FP=0xffbef400 [fork.cxx#310]
   1 ***** Eval Function Call (8) ****************
   2 nothing2    PC=0x00012520, FP=0xffbef4f0 [fork.cxx#310]
   3 ***** Eval Function Call (7) ****************
   4 forker      PC=0x00013fd8, FP=0xffbef648 [fork.cxx#1120]
   5 fork_wrap   PC=0x00014780, FP=0xffbef6c8 [fork.cxx#1278]

#
# Use dflush -all to remove all frames. Only the frames
# associated with the program remain.
#

d1.<> dflush -all
# Top of call stack after dflush -all
d1.<> dwhere
   0 forker      PC=0x00013fd8, FP=0xffbef648 [fork.cxx#1120]
   1 fork_wrap   PC=0x00014780, FP=0xffbef6c8 [fork.cxx#1278]
```

dfocus

Changes the current (Process/Thread P/T) set

Format:

Changes the target of future CLI commands to this P/T set or returns the value of the current P/T set

dfocus [*p/t-set*]

Executes a command in this P/T set

dfocus *p/t-set command*

Arguments:

p/t-set	A set of processes and threads. This set defines the target upon which the CLI commands that follow will act.
command	A CLI command that operates on its own local focus.

Description:

The **dfocus** command changes the set of processes, threads, and groups upon which a command acts. This command can change the focus for all commands that follow it or just the command that immediately follows.

The **dfocus** command always expects a P/T value as its first argument. This value can either be a single arena specifier or a list of arena specifiers. The default focus is **d1.<**, which selects the first user thread. The **d** (for default) indicates that each CLI command is free to use its own default width.

If you enter an optional *command*, the focus is set temporarily, and the CLI executes the *command* in the new focus. After the *command* executes, the CLI restores focus to its original value. The *command* argument can be a single command or a list.

If you use a *command* argument, the **dfocus** command returns the result of this command's execution. If you do not enter use a *command* argument, the **dfocus** command returns the focus as a string value.

Instead of a P/T *set, you can type a* P/T *set expression. These expressions are described in* "Using P/T Set Operators" *in* Chapter 12 *of the* TotalView Users Guide.

Command alias:

Alias	Definition	Description
f	dfocus	Changes the object upon which a command acts

Examples:

`dfocus g dgo` Continues the TotalView group that contains the focus process.

`dfocus p3 {dhalt; dwhere}`
Stops process 3 and displays backtraces for each of its threads.

`dfocus 2.3` Sets the focus to thread 3 of process 2, where 2 and 3 are TotalView process and thread identifier values. The focus becomes **d2.3**.

`dfocus 3.2`
`dfocus .5`
	Sets and then resets command focus. A focus command that includes a dot and omits the process value tells the CLI to use the current process. Thus, this sequence of commands changes the focus to *process* 3, *thread* 5 (**d3.5**).

`dfocus g dstep`	Steps the current group. Although the thread of interest is determined by the current focus, this command acts on the entire group that contains that thread.
`dfocus {p2 p3} {dwhere ; dgo}`	Performs a backtrace on all threads in processes 2 and 3, and then tells these processes to execute.
`f 2.3 {f p w; f t s; g}`	Executes a backtrace (**dwhere**) on all the threads in process 2, steps thread 3 in process 2 (without running any other threads in the process), and continues the process.
`dfocus p1`	Changes the current focus to include just those threads currently in process 1. The width is set to process. The CLI sets the prompt to **p1.<**.
`dfocus a`	Changes the current set to include all threads in all processes. After you execute this command, your prompt changes to **a1.<**. This command alters CLI behavior so that actions that previously operated on a thread now apply to all threads in all processes.
`dfocus gW dstatus`	Displays the status of all worker threads in the control group. The width is group level and the target is the workers group.
`dfocus pW dstatus`	Displays the status of all worker threads in the current focus process. The width is process level and the target is the workers group.
`f {breakpoint(a) | watchpoint(a)} st`	Shows all threads that are stopped at breakpoints or watchpoints.
`f {stopped(a) - breakpoint(a)} st`	Shows all stopped threads that are not stopped at breakpoints.

Chapter 12 of the *TotalView Users Guide* contains additional **dfocus** examples.

dga

Displays Global Array variables

Format: **dga** [**–lang** *lang_type*] [*handle_or_name*] [*slice*]

Arguments:

–lang	Tells TotalView which language conventions to use when displaying information. If you omit this option, TotalView uses the language used by the thread of interest.
lang_type	Tells TotalView which language type it uses when displaying a global array. Your choices are either "**c**" or "**f**".
handle_or_name	Tells TotalView to display an array. This can either be a numeric handle or the name of the array. If you omit this argument, TotalView displays a list of all Global Arrays.
slice	Tells TotalView to only display a slice (that is, part of an array). If you are using C, you must place the array designators within braces **{}** because square brackets ([]) has special meaning in Tcl.

Description: The **dga** command tells TotalView to display information about Global Arrays.

If the focus includes more than one process, TotalView prints a list for each process in the focus. Because the arrays are global, each list is identical. If there is more than one thread in the focus, the CLI prints the value of the array as it is seen from that thread.

In almost all cases, you should change the focus to **d2.<** so that you don't include a starter process such as **prun**.

Examples:

```
dga
```

The CLI displays a list of Global Arrays; for example:

```
lb_dist
  Handle        -1000
  Ghosts        yes
  C type        $double[129][129][27]
  Fortran Type     \
             $double_precision(27,129,129)

bc_mask
  Handle        -999
  Ghosts        yes
  C type        long[129][129]
  Fortran Type  $integer(129,129)
```

```
dga bc_mask (:2,:2)
```

Displays a slice of the **bc_mask** variable; for example:

```
bc_mask(:2,:2) = {
   (1,1) = 1 (0x00000001)
   (2,1) = 1 (0x00000001)
   (1,2) = 1 (0x00000001)
   (2,2) = 0 (0x00000000)
}
```

```
dga -lang c -998 {[:1]{:1]}
```

Displays the same **bc_mask** variable as in the previous example in C format. In this case, the command refers to the variable to by its handle.

dgo

Resumes execution of processes

Format:	**dgo**	
Arguments:	–back	(ReplayEngine only) Runs the non-held process in the current focus backward until it hits some action point or the beginning of recorded Replay history. This option can be abbreviated to **–b.**

Description: The **dgo** command tells all nonheld processes and threads in the current focus to resume execution. If the process does not exist, this command creates it, passing it the default command arguments. These can be arguments passed into the CLI, or they can be the arguments set with the **drerun** command. If you are also using the TotalView GUI, you can set this value by using the **Process > Startup Parameters** command.

If a process or thread is held, it ignores this command.

You cannot use a **dgo** command when you are debugging a core file, nor can you use it before the CLI loads an executable and starts executing it.

If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
g	dgo	Resumes execution
G	{dfocus g dgo}	Resume group

Examples:

`dgo`	Resumes execution of all stopped threads that are not held and which belong to processes in the current focus. (Threads held at barriers are not affected.)
`G`	Resumes execution of all threads in the current control group.
`f p g`	Continues the current process. Only threads that are not held can run.
`f g g`	Continues all processes in the control group. Only processes and threads that are not held are allowed to run.
`f gL g`	Continues all threads in the share group that are at the same PC as the thread of interest.
`f pL g`	Continues all threads in the current process that are at the same PC as the thread of interest.
`f t g`	Continues a single thread.

dgroups

Manipulates and manages groups

Format:

Adds members to thread and process groups

dgroups –add [**–g** *gid*] [*id-list*]

Deletes groups

dgroups –delete [**–g** *gid*]

Intersects a group with a list of processes and threads

dgroups –intersect [**–g** *gid*] [*id-list*]

Prints process and thread group information

dgroups [**–list**] [*pattern-list*]

Creates a new thread or process group

dgroups –new [*thread_or_process*] [**–g** *gid*] [*id-list*]

Removes members from thread or process groups

dgroups –remove [**–g** *gid*] [*id-list*]

Arguments:

–g *gid*	The group ID upon which the command operates. The *gid* value can be an existing numeric group ID, an existing group name, or, if you are using the **–new** option, a new group name.
id-list	A Tcl list that contains process and thread IDs. Process IDs are integers; for example, 2 indicates process 2. Thread IDs define a *pid.tid* pair and look like decimal numbers; for example, 2.3 indicates process 2, thread 3. If the first element of this list is a group tag, such as the word **control**, the CLI ignores it. This makes it easy to insert all members of an existing group as the items to be used in any of these operations. (See the **dset** command's discussion of the **GROUP(gid)** variable for information on group designators.) These words appear in some circumstances when the CLI returns lists of elements in P/T sets.
pattern-list	A pattern to be matched against group names. The pattern is a Tcl regular expression.
thread_or_process	Keywords that tell the CLI to create a new process or thread group. You can specify one of the following arguments: **t**, **thread**, **p**, or **process**.

Description:

The **dgroups** command lets you perform the following functions:

- Add members to process and thread groups.
- Create a group.
- Intersect a group with a set of processes and threads.
- Delete groups.
- Display the name and contents of groups.
- Remove members from a group.

dgroups –add

The **dgroups –add** command adds members to one or more thread or process groups. The CLI adds each of these threads and processes to the group. If you add a:

- Process to a thread group, The CLI adds all of its threads.
- Thread to a process group, The CLI adds the thread's parent process.

You can abbreviate the –**add** option to –**a**.

The CLI returns the ID of this group.

You can explicitly name the items being added by using an *id-list* argument. If you do not use an *id-list* argument, the CLI adds the threads and processes in the current focus. Similarly, you can name the group to which the CLI adds members if you use the –**g** option. If you omit the –**g** option, the CLI uses the groups in the current focus.

If the *id-list* argument contains processes, and the target is a thread group, the CLI adds all threads from these processes. If it contains threads and the target is a process group, the CLI adds the parent process for each thread.

*If you specify an id-list argument and you also use the –***g** *option, the* CLI *ignores the focus. You can use two* **dgroups -add** *commands instead.*

If you try to add the same object more than once to a group, the CLI only adds it once.

The CLI does not let you use this command to add a process to a control group. If you need to perform this operation, you can use the **CGROUP(dpid)** variable; for example:

```
dset CGROUP($mypid) $new_group_id
```

dgroups –delete

The **dgroups –delete** command deletes the target group. You can only delete groups that you create; you cannot delete groups that TotalView creates.

dgroups –intersect

The **dgroups –intersect** command intersects a group with a set of processes and threads. If you intersect a thread group with a process, the CLI includes all of the process's threads. If you intersect a process group with a thread, the CLI uses the thread's process.

After this command executes, the group no longer contains members that were not in this intersection.

You can abbreviate the –**intersect** option to –**i**.

dgroups –list

The **dgroups –list** command prints the name and contents of process and thread groups. If you specify a *pattern-list* as an argument, the CLI only prints information about groups whose names match this pattern. When entering a list, you can specify a *pattern*. The CLI matches this pattern against the list of group names by using the Tcl **regex** command.

If you do not enter a pattern, the CLI *only displays groups that you have created which have nonnumeric names.*

The CLI returns information from this command; the information is not returned.

You can abbreviate –**list** to –**l**.

dgroups –new

The **dgroups** –**new** command creates a new thread or process group and adds threads and processes to it. If you use a name with the –**g** option, the CLI uses that name for the group ID; otherwise, it assigns a new numeric ID. If the group you name already exists, the CLI replaces it with the newly created group.

The CLI returns the ID of the newly created group.

You can explicitly name the items being added by using an *id-list* argument. If you do not use an *id-list* argument, the CLI adds the threads and processes in the current focus.

If the *id-list* argument contains processes, and the target is a thread group, the CLI adds all threads from these processes. If it contains threads and the target is a process group, TotalView adds the parent process for each thread.

If you use an id-list argument and also use the –**g** *option, the* CLI *ignores the focus.You can use two* **dgroups -add** *commands instead.*

If you are adding more than one object and one of these objects is a duplicate, The CLI adds the nonduplicate objects to the group.

You can abbreviate the –**new** option to –**n**.

dgroups –remove

The **dgroups** –**remove** command removes members from one or more thread or process groups. If you remove a process from a thread group, The CLI removes all of its threads. If remove a thread from a process group, The CLI removes its parent process.

You cannot remove processes from a control group. You can, however, move a process from one control group to another by using the **dset** command to assign it to the **CGROUP(dpid)** variable group.

Also, you cannot use this command on read-only groups, such as share groups.

You can abbreviate the –**remove** option to –**r**.

Command alias:

Alias	Definition	Description
gr	dgroups	Manipulates a group

Examples:

dgroups –add

`f tW gr -add`	Adds the focus thread to its workers group.
`dgroups -add`	Adds the current focus thread to the current focus group.

```
set gid [dgroups -new thread ($CGROUP(1))]
```

Creates a new thread group that contains all threads from all processes in the control group for process 1.

```
f $a_group/9 dgroups -add
```

Adds process 9 to a user-defined group.

dgroups –delete

```
gr -delete -g mygroup
```

Deletes the **mygroup** group.

dgroups –intersect

```
dgroups -intersect -g 3 3.2
```

Intersects thread 3.2 with group 3. If group 3 is a thread group, this command removes all threads except 3.2 from the group; if it is a process group, this command removes all processes except process 3 from it.

`f tW gr -i`	Intersects the focus thread with the threads in its workers group.

```
f gW gr -i -g mygroup
```

Removes all nonworker threads from the **mygroup** group.

dgroups –list

`dgroups -list` Tells the CLI to display information about all named groups; for example:

```
ODD_P: {process 1 3}
EVEN_P: {process 2 4}
```

`gr -l *` Tells the CLI to display information about groups in the current focus.

```
1: {control 1 2 3 4}
2: {workers 1.1 1.2 1.3 1.4 2.1 2.2 2.3
2.4 3.1
   3.2 3.3 3.4 4.1 4.2 4.3 4.4}
3: {share 1 2 3 4}
ODD_P: {process 1 3}
EVEN_P: {process 2 4}
```

dgroups –new

`gr -n t -g mygroup $GROUP($CGROUP(1))`

Creates a new thread group named **mygroup** that contains all threads from all processes in the control group for process 1.

`set mygroup [dgroups -new]`

Creates a new process group that contains the current focus process.

dgroups –remove

`dgroups -remove -g 3 3.2`

Removes thread 3.2 from group 3.

`f W dgroups -add`

Marks the current thread as being a worker thread.

`f W dgroups -r`

Indicates that the current thread is not a worker thread.

dhalt

Suspends execution of processes

Format: dhalt

Arguments: This command has no arguments

Description: The **dhalt** command stops all processes and threads in the current focus.

If you do not indicate a focus, the default focus is the Process of Interest.

Command alias:

Alias	Definition	Description
h	dhalt	Suspends execution
H	{dfocus g dhalt}	Suspends group execution

Examples:

`dhalt`	Suspends execution of all *running* threads belonging to processes in the current focus. (This command does not affect threads held at barriers.)
`f t 1.1 h`	Suspends execution of thread 1 in process 1. Note the difference between this command and **f 1.< dhalt**. If the focus is set as thread level, this command halts the first user thread, which is probably thread 1.

dheap

Controls heap debugging

Format: Shows Memory Debugger state

dheap [**–status**]

Applies a saved configuration file

dheap –apply_config { **default** | *filename* }

Shows information about a backtrace

dheap –backtrace [*subcommands*]

Compares memory states

dheap –compare *subcommands* [*optional_subcommands*]
[*process* | *filename* [*process* | *filename*]]

Enables or disables the Memory Debugger

dheap { **–enable** | **–disable** }

Enables or disables event notification

dheap –event_filter *subcommands*

Writes memory information

dheap –export *subcommands*

Specifies which filters the Memory Debugger uses

dheap –filter *subcommands*

Writes guard blocks (memory before and after an allocation)

dheap –guard [*subcommands*]

Enables or disables the retaining (hoarding) of freed memory blocks

dheap –hoard [*subcommands*]

Displays Memory Debugger information

dheap –info [*subcommands*]

Indicates whether an address is in a deallocated block

dheap –is_dangling *address*

Locates memory leaks

dheap –leaks [**–check_interior**]

Enables or disables Memory Debugger event notification

dheap –[no]notify

Paints memory with a distinct pattern

dheap –paint [*subcommands*]

Enables or disables the ability to catch bounds errors and use-after-free errors retaining freed memory blocks

dheap –red_zones [*subcommands*]

Enables or disables allocation and reallocation notification

dheap –tag_alloc *subcommand* [*start_address* [*end_address*]]

Displays the Memory Debugger version number

dheap –version

Description: The **dheap** command is described in the B*atch Scripting* section of Locating Memory Problems in the MemoryScape documentation.

dhistory

Performs actions upon ReplayEngine

Format: **dhistory [–info] [–get_time] [–go_time *time*] [–go_live] [–enable] [–disable]**

Arguments:

–info	Writes ReplayEngine information including the current time, the live time and if the process is in replay or record mode. If you do not enter any arguments, **–info** is the default.
–get_time	Displays the current time. The output of this command shows an integer value followed by an address. The first integer value is a virtual timestamp. This virtual timestamp does not refer to the exact point in time; it has a granularity that is typically a few lines of code. The address value is a PC value that corresponds to a precise point within that block of code.
–go_time *time*	Places the process back to the virtual time specified by the *time* integer argument. The *time* argument is a virtual timestamp as reported by **dhistory -get_time**. You cannot use this command to move to a specific instruction but you can use it to get to within a small block of code (usually within a few lines of your intended point in execution history). This command is typically used either for roughly bookmarking a point in a code or for searching execution history. It may need to be combined with stepping and **duntil** commands to return to an exact position.
–go_live	Places the process back to the PC and back into record mode. You can resume your "regular" debugging session.
–enable	Enables the ReplayEngine so that it will be active when you next restart the program.
–disable	Disables the ReplayEngine so that it will be inactive when you next restart the program.

Description: The **dhistory** command displays information about the current process either by default or when you use the **–info** argument. In addition, options to this command let you obtain a debugging time, which can be stored in a variable and go back to a time.

In addition, you can enable and display the ReplayEngine as well as put it back into regular debugging mode using the **–go_live** option. You'll need to do this after your program is placed into replay mode. This occurs whenever you use any GUI or CLI command that moves to replay mode. For example, in the CLI, this can occur when you execute such commands as **dnext** or **dout**.

Examples:

dhistory	Typing this command displays the following information:

```
History info for process 1
     Live time: 421 0x80485d6
  Current time: 421 0x80485d6
       Live PC: 0x80485d6
   Record Mode: True
 Replay Wanted: True
   Stop Reason: Normal result [waitpid, search, or goto_time]
Temp directory: /tmp/replay_jsm_local/replay_session_pZikY9
Event log mode: circular
Event log size: 268435456
```

dhold

Holds threads or processes

Format:

Holds processes

dhold –process

Holds threads

dhold –thread

Arguments:

–process	Holds processes in the current focus. You can abbreviate the **–process** option to **–p**.
–thread	Holds threads in the current focus. You can abbreviate the **–thread** option to **–t**.

Description:

The **dhold** command holds the threads and processes in the current focus.

You cannot hold system manager threads. In all cases, holding threads that aren't part of your program always involves some risk.

Command alias:

Alias	Definition	Description
hp	{dhold –process}	Holds the focus process
HP	{f g dhold –process}	Holds all processes in the focus group
ht	{f t dhold –thread}	Holds the focus thread
HT	{f g dhold –thread}	Holds all threads in the focus group
htp	{f p dhold –thread}	Holds all threads in the focus process

Examples:

`f W HT`	Holds all worker threads in the focus group.
`f s HP`	Holds all processes in the share group.
`f $mygroup/ HP`	Holds all processes in the group identified by the contents of **mygroup**.

dkill

Terminates execution of processes

Format: dkill [–remove]

Arguments:

–remove	Tells TotalView that it should remove all knowledge of the process from its internal tables. If you are using TotalView Team, this frees a token so that you can reuse it.

Description: The **dkill** command terminates all processes in the current focus.

Because the executables associated with the defined processes are still loaded, using the **drun** command restarts the processes.

The **dkill** command alters program state by terminating all processes in the affected set. In addition, TotalView destroys any spawned processes when the process that created them is killed. The **drun** command can only restart the initial process.

If you do not indicate a focus, the default focus is the Process of Interest. If, however, you kill the primary process for a control group, all of the slave processes are killed.

Command alias:

Alias	Definition	Description
k	dkill	Terminates a process's execution

Examples:

`dkill`

Terminates all threads belonging to processes in the current focus.

`dfocus {p1 p3} dkill`

Terminates all threads belonging to processes 1 and 3.

dlappend

Appends list elements to a TotalView variable

Format: **dlappend** *variable-name value* [...]

Arguments:

variable-name	The variable to which values are appended.
value	The values to append.

Description: The **dlappend** command appends list elements to a TotalView variable. The **dlappend** command performs the same function as the Tcl **lappend** command, differing in that **dlappend** does not create a new debugger variable. That is, the following Tcl command creates a variable named **foo**:

```
lappend foo 1 3 5
```

In contrast, the following command displays an error message:

```
dlappend foo 1 3 5
```

Examples:

```
dlappend TV::process_load_callbacks my_load_callback
```

Adds the **my_load_callback** function to the list of functions in the **TV::process_load_callbacks** variable.

dlist

Displays source code lines

Format:

Displays source code relative to the current list location

dlist [**–n** *num-lines*]

Displays source code relative to a named place

dlist *breakpoint-expr* [**–n** *num-lines*]

Displays source code relative to the current execution location

dlist –e [**–n** *num-lines*]

Arguments:

–n *num-lines*	Displays this number of lines rather than the default number. (The default is the value of the **MAX_LIST** variable.) If *num-lines* is negative, the CLI displays lines before the current location, and additional **dlist** commands show preceding lines in the file rather than succeeding lines. This option also sets the value of the **MAX_LIST** variable to *num-lines*.
breakpoint-expr	The location at which the CLI begins displaying information. In most cases, you will specify this location as a line number or as a string that contains a file name, function name, and line number, each separated by **#** characters; for example: **file#func#line**. (For more information, see "*Qualifying Symbol Names*" in Chapter 12 of the TotalView *Users Guide*.) The CLI creates defaults if you omit parts of this specification. If you enter a different file, this will be the file that the CLI uses for future displays. This means that if you want to display information relative to the current thread's execution point, use the **–e** option to **dlist**. If the breakpoint expression evaluates to more than one location, TotalView chooses one. For other ways to enter these expressions, see "*Breakpoint Expressions*" on page 32. If what you enter names more than one address, TotalView picks one.
–e	Sets the display location to include the current execution point of the thread of interest. If you use **dup** and **ddown** commands to select a buried stack frame, this location includes the PC (program counter) for that stack frame.

Description:

The **dlist** command displays source code lines relative to a place in the source code. (This position is called the *list location*.) The CLI prints this information; it is not returned. If you do not specify *source-loc* or **–e**, the command continues where the previous list command stopped. To display the thread's execution point, use the **dlist –e** command.

If you enter a file or procedure name, the listing begins at the file or procedure's first line.

The default focus for this command is thread level. If your focus is at process level, TotalView acts on each thread in the process.

The first time you use the **dlist** command after you focus on a different thread—or after the focus thread runs and stops again—the location changes to include the current execution point of the new focus thread.

The CLI expands tabs in the source file as blanks in the output. The **TAB_WIDTH** variable controls the tab stop width, which has a default value of 8. If **TAB_WIDTH** is set to –1, no tab processing is done, and the CLI displays tabs using their ASCII value.

All lines appear with a line number and the source text for the line. The following symbols are also used:

@	An action point is set at this line.
>	The PC for the current stack frame is at the indicated line and this is the leaf frame.
=	The PC for the current stack frame is at the indicated line and this is a buried frame; this frame has called another function so that this frame is not the active frame.

These correspond to the marks shown in the backtrace displayed by the **dwhere** command that indicates the selected frame.

Here are some general rules:

- The initial display location is **main()**.
- The CLI sets the display location to the current execution location when the focus is on a different thread.

If the *source-loc* argument is not fully qualified, the CLI looks for it in the directories named in the CLI **EXECUTABLE_PATH** variable.

Command alias:

Alias	Definition	Description
l	dlist	Displays lines

Examples: The following examples assume that the **MAX_LIST** variables equals 20, which is its initial value.

`dlist`	Displays 20 lines of source code, beginning at the current list location. The list location is incremented by 20 when the command completes.
`dlist 10`	Displays 20 lines, starting with line 10 of the file that corresponds to the current list location. Because this uses an explicit value, the CLI ignores the previous command. The list location is changed to line 30.
`dlist -n 10`	Displays 10 lines, starting with the current list location. The value of the list location is incremented by 10.
`dlist -n -50`	Displays source code preceding the current list location; shows 50 lines, ending with the current source code location. The list location is decremented by 50.
`dlist do_it`	Displays 20 lines in procedure **do_it**. Changes the list location so that it is the twentieth line of the procedure.

`dfocus 2.< dlist do_it`
: Displays 20 lines in the **do_it** routine associated with process 2. If the current source file is named **foo**, you can also specify this as **dlist foo#do_it**, naming the executable for process 2.

`dlist -e`
: Displays 20 lines starting 10 lines above the current execution location.

`f 1.2 l -e`
: Lists the lines around the current execution location of thread 2 in process 1.

`dfocus 1.2 dlist -e -n 10`
: Produces essentially the same listing as the previous example, differing in that it displays 10 lines.

`dlist do_it.f#80 -n 10`
: Displays 10 lines, starting with line 80 in file **do_it.f**. Updates the list location to line 90.

dload

Loads debugging information

Format: dload [–g *gid*] [–r *hname*]
[{ –np | –procs | –tasks } num]
[–nodes num]
[–replay | –no_replay]
[–mpi *starter*]
[–starter_args *argument*]
[–env *variable*=*value*] ...
[–e] *executable*
[–parallel_attach_subset *subset_specification*]

Arguments:

–g *gid*	Sets the control group for the process being added to the group ID specified by *gid*. This group must already exist. (The CLI **GROUPS** variable contains a list of all groups.)
–r *hname*	The host on which the process will run. The CLI launches a TotalView Debugger Server on the host machine if one is not already running there. (See Chapters 5 and 6 in the *TotalView Users Guide* for information on the server launch commands.)
{ –np | –procs | –tasks } *num*	Indicates the number of processes or tasks that the starter program will create.
–nodes *num*	Indicates the number of nodes upon which your program will execute.
–replay | –no_replay	These options enable and disable the ReplayEngine the next time the program is restarted.
–mpi *starter*	Names the starter configuration. This can be the name of a configuration supplied by TotalView Technologies or one that you have created and have named using the **TV::parallel_configs** variable. For information on defining configurations, see Appendix A , "MPI S*tartup*," on page 319.
–starter_args *argument*	Indicates additional arguments that are passed to the starter program.
–env *variable*=*value*	Sets a variable that is added to the program's environment.
–e	Indicates that the next argument is a file name. You need to use this argument if the file name begins with a dash or only uses numeric characters.
executable	A fully or partially qualified file name for the file corresponding to the program.
parallel_attach_subset *subset_specification*	Defines a list of MPI ranks to be attached to when an MPI job is created or attached to. The list is space-separated; each element can have one of 3 forms:

rank: specifies that rank only

rank1-rank2: specifies all ranks between rank1 and rank2, inclusive

rank1-rank2:stride: specifies every strideth rank between rank1 and rank2

A rank specification can be either a number or "**$max**" which represents the last rank in the MPI job.

For example:

```
dload -parallel_attach_subset {1 2 4-6 7-
$max:2} mpirun
```

will attach to ranks 1, 2, 4, 5, 6, 7, 9, 11, 13,...

Description: The **dload** command creates a new TotalView process object for the *executable* file. The **dload** command returns the TotalView ID for the new object.

Your license limits the number of processes that you can run at the same time. For example, the maximum number of processes for TotalView Individual is 16. As some systems and run time environments create threads to manage a process, you may not be able to get this many processes running at the same time. (Only TotalView Individual counts threads against your license. TotalView Enterprise and Team allows an unlimited number of threads to be running at the same time.)

Command alias:

Alias	Definition	Description
lo	dload	Loads debugging information

Examples:

```
dload do_this
```

Loads the debugging information for the **do_this** executable into the CLI. After this command completes, the process does not yet exist and no address space or memory is allocated to it.

```
dload -mpi POE -starter_args "hfile=~/my_hosts" \
       -np 2 -nodes
```

Loads an MPI job using the POE configuration. Two processes will be used across nodes. The **hfiles** starter argument is used.

```
lo -g 3 -r other_computer do_this
```

Loads the debugging information for the **do_this** executable that is executing on the **other_computer** machine into the CLI. This process is placed into group 3.

```
f g3 lo -r other_computer do_this
```

Does not do what you would expect it to do because the **dload** command ignores the focus command. Instead, this does exactly the same thing as the previous example.

```
dload -g $CGROUP(2) -r slowhost foo
```

Loads another process based on image **foo** on machine **slowhost**. The CLI places this process in the same group as process 2.

```
dload -env DISPLAY=aurora:0.0
            -env STARTER=~/starter myprog
```

Loads another process based on image **foo** on machine **slowhost**. The CLI places this process in the same group as process 2.

dmstat

Displays memory use information

Format: dmstat

Arguments: This command has no arguments

Description: The **dmstat** command displays information about how your program is using memory. The CLI returns memory information in three parts, as follows:

- **Memory usage summary**: Indicates the minimum and maximum amounts of memory used by the text and data segments, the heap, and the stack, as well as the virtual memory stack usage and the virtual memory size.
- **Individual process statistics**: Shows the amount of memory that each process is currently using.
- **Image information:** Lists the name of the image, the image's text size, the image's data size, and the set of processes using the image.

The following table contains definitions of the columns the CLI displays:

Column	Description
text	The amount of memory used to store your program's machine code instructions. The text segment is sometimes called the code segment.
data	The amount of memory used to store initialized and uninitialized data.
heap	The amount of memory currently being used for data created at run time; for example, calls to the **malloc()** function allocate space on the heap while the **free()** function releases it.
stack	The amount of memory used by the currently executing routine and all of the routines in its backtrace. If this is a multithreaded process, TotalView only shows information for the main thread's stack. Note that the stacks of other threads might not change over time on some architectures. On some systems, the space allocated for a thread is considered part of the heap. For example, if your main routine invokes function **foo()**, the stack contains two groups of information—these groups are called frames. The first frame contains the information required for the execution of your main routine, and the second, which is the current frame, contains the information needed by the **foo()** function. If **foo()** invokes the **bar()** function, the stack contains three frames. When **foo()** finishes executing, the stack only contains one frame.

Column	Description
stack_vm	The logical size of the stack is the difference between the current value of the stack pointer and the address from which the stack originally grew. This value can differ from the size of the virtual memory mapping in which the stack resides. For example, the mapping can be larger than the logical size of the stack if the process previously had a deeper nesting of procedure calls or made memory allocations on the stack, or it can be smaller if the stack pointer has advanced but the intermediate memory has not been touched. The **stack_vm** value is this size difference.
vm_size	The sum of the sizes of the mappings in the process's address space.

Examples: `dmstat` **dmstat** is sensitive to the focus, as this example from a four-process program shows:

```
process: text      data     heap  stack  [stack_vm]   vm_size
   1    (9271): 1128.54K   16.15M   9976       10432   [16384]

image information:
                     image_name       text         data  dpids
 ....ry/forked_mem_exampleLINUX       2524     16778479  1
      /lib/i686/libpthread.so.0      32172        27948  1
            /lib/i686/libc.so.6    1050688       122338  1
             /lib/ld-linux.so.2      70240        10813  1
```

`dfocus a dmstat`

The CLI prints the following for a four-process program:

```
     process:     text    data  heap  stack [stack_vm]  vm_size
 1     (9979): 1128.54K 16.15M 14072 273168 [   278528]  17.69M
 5     (9982): 1128.54K 16.15M  9976  10944 [    16384]  17.44M
 6     (9983): 1128.54K 16.15M  9976  10944 [    16384]  17.44M
 7     (9984): 1128.54K 16.15M  9976  10944 [    16384]  17.44M

maximum:
 1     (9979): 1128.54K 16.15M 14072 273168 [   278528]  17.69M
minimum
 5     (9982): 1128.54K 16.15M  9976  10944 [    16384]  17.44M

image information:
                    image_name      text         data  dpids
....ry/forked_mem_exampleLINUX      2524     16778479  1 5 6 7
     /lib/i686/libpthread.so.0     32172        27948  1 5 6 7
           /lib/i686/libc.so.6   1050688       122338  1 5 6 7
            /lib/ld-linux.so.2     70240        10813  1 5 6 7
```

dnext

Steps source lines, stepping over subroutines

Format: dnext [–back] [*num-steps*]

Arguments:

–back	(ReplayEngine only) Steps to the previous source line, stepping over subroutines. This option can be abbreviated to **–b.**
num-steps	An integer greater than 0, indicating the number of source lines to be executed.

Description: The **dnext** command executes source lines; that is, it advances the program by steps (source line statements). However, if a statement in a source line invokes a routine, the **dnext** command executes the routine as if it were one statement; that is, it steps *over* the call.

The optional *num-steps* argument tells the CLI how many **dnext** operations to perform. If you do not specify *num-steps*, the default is 1.

The **dnext** command iterates over the arenas in its focus set, performing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is **process** (**p**).

For more information on stepping in processes and threads, see **dstep** on page 100.

Command alias:

Alias	Definition	Description
n	dnext	Runs the thread of interest one statement while allowing other threads in the process to run.
N	{dfocus g dnext}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and steps one such aligned thread in each member one statement. The rest of the control group runs freely.
nl	{dfocus L dnext}	Steps the process threads in lockstep. This steps the thread of interest one statement and runs all threads in the process that are at the same PC as the thread of interest to the same statement. Other threads in the process run freely. The group of threads that is at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.

Alias	Definition	Description
NL	{dfocus gL dnext}	Steps lockstep threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one statement. Other threads in the control group run freely.
nw	{dfocus W dnext}	Steps worker threads in the process. This steps the thread of interest one statement, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
NW	{dfocus gW dnext}	Steps worker threads in the group. This steps the thread of interest one statement, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

`dnext`	Steps one source line.
`n 10`	Steps ten source lines.
`N`	Steps one source line. It also runs all other processes in the group that are in the same lockstep group to the same line.
`f t n`	Steps the thread one statement.
`dfocus 3. dnext`	Steps process 3 one step.

dnexti

Steps machine instructions, stepping over subroutines

Format: dnexti [–back] [*num-steps*]

Arguments:

–back	(ReplayEngine only) Steps a machine instruction back to the previous instruction, stepping over subroutines. This option can be abbreviated to **–b.**
num-steps	An integer greater than 0, indicating the number of instructions to be executed.

Description: The **dnexti** command executes machine-level instructions; that is, it advances the program by a single instruction. However, if the instruction invokes a subfunction, the **dnexti** command executes the subfunction as if it were one instruction; that is, it steps *over* the call. This command steps the thread of interest while allowing other threads in the process to run.

The optional *num-steps* argument tells the CLI how many **dnexti** operations to perform. If you do not specify *num-steps*, the default is 1.

The **dnexti** command iterates over the arenas in the focus set, performing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is **process** (**p**).

For more information on stepping in processes and threads, see **dstep** on page 100.

Command alias:

Alias	Definition	Description
ni	dnexti	Runs the thread of interest one instruction while allowing other threads in the process to run.
NI	{dfocus g dnexti}	A group stepping command. This searches for threads in the share group that are at the same PC as the thread of interest, and steps one such aligned thread in each member one instruction. The rest of the control group runs freely.
nil	{dfocus L dnexti}	Steps the process threads in lockstep. This steps the thread of interest one instruction, and runs all threads in the process that are at the same PC as the thread of interest to the same statement. Other threads in the process run freely. The group of threads that is at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.

Alias	Definition	Description
NIL	{dfocus gL dnexti}	Steps lockstep threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one instruction. Other threads in the control group run freely.
niw	{dfocus W dnexti}	Steps worker threads in the process. This steps the thread of interest one instruction, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
NIW	{dfocus gW dnexti}	Steps worker threads in the group. This steps the thread of interest one instruction, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

`dnexti`	Steps one machine-level instruction.
`ni 10`	Steps ten machine-level instructions.
`NI`	Steps one instruction and runs all other processes in the group that were executing at that instruction to the next instruction.
`f t n`	Steps the thread one machine-level instruction.
`dfocus 3. dnexti`	Steps process 3 one machine-level instruction.

dout

Executes until just after the place that called the current routine

Format: dout [–back] [*frame-count*]

Arguments:

–back	(ReplayEngine only) Returns to the function call that placed the PC into the current routine. This option can be abbreviated to **–b.**
frame-count	An integer that specifies that the thread returns out of this many levels of subroutine calls. If you omit this number, the thread returns from the current level.

Description: The **dout** command runs a thread until it returns from either of the following:

- The current subroutine.
- One or more nested subroutines.

When you specify process width, TotalView allows all threads in the process that are not running to this goal to run free. (Specifying process width is the default.)

Command alias:

Alias	Definition	Description
ou	dout	Runs the thread of interest out of the current function, while allowing other threads in the process to run.
OU	{dfocus g dout}	Searches for threads in the share group that are at the same PC as the thread of interest, and runs one such aligned thread in each member out of the current function. The rest of the control group runs freely. This is a group stepping command.
oul	{dfocus L dout}	Runs the process threads in lockstep. This runs the thread of interest out of the current function, and also runs all threads in the process that are at the same PC as the thread of interest out of the current function. Other threads in the process run freely. The group of threads that is at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
OUL	{dfocus gL dout}	Runs lockstep threads in the group. This runs all threads in the share group that are at the same PC as the thread of interest out of the current function. Other threads in the control group run freely.

Alias	Definition	Description
ouw	{dfocus W dout}	Runs worker threads in the process. This runs the thread of interest out of the current function and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
OUW	{dfocus gW dout}	Runs worker threads in the group. This runs the thread of interest out of the current function and also runs all worker threads in the same share group out of the current function. All other threads in the control group run freely.

For additional information on the different kinds of stepping, see the **dstep** on page 100 command information.

Examples:

`f t ou`	Runs the current thread of interest out of the current subroutine.
`f p dout 3`	Unwinds the process in the current focus out of the current subroutine to the routine three levels above it in the call stack.

dprint

Evaluates and displays information

Format: Prints the value of a variable

dprint [–nowait] [–slice "*slice_expr***"]** *variable*

Prints the value of an expression

dprint [–nowait] [–slice "*slice_expr***"]** *expression*

Arguments:

–nowait	Tells TotalView to evaluate the expression in the background. You need to use **TV::expr** to obtain the results, as they are not displayed.
–slice "*slice_expr***"**	Defines an array slice—that is, a portion of the array—that will be printed. If the programming language is C or C++, you will need to use a backquote (\) when you enter the array subscripts. For example, **"\[100:110\]"**.
variable	A variable whose value is displayed. The variable can be local to the current stack frame or it can be global. If the variable being displayed is an array, you can qualify the variable's name with a slice that tells the CLI to display a portion of the array,
expression	A source-language expression to evaluate and print. Because *expression* must also conform to Tcl syntax, you must enclose it within quotation marks it if it includes any blanks, and in braces (**{}**) if it includes brackets ([ ]), dollar signs (**$**), quotation marks ("), or other Tcl special characters.

Description: The **dprint** command evaluates and displays a variable or an expression. The CLI interprets the expression by looking up the values associated with each symbol and applying the operators. The result of an expression can be a scalar value or an aggregate (array, array slice, or structure).

If an event such as a **$stop**, SEGV, breakpoint occurs, the **dprint** command throws an exception that describes the event. The first exception subcode returned by **TV::errorCodes** is the *susp-eval-id* (a suspension-evaluation-ID). You can use this to manipulate suspended evaluations with the **dflush** and **TV::expr** commands. For example:

```
dfocus tdpid.dtid TV::expr get susp-eval-id
```

If the expression calls a function, the focus must not specify more than one thread for each process.

If you use the **–nowait** option, TotalView evaluates the expression in the background. It also returns a *susp-eval-id* that you can use to obtain the results of the evaluation using **TV::expr**.

As the CLI displays data, it passes the data through a simple *more* processor that prompts you after it displays each screen of text. At this time, you can press the Enter key to tell the CLI to continue displaying information. Entering **q** tells the CLI to stop printing this information.

Since the **dprint** command can generate a considerable amount of output, you might want to use the **capture** command described on page 18 to save the output to a variable.

Structure output appears with one field printed per line; for example:

```
sbfo = {
   f3 = 0x03 (3)
   f4 = 0x04 (4)
   f5 = 0x05 (5)
   f20 = 0x000014 (20)
   f32 = 0x00000020 (32)
}
```

Arrays print in a similar manner; for example:

```
foo = {
   [0][0] = 0x00000000 (0)
   [0][1] = 0x00000004 (4)
   [1][0] = 0x00000001 (1)
   [1][1] = 0x00000005 (5)
   [2][0] = 0x00000002 (2)
   [2][1] = 0x00000006 (6)
   [3][0] = 0x00000003 (3)
   [3][1] = 0x00000007 (7)
}
```

You can append a slice to the variable's name to tell the CLI to display a portion of an array; for example:

```
d.1<> p -slice "\[10:20\]" random
random slice:(10:30) = {
   (10) = 0.479426
   (11) = 0.877583
   (12) = 0.564642
   (13) = 0.825336
   (14) = 0.644218
   (15) = 0.764842
   (16) = 0.717356
   (17) = 0.696707
   (18) = 0.783327
   (19) = 0.62161
    (20) = 0.841471
}
```

The following is an another way of specifying the same slice:

```
d.1<> set my_var \[10:20\]
d.1<> p -slice $my_var random
random slice:(10:30) = {
```

The CLI evaluates the expression or variable in the context of each thread in the target focus. Thus, the overall format of **dprint** output is as follows:

```
first process or thread:
   expression result

second process or thread:
   expression result
```

```
...

last process or thread:
   expression result
```

TotalView lets you cast variables and cast a variable to an array. If you are casting a variable, the first array address is the address of the variable. For example, assume the following declaration:

```
float bint;
```

The following statement displays the variable as an array of one integer:

```
dprint {(int \[1\])bint:
```

If the expression is a pointer, the first addresses is the value of the pointer. Here is an array declaration:

```
float bing[2], *bp = bint;
```

TotalView assumes the first array address is the address of what **bp** is pointing to. So, the following command displays the array:

```
dprint {(int \[2\])bp}
```

You can also use the **dprint** command to obtain values for your computer's registers. For example, on most architectures, **$r1** is register 1. To obtain the contents of this register, type:

```
dprint \$r1
```

You must precede the dollar sign (**$**) with a backslash to escape it since the register's name includes the **$**. This **$** is not the standard indicator that tells Tcl to fetch a variable's value. Appendix A , *"Architectures,"* on page 287 lists the mnemonic names assigned to registers.

Do not use a $ when asking the **dprint** *command to display your program's variables.*

Command alias:

Alias	Definition	Description
p	dprint	Evaluates and displays information

Examples:

`dprint scalar_y`
Displays the values of variable **scalar_y** in all processes and threads in the current focus.

`p argc`
Displays the value of **argc**.

`p argv`
Displays the value of **argv**, along with the first string to which it points.

`p {argv[argc-1]}`
Prints the value of **argv[argc-1]**. If the execution point is in **main()**, this is the last argument passed to **main()**.

`dfocus p1 dprint scalar_y`
Displays the values of variable **scalar_y** for the threads in process 1.

`f 1.2 p arrayx` Displays the values of the array **arrayx** for the second thread in process 1.

`for {set i 0} {$i < 100} {incr i} {p argv\[$i\]}`

If **main()** is in the current scope, prints the program's arguments followed by the program's environment strings.

`f {t1.1 t2.1 t3.1} dprint {f()}`

Evaluates a function contained in three threads. Each thread is in a different process:

```
Thread 1.1:
f(): 2
Thread 2.1:
f(): 3
Thread 3.1:
f(): 5
```

```
f {t1.1 t2.1 t3.1} dprint -nowait {f()}
1
```

Evaluates a function without waiting. Later, you can obtain the results using **TV::expr**. The number displayed immediately after the command, which is "1", is the *susp-eval-id*. The following example shows how to get this result:

```
f t1.1 TV::expr get 1 result
2
f t2.1 TV::expr get 1 result
Thread 1.1:
f(): 2
Thread 2.1:
f(): 3
Thread 3.1:
f(): 5
3
f t3.1 TV::expr get 1 result
5
```

dptsets

Shows the status of processes and threads

Format: **dptsets** [*ptset_array*] ...

Arguments: *ptset_array* An optional array that indicates the P/T sets to show. An element of the array can be a number or it can be a more complicated P/T expression. (For more information, see "*Using P/T Set Operators*" in Chapter 12 of the *TotalView Users Guide*.)

Description: The **dptsets** command shows the status of each process and thread in a Tcl array of P/T expressions. These array elements are P/T expressions (see Chapter 12 of the *TotalView Users Guide*), and the elements' array indices are strings that label each element's section in the output.

If you do not use the optional *ptset_array* argument, the CLI supplies a default array that contains all P/T set designators: **error**, **existent**, **held**, **running**, **stopped**, **unheld**, and **watchpoint**.

Examples: The following example displays information about processes and threads in the current focus:

```
d.1<> dptsets
unheld:
1:      808694   Stopped [fork_loopSGI]
  1.1: 808694.1 Stopped PC=0x0d9cae64
  1.2: 808694.2 Stopped PC=0x0d9cae64
  1.3: 808694.3 Stopped PC=0x0d9cae64
  1.4: 808694.4 Stopped PC=0x0d9cae64

existent:
1:      808694   Stopped [fork_loopSGI]
  1.1: 808694.1 Stopped PC=0x0d9cae64
  1.2: 808694.2 Stopped PC=0x0d9cae64
  1.3: 808694.3 Stopped PC=0x0d9cae64
  1.4: 808694.4 Stopped PC=0x0d9cae64

watchpoint:

running:

held:

error:

stopped:
1:      808694   Stopped [fork_loopSGI]
  1.1: 808694.1 Stopped PC=0x0d9cae64
  1.2: 808694.2 Stopped PC=0x0d9cae64
  1.3: 808694.3 Stopped PC=0x0d9cae64
  1.4: 808694.4 Stopped PC=0x0d9cae64
...
```

The following example creates a two-element P/T set array, and then displays the results. Notice the labels in this example.

```
d1.<> set set_info(0) breakpoint(1)
breakpoint(1)
d1.<> set set_info(1) stopped(1)
stopped(1)
d1.<> dptsets set_info
0:
1:      892484   Breakpoint   [arraySGI]
  1.1: 892484.1 Breakpoint   PC=0x10001544, [array.F#81]

1:
1:      892484   Breakpoint   [arraySGI]
  1.1: 892484.1 Breakpoint   PC=0x10001544, [array.F#81]
```

The array index to **set_info** becomes a label identifying the type of information being displayed. In contrast, the information within parentheses in the **breakpoint** and **stopped** functions identifies the arena for which the function returns information.

If you use a number as an array index, you might not remember what is being printed. The following very similar example shows a better way to use these array indices:

```
d1.<> set set_info(my_breakpoints) breakpoint(1)
breakpoint(1)
d1.<> set set_info(my_stopped) stopped(1)
stopped(1)
d1.<> dptsets set_info
my_stopped:
1:      882547   Breakpoint   [arraysSGI]
  1.1: 882547.1 Breakpoint   PC=0x10001544,
[arrays.F#81]

my_breakpoints:
1:      882547   Breakpoint   [arraysSGI]
  1.1: 882547.1 Breakpoint   PC=0x10001544,
[arrays.F#81]
```

The following commands also create a two-element array. This example differs in that the second element is the difference between three P/T sets.

```
d.1<> set mystat(system) a-gW
d.1<> set mystat(reallystopped) \
      stopped(a)-breakpoint(a)-watchpoint(a)
d.1<> dptsets t mystat
system:
```

```
Threads in process 1 [regress/fork_loop]:
1.-1:  21587.[-1] Running PC=0x3ff805c6998
1.-2:  21587.[-2] Running PC=0x3ff805c669c
...
Threads in process 2 [regress/fork_loop.1]:
2.-1:  15224.[-1] Stopped PC=0x3ff805c6998
2.-2:  15224.[-2] Stopped PC=0x3ff805c669c
...

reallystopped:
2.2    224.2 Stopped PC=0x3ff800d5758
2.-1   5224.[-1] Stopped PC=0x3ff805c6998
2.-2:  15224.[-2] Stopped PC=0x3ff805c669c
...
```

drerun

Restarts processes

Format: **drerun** [*cmd_args*] [*in_operation*]
[*out_operations*]
[*error_operations*]

Arguments:

cmd_args		The arguments to be used for restarting a process.
in_operation		Names the file from which the CLI reads input.
	< *infile*	Reads from *infile* instead of **stdin**. *infile* indicates a file from which the launched process reads information.
out_operations		Names the file to which the CLI writes output. In the following, *outfile* indicates the file into which the launched processes writes information.
	> *outfile*	Sends output to *outfile* instead of **stdout**.
	>& *outfile*	Sends output and error messages to *outfile* instead of **stdout** and **stderr**.
	>>& *outfile*	Appends output and error messages to *outfile*.
	>> *outfile*	Appends output to *outfile*.
error_operations		Names the file to which the CLI writes error output. In the following, *errfile* indicates the file into which the launched processes writes error information.
	2> *errfile*	Sends error messages to *errfile* instead of **stderr**.
	2>>*errfile*	Appends error messages to *errfile*.

Description: The **drerun** command restarts the process that is in the current focus set from its beginning. The **drerun** command uses the arguments stored in the **ARGS(dpmid)** and **ARGS_DEFAULT** variables. These are set every time you run the process with different arguments. Consequently, if you do not specify the arguments that the CLI uses when restarting the process, it uses the arguments you used when the CLI previously ran the process. (See **drun** on page 95 for more information.)

The **drerun** command differs from the **drun** command in that:

- If you do not specify an argument, the **drerun** command uses the default values. In contrast, the **drun** command clears the argument list for the program. This means that you cannot use an empty argument list with the **drerun** command to tell the CLI to restart a process and expect that it does not use any arguments.
- If the process already exists, the **drun** command does not restart it. (If you must use the **drun** command, you must first kill the process.) In contrast, the **drerun** command kills and then restarts the process.

The arguments to this command are similar to the arguments used in the Bourne shell.

Issues When Using Starter Programs

Starter programs such as **poe** or **aprun** and the CLI can interfere with one another because each believes that it owns **stdin**. Because the starter program is trying to manage **stdin** on behalf of your processes, it continually reads from **stdin**, acquiring all characters that it sees. This means that the CLI never sees these characters. If your target process does not use **stdin**, you can use the **–stdinmode none** option. Unfortunately, this option is incompatible with **poe –cmdfile** option that is used when specifying **–pgmmodel mpmd**.

If you encounter these problems, try redirecting **stdin** within the CLI; for example:

```
drun < in.txt
```

Command alias:

Alias	Definition	Description
rr	{drerun}	Restarts processes

Examples:

`drerun` Reruns the current process. Because it doesn't use arguments, the process restarts using its previous values.

`rr -firstArg an_argument -aSecondArg a_second_argument`
Reruns the current process. The CLI does not use the process's default arguments because replacement arguments exist.

drestart

Restarts a checkpoint (IBM and SGI only)

Format: Restarts a checkpoint on IBM AIX

drestart [**–halt**] [**–g** *gid*] [**–r** *host*] [**–no_same_hosts**]

Restarts a checkpoint on SGI IRIX

drestart [*process-state*] [**–no_unpark**] [**–g** *gid*] [**–r** *host*]
[**–ask_attach_parallel** | **–no_attach_parallel**]
[**–no_preserve_ids**] *checkpoint-name*

Arguments:

process_state	Defines the state of the process both before and after the checkpoint. If you do not specify a process state, parallel processes are held immediately after the place where the checkpoint occurred. The CLI attaches to these created parallel processes. You can use one of the following options:
–detach	(SGI only) Although TotalView starts checkpointed processes, it does not attach to them.
–go	(SGI only) TotalView starts checkpointed parallel processes and attaches to them.
–halt	TotalView stops checkpointed processes after it restarts them.
–no_unpark	(SGI only) Indicates that the checkpoint was created outside of TotalView or that you used the **dcheckpoint** command's **–no_park** option when you created the checkpoint file.
–g *gid*	Names the control group into which TotalView places all created processes.
–r *host*	Names the remote host upon which the restart occurs.
–ask_attach_parallel	(SGI only) Asks if the CLI automatically attaches to the parallel processes being created. This is most often used in procedures.
–no_attach_parallel	(SGI only) Attach only to the base process. That is, the CLI does not attach to the parallel processes being created.
–no_preserve_ids	(SGI only) Use new IDs after it restarts the processes. If you omit this option, TotalView causes the process to use the same process, group, session, or **ash** IDs after it restarts.
–no_same_hosts	(IBM only) Restart can use any available hosts. If you do not use this option, the restart occurs on the same hosts upon which the program was executing when the checkpoint file was made. If these hosts are not available, the restart operation fails.
checkpoint-name	The file name used when the checkpoint file was saved.

Description: The **drestart** command restores and restarts all of the checkpointed processes. By default, the CLI attaches to the base process. You have the following choices, among others:

- If there are parallel processes related to this base process, TotalView attaches to them.
- If you do not want the CLI to automatically attach to these parallel processes, use the **–no_attach_parallel** option.
- If you do not know if there are parallel processes, if you want the user to decide, or if you are using this command in a Tcl procedure, use the **–ask_parallel_process** option.

Restarting on AIX using LoadLeveler

On the RS/6000, if you want to debug a **LoadLeveler poe** job from the point at which the checkpoint was made, you must resubmit the program as a **LoadLeveler** job to restart the checkpoint. You also need to set the **MP_POE_RESTART_SLEEP** environment variable to an appropriate number of seconds. After you restart **poe**, start TotalView and attach to **poe**.

When attaching to **poe***, parallel tasks are not yet created, so do not try to attach to any of them. Also, use the* **–no_attach_parallel** *option when using the* **dattach** *command.*

You cannot restart the checkpoint using the **drestart** command. **poe** will tell TotalView when it is time to attach to the parallel task so that it can complete the restart operation.

Examples:

```
drestart check1
```

Restarts the processes checkpointed in the **check1** file. The CLI automatically attaches to parallel processes. This is an SGI checkpoint because it specifies a name.

```
drestart -no_unpark check1
```

Restarts the processes checkpointed in the **check1** file. This file was either created outside of TotalView or it was created using the **–no_park** option. This is an SGI checkpoint because it specifies a name.

drun

Starts or restarts processes

Format: **drun** [*cmd_arguments*] [*in_operation infile*]
[*out_operations outfile*]
[*error_operations errfile*]

Arguments:

cmd_arguments	The argument list passed to the process.
in_operation	Names the file from which the CLI reads input.
< *infile*	Reads from *infile* instead of **stdin**. *infile* indicates a file from which the launched process reads information.
out_operations	Names the file to which the CLI writes output. In the following, *outfile* indicates the file into which the launched processes writes information.
> *outfile*	Sends output to *outfile* instead of **stdout**.
>& *outfile*	Sends output and error messages to *outfile* instead of **stdout** and **stderr**.
>>& *outfile*	Appends output and error messages to *outfile*.
>> *outfile*	Appends output to *outfile*.
error_operations	Names the file to which the CLI writes error output. In the following, *errfile* indicates the file into which the launched processes writes error information.
2> *errfile*	Sends error messages to *errfile* instead of **stderr**.
2>>*errfile*	Appends error messages to *errfile*.

Description: The **drun** command launches each process in the current focus and starts it running. The CLI passes the command arguments to the processes. You can also indicate I/O redirection for input and output information. Later in the session, you can use the **drerun** command to restart the program.

The arguments to this command are similar to the arguments used in the Bourne shell.

In addition, the CLI uses the following variables to hold the default argument list for each process:

ARGS_DEFAULT	The CLI sets this variable if you use the **–a** command-line *option* when you started the CLI or TotalView. (This option passes command-line arguments that TotalView uses when it invokes a process.) This variable holds the default arguments that TotalView passes to a process when the process has no default arguments of its own.
ARGS(dpmid)	An array variable that contains the command-line arguments. The index *dpid* is the process ID. This variable holds a process's default arguments. It is always set by the **drun** command, and it also contains any arguments you used when executing a **drerun** command.

If more than one process is launched with a single **drun** command, each receives the same command-line arguments.

In addition to setting these variables by using the –**a** *command-line* option or specifying *cmd_arguments* when you use this or the **drerun** command, you can modify these variables directly with the **dset** and **dunset** commands.

You can only use this command to tell TotalView to execute initial processes, because TotalView cannot directly run processes that your program spawns. When you enter this command, the initial process must have terminated; if it was not terminated, you are told to kill it and retry. (You could, use the **drerun** command instead because the **drerun** commands first kills the process.)

The first time you use the **drun** command, TotalView copies arguments to program variables. It also sets up any requested I/O redirection. If you re-enter this command for processes that TotalView previously started—or use it when you use the **dattach** command to attach to a process—the CLI reinitializes your program.

Issues When Using Starter Programs

Starter programs such as **poe** or **aprun** and the CLI can interfere with one another because each believes that it owns **stdin**. Because the starter program is trying to manage **stdin** on behalf of your processes, it continually reads from **stdin**, acquiring all characters that it sees. This means that the CLI never sees these characters. If your target process does not use **stdin**, you can use the **–stdinmode none** option. Unfortunately, this option is incompatible with **poe –cmdfile** option that is used when specifying **–pgmmodel mpmd**.

If you encounter these problems, try redirecting **stdin** within the CLI; for example:

```
drun < in.txt
```

Command alias:

Alias	Definition	Description
r	drun	Starts or restarts processes

Examples:

`drun` Tells the CLI to begin executing processes represented in the current focus.

`f {p2 p3} drun`
Begins execution of processes 2 and 3.

`f 4.2 r` Begins execution of process 4. This is the same as **f 4 drun**.

`dfocus a drun`
Restarts execution of all processes known to the CLI. If they were not previously killed, you are told to use the **dkill** command and then try again.

`drun < in.txt` Restarts execution of all processes in the current focus, setting them up to get standard input from **in.txt** file.

dset

Changes or views CLI variables

Format: Changes a CLI variable

dset *debugger-var value*

Views current CLI variables

dset [*debugger-var*]

Sets the default for a CLI variable

dset –set_as_default *debugger-var value*

Arguments:

debugger-var	Name of a CLI variable.
value	Value to be assigned to *debugger-var.*
–set_as_default	Sets the value to use as the variable's default. This option is most often used by system administrators to set site-specific defaults in the global **.tvdrc** startup script. Values set using this option replace the CLI built-in default.

Description: The **dset** command sets the value of CLI debugger variables. CLI and TotalView variables are described in Appendix A , "*TotalView Variables*," on page 185.

If you use the **dset** command with no arguments, the CLI displays the names and current values for all CLI variables in the global namespace. If you use only one argument, the CLI returns and displays that variable's value.

The second argument defines the value that replaces a variable's previous value. You must enclose it in quotation marks if it contains more than one word.

If you do not use an argument, the CLI only displays variables in the current namespace. To show all variables in a namespace, enter the namespace name immediately followed by a double colon; for example, **TV::**.

You can use an asterisk (*) as a wildcard character to tell the CLI to match more than one string; for example, **TV::g*** matches all variables in the **TV::** namespace beginning with **g**. For example, to view all variables in the **TV::** namespace, enter the following:

```
dset TV::
```

or:

```
dset TV::GUI::
```

You need to type the double colons at the end of this example when obtaining listings for a namespace. If you omit them, Tcl assumes that you are requesting information on a variable. For example, **dset TV::GUI** looks for a variable named GUI in the **TV** namespace.

Using –set_as_default

When you press a default button within a **File > Preferences** dialog box, TotalView reinitializes some settings to their original values. However, what happens if you set a value in your **tvdrc** file when you press a default button? In this case, setting a variable doesn't change what TotalView thinks the default is, so it still changes the setting back to its defaults.

The next time you invoke TotalView, TotalView will again use the value in your **tvdrc**.

You can tell TotalView that the value set in your **tvdrc** file is the default if you use the **–set_as_default** option. Now when you press a default button, it will use your value instead of its own.

If your TotalView administrator sets up a global **.tvdrc** file, TotalView reads values from that file and merges them with your preferences and other settings. If the value in the **.tvdrc** file changes, TotalView ignores the change because it has already set a value in your local preferences file. If the administrator uses the **–set_as_default** option, you can be told to press the default button to get the changes. If, however, the administrator doesn't use this option, the only way to get changes is by deleting your preferences file.

Examples:

```
dset PROMPT "Fixme% "
```

Sets the prompt to **Fixme%** followed by a space.

```
dset *
```

Displays all CLI variables and their current settings.

```
dset VERBOSE
```

Displays the current setting for output verbosity.

```
dset EXECUTABLE_PATH ../test_dir;$EXECUTABLE_PATH
```

Places **../test_dir** at the beginning of the previous value for the executable path.

```
dset -set_as_default TV::server_launch_string \
   {/use/this/one/tvdsvr}
```

Sets the default value of the **TV::server_launch_string**. If you change this value, you can later select the **Defaults** button within the **File > Preferences Launch String Page** to reset it to this value.

```
dset TV::GUI::fixed_font_size 12
```

Sets the TotalView GUI to display information using a 12-point, fixed-width font. Commands such as this are often found in a startup file.

dstatus

Shows current status of processes and threads

Format: dstatus

Arguments: This command has no arguments

Description: The **dstatus** command prints information about the current state of each process and thread in the current focus.

If you have not changed the focus, the default width is process. In this case, the **dstatus** command shows the status for each thread in process 1. In contrast, if you set the focus to **g1.<**, the CLI displays the status for every thread in the control group that contains process 1.

Command alias:

Alias	Definition	Description
st	dstatus	Shows current status
ST	{dfocus g dstatus}	Shows group status

Examples:

`dstatus` Displays the status of all processes and threads in the current focus; for example:

```
1:      42898    Breakpoint  [arraysAIX]
  1.1: 42898.1   Breakpoint  \
           PC=0x100006a0, [./arrays.F#87]
```

`f a st` Displays the status for all threads in all processes.

`f p1 st` Displays the status of the threads associated with process 1. If the focus is at its default (**d1.<**), this is the same as typing **st**.

`ST` Displays the status of all processes and threads in the control group having the focus process; for example:

```
1:      773686    Stopped  [fork_loop_64]
  1.1: 773686.1  Stopped  PC=0x0d9cae64
  1.2: 773686.2  Stopped  PC=0x0d9cae64
  1.3: 773686.3  Stopped  PC=0x0d9cae64
  1.4: 773686.4  Stopped  PC=0x0d9cae64
2:      779490    Stopped  [fork_loop_64.1]
  2.1: 779490.1  Stopped  PC=0x0d9cae64
  2.2: 779490.2  Stopped  PC=0x0d9cae64
  2.3: 779490.3  Stopped  PC=0x0d9cae64
  2.4: 779490.4  Stopped  PC=0x0d9cae64
```

`f W st` Shows status for all worker threads in the focus set. If the focus is set to **d1.<**, the CLI shows the status of each worker thread in process 1.

`f W ST` Shows status for all worker threads in the control group associated with the current focus.

In this case, TotalView merges the **W** and **g** specifiers in the **ST** alias. The result is the same as if you had entered **f gW st**.

`f L ST` Shows status for every thread in the share group that is at the same PC as the thread of interest.

dstep

Steps lines, stepping into subfunctions

Format: dstep [–back] [*num-steps*]

Arguments:

–back	(ReplayEngine only) Steps to the previous source line, moving into subroutines that called the current function. This option can be abbreviated to **–b.**
num-steps	An integer greater than 0, indicating the number of source lines to execute.

Description: The **dstep** command executes source lines; that is, it advances the program by steps (source lines). If a statement in a source line invokes a subfunction, the **dstep** command steps into the function.

The optional *num-steps* argument tells the CLI how many **dstep** operations to perform. If you do not specify *num-steps*, the default is 1.

The **dstep** command iterates over the arenas in the focus set by doing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is **process** (**p**).

If the width is **process**, the **dstep** command affects the entire process that contains the thread being stepped. Thus, although the CLI is only stepping one thread, all other threads in the same process also resume executing. In contrast, the **dfocus t dstep** command tells the CLI that to only step the *thread of interest*.

On systems having identifiable manager threads, the **dfocus t dstep** *command allows the manager threads as well as the thread of interest to run.*

The action taken on each term in the focus list depends on whether its width is thread, process, or group, and on the group specified in the current focus. (If you do not explicitly specify a group, the default is the control group.)

If some thread hits an action point other than the goal breakpoint during a step operation, that ends the step.

Group Width

The behavior depends on the group specified in the arena:

Process group	TotalView examines that group and identifies each process having a thread stopped at the same location as the thread of interest. TotalView selects one matching thread from each matching process. TotalView then runs all processes in the group and waits until the thread of interest arrives at its goal location; each selected thread also arrives there.
Thread group	The behavior is similar to process width behavior except that all processes in the program control group run, rather than just the process of interest. Regardless of which threads are in the group of interest, TotalView only waits for threads that are in the same share group as the

thread of interest. This is because it is not useful to run threads executing in different images to the same goal.

Process Width (default) The behavior depends on the group specified in the arena. Process width is the default.

Process group	TotalView allows the entire process to run, and execution continues until the thread of interest arrives at its goal location. TotalView plants a temporary breakpoint at the goal location while this command executes. If another thread reaches this goal breakpoint first, your program continues to execute until the thread of interest reaches the goal.
Thread group	TotalView runs all threads in the process that are in that group to the same goal as the thread of interest. If a thread arrives at the goal that is not in the group of interest, this thread also stops there. The group of interest specifies the set of threads for which TotalView waits. This means that the command does not complete until all threads in the group of interest are at the goal.

Thread Width Only the thread of interest is allowed to run. (This is not supported on all systems.)

Command alias:

Alias	Definition	Description
s	dstep	Runs the thread of interest one statement, while allowing other threads in the process to run.
S	{dfocus g dstep}	Searches for threads in the share group that are at the same PC as the thread of interest, and steps one such aligned thread in each member one statement. The rest of the control group runs freely. This is a group stepping command.
sl	{dfocus L dstep}	Steps the process threads in lockstep. This steps the thread of interest one statement, and runs all threads in the process that are at the same PC as the thread of interest to the same (goal) statement. Other threads in the process run freely. The group of threads that is at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
SL	{dfocus gL dstep}	Steps lockstep threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one statement. Other threads in the control group run freely.

Alias	Definition	Description
sw	{dfocus W dstep}	Steps worker threads in the process. This steps the thread of interest one statement, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
SW	{dfocus gW dstep}	Steps worker threads in the group. This steps the thread of interest one statement, and runs all worker threads in the same share group to the same (goal) statement. All other threads in the control group run freely.

Examples:

`dstep`	Executes the next source line, stepping into any procedure call it encounters. Although the CLI only steps the current thread, other threads in the process run.
`s 15`	Executes the next 15 source lines.
`f p1.2 dstep`	Steps thread 2 in process 1 by one source line. This also resumes execution of all threads in process 1; they halt as soon as thread 2 in process 1 executes its statement.
`f t1.2 s`	Steps thread 2 in process 1 by one source line. No other threads in process 1 execute.

dstepi

Steps machine instructions, stepping into subfunctions

Format:	dstepi [–back] [*num-steps*]	
Arguments:	–back	(ReplayEngine only) Steps to the previous instruction, moving into subroutines that called the current function. This option can be abbreviated to **–b.**
	num-steps	An integer greater than 0, indicating the number of instructions to execute.

Description:

The **dstepi** command executes assembler instruction lines; that is, it advances the program by single instructions.

The optional *num-steps* argument tells the CLI how many **dstepi** operations to perform. If you do not specify *num-steps*, the default is 1.

For more information, see **dstep** on page 100.

Command alias:

Alias	Definition	Description
si	dstepi	Runs the thread of interest one instruction while allowing other threads in the process to run.
SI	{dfocus g dstepi}	Searches for threads in the share group that are at the same PC as the thread of interest, and steps one such aligned thread in each member one instruction. The rest of the control group runs freely. This is a group stepping command.
sil	{dfocus L dstepi}	Steps the process threads in lockstep. This steps the thread of interest one instruction, and runs all threads in the process that are at the same PC as the thread of interest to the same instruction. Other threads in the process run freely. The group of threads that is at the same PC is called the *lockstep group*. This alias does not force process width. If the default focus is set to **group**, this steps the group.
SIL	{dfocus gL dstepi}	Steps lockstep threads in the group. This steps all threads in the share group that are at the same PC as the thread of interest one instruction. Other threads in the control group run freely.

Alias	Definition	Description
siw	{dfocus W dstepi}	Steps worker threads in the process. This steps the thread of interest one instruction, and runs all worker threads in the process to the same (goal) statement. The nonworker threads in the process run freely. This alias does not force process width. If the default focus is set to **group**, this steps the group.
SIW	{dfocus gW dstepi}	Steps worker threads in the group. This steps the thread of interest one instruction, and runs all worker threads in the same share group to the same statement. All other threads in the control group run freely.

Examples:

`dstepi`	Executes the next machine instruction, stepping into any procedure call it encounters. Although the CLI only steps the current thread, other threads in the process run.
`si 15`	Executes the next 15 instructions.
`f p1.2 dstepi`	Steps thread 2 in process 1 by one instruction, and resumes execution of all other threads in process 1; they halt as soon as thread 2 in process 1 executes its instruction.
`f t1.2 si`	Steps thread 2 in process 1 by one instruction. No other threads in process 1 execute.

dunhold

Releases a held process or thread

Format: Releases a process

dunhold –process

Releases a thread

dunhold –thread

Arguments:

–process	Tells TotalView to release processes in the current focus. You can abbreviate the **–process** option argument to **–p**.
–thread	Tells TotalView to release threads in the current focus. You can abbreviate the **–thread** option to **–t**.

Description: The **dunhold** command releases the threads or processes in the current focus. You cannot hold or release system manager threads.

Command alias:

Alias	Definition	Description
uhp	{dfocus p dunhold -process}	Releases the process of interest
UHP	{dfocus g dunhold -process}	Releases the processes in the focus group
uht	{dfocus t dunhold –thread}	Releases the thread of interest
UHT	{dfocus g dunhold –thread}	Releases all threads in the focus group
uhtp	{dfocus p dunhold –thread}	Releases the threads in the current process

Examples:

`f w uhtp`	Releases all worker threads in the focus process.
`htp; uht`	Holds all threads in the focus process except the thread of interest.

dunset

Restores default settings for variables

Format:	Restores a CLI variable to its default value dunset *debugger-var* Restores all CLI variables to their default values dunset –all	
Arguments:	*debugger-var*	Name of the CLI variable whose default setting is being restored.
	–all	Restores the default settings of all CLI variables.

Description: The **dunset** command reverses the effects of any previous **dset** commands, restoring CLI variables to their default settings. See Appendix A , "*TotalView Variables*," on page 185 for information on these variables.

Tcl variables (those created with the Tcl **set** command) are not affected by this command.

If you use the –**all** option, the **dunset** command affects all changed CLI variables, restoring them to the settings that existed when the CLI session began. Similarly, specifying *debugger-var* tells the CLI to restore that one variable.

Examples:		
	`dunset PROMPT`	Restores the prompt string to its default setting; that is, {[dfocus]>}.
	`dunset -all`	Restores all CLI variables to their default settings.

duntil

Runs the process until a target place is reached

Format: Runs to a line

duntil [–back] *line-number*

Runs to an address

duntil [–back] –address *addr*

Runs into a function

duntil [–back] *proc-name*

Arguments:

–back	(ReplayEngine only) Steps to the previous instruction, moving into subroutines that called the current function. This option can be abbreviated to **–b.**
line-number	A line number in your program.
–address *addr*	An address in your program.
proc-name	The name of a procedure, function, or subroutine in your program.

Description: The **duntil** command runs the thread of interest until execution reaches a line or absolute address, or until it enters a function.

If you use a process or group width, all threads in the process or group that are not running to the goal are allowed to run. If one of the secondary threads arrives at the goal before the thread of interest, the thread continues running, ignoring this goal. In contrast, if you specify thread width, only the thread of interest runs.

The **duntil** command differs from other step commands when you apply it to a group, as follows:

Process group	TotalView runs the entire group, and the CLI waits until all processes in the group have at least one thread that has arrived at the goal breakpoint. This lets you *sync* up all the processes in a group in preparation for group-stepping them.
Thread group	TotalView runs the process (for **p** width) or the control group (for **g** width) and waits until all the running threads in the group of interest arrive at the goal.

There are some differences between how processes and threads run using the **duntil** command and other stepping commands. Here is how this command determines what it runs:

- **Process Group Operation**: TotalView examines the thread of interest to see if it is already at the goal. If it is, TotalView does not run the process of interest. Similarly, TotalView examines all other processes in the share group, and it only runs processes that do not have a thread at the goal. It also runs members of the control group that are not in the share group.
- **Group-Width Thread Group Operation**: TotalView identifies all threads in the entire control group that are not at the goal. Only those threads run. Although TotalView runs share group members in which all

worker threads are already at the goal, it does not run the workers. TotalView also runs processes in the control group that are outside the share group. The **duntil** command operation ends when all members of the focus thread group are at the goal.

- **Process-Width Thread Group Operation**: TotalView identifies all threads in the entire focus process that are not already at the goal. Only those threads run. The **duntil** command operation ends when all threads in the process that are also members of the focus group arrive at the goal.

Command alias:

Alias	Definition	Description
un	duntil	Runs the thread of interest until it reaches a target, while allowing other threads in the process to run.
UN	{dfocus g duntil}	Runs the entire control group until every process in the share group has at least one thread at the goal. Processes that have a thread at the goal do not run.
unl	{dfocus L duntil}	Runs the thread of interest until it reaches the target, and runs all threads in the process that are at the same PC as the thread of interest to the same target. Other threads in the process run freely. The group of threads that is at the same PC is called the *lockstep group*. This does not force process width. If the default focus is set to **group**, this runs the group.
UNL	{dfocus gL duntil}	Runs lockstep threads in the share group until they reach the target. Other threads in the control group run freely.
unw	{dfocus W duntil}	Runs worker threads in the process to a target. The nonworker threads in the process run freely. This does not force process width. If the default focus is set to **group**, this runs the group.
UNW	{dfocus gW duntil}	Runs worker threads in the same share group to a target. All other threads in the control group run freely.

Examples:

`UNW 580`	Runs all worker threads to line 580.
`un buggy_subr`	Runs to the start of the **buggy_subr** routine.

dup

Moves up the call stack

Format: **dup** [*num-levels*]

Arguments:

num-levels	Number of levels to move up. The default is **1**.

Description: The **dup** command moves the current stack frame up one or more levels. It also prints the new frame number and function.

Call stack movements are all relative, so **dup** effectively "moves up" in the call stack. ("Up" is in the direction of **main()**.)

Frame 0 is the most recent—that is, currently executing—frame in the call stack; frame 1 corresponds to the procedure that invoked the currently executing frame, and so on. The call stack's depth is increased by one each time a your program enters a procedure, and decreases by one when your program exits from it. The effect of the **dup** command is to change the context of commands that follow. For example, moving up one level lets you access variables that are local to the procedure that called the current routine.

Each **dup** command updates the frame location by adding the appropriate number of levels.

The **dup** command also modifies the current list location to be the current execution location for the new frame, so a subsequent the **dlist** command displays the code surrounding this location. Entering the **dup 2** command (while in frame 0) followed by a **dlist** command, for instance, displays source lines centered around the location from which the current routine's parent was invoked. These lines are in frame 2.

Command alias:

Alias	Definition	Description
u	dup	Moves up the call stack

Examples:

`dup`

Moves up one level in the call stack. As a result, subsequent **dlist** commands refer to the procedure that invoked this one. After this command executes, it displays information about the new frame; for example:

```
1 check_fortran_arrays_ PC=0x10001254,
   FP=0x7fff2ed0 [arrays.F#48]
```

`dfocus p1 u 5`

Moves up five levels in the call stack for each thread involved in process 1. If fewer than five levels exist, the CLI moves up as far as it can.

dwait

Blocks command input until the target processes stop

Format: **dwait**

Arguments: This command has no arguments

Description: The **dwait** command tells the CLI to wait for all threads in the current focus to stop or exit. Generally, this command treats the focus the same as other CLI commands.

If you interrupt this command—typically by entering Ctrl+C—the CLI manually stops all processes in the current focus before it returns.

Unlike most other CLI commands, this command blocks additional CLI input until the blocking action is complete.

Examples:

`dwait`	Blocks further command input until all processes in the current focus have stopped (that is, none of their threads are still *running*).
`dfocus {p1 p2} dwait`	Blocks command input until processes 1 and 2 stop.

dwatch

Defines a watchpoint

Format: Defines a watchpoint for a variable

dwatch *variable* [**–length** *byte-count*] [**–g** | **–p** | **–t**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Defines a watchpoint for an address

dwatch –address *addr* **–length** *byte-count* [**–g** | **–p** | **–t**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Arguments:

variable	A symbol name corresponding to a scalar or aggregate identifier, an element of an aggregate, or a dereferenced pointer.
–address *addr*	An absolute address in the file.
–length *byte-count*	The number of bytes to watch. If you enter a variable, the default is the variable's byte length. If you are watching a variable, you only need to specify the amount of storage to watch if you want to override the default value.
–g	Tells TotalView to stop all processes in the process's control group when the watchpoint triggers.
–p	Tells TotalView to stop the process that hit this watchpoint.
–t	Tells TotalView to stop the thread that hit this watchpoint.
–l *lang*	Specifies the language in which you are writing an expression. The values you can use for *lang* are **c**, **c++**, **f7**, **f9**, and **asm**, for C, C++, FORTRAN 77, Fortran-9x, and assembler, respectively. If you do not use a language code, TotalView picks one based on the variable's type. If you only specify an address, TotalView uses the C language. Not all languages are supported on all systems.
–e *expr*	When the watchpoint is triggered, evaluates *expr* in the context of the thread that hit the watchpoint. In most cases, you need to enclose the expression in braces (**{ }**).
–t *type*	The data type of **$oldval**/**$newval** in the expression. If you do not use this option, TotalView uses the variable's datatype. If what you specify is an address and you also use an expression, you must use this option.

Description: The **dwatch** command defines a watchpoint on a memory location where the specified variables are stored. The watchpoint triggers whenever the value of the variable changes. The CLI returns the ID of the newly created watchpoint.

Watchpoints are not available on Macintosh computers running OS X, IBM PowerPC *computers running* Linux Power, IBM Blue Gene/L, *and* Hewlett Packard (HP) *computers running or* HP-UX.

The value set in the **STOP_ALL** variable indicates which processes and threads stop executing.

The watched variable can be a scalar, array, record, or structure object, or a reference to a particular element in an array, record, or structure. It can also be a dereferenced pointer variable.

The CLI lets you obtain a variable's address in the following ways if your application demands that you specify a watchpoint with an address instead of a variable name:

- dprint *&variable*
- dwhat *variable*

The **dprint** command displays an error message if the variable is in a register.

Chapter 15 of the TotalView Users Guide contains additional information on watchpoints.

If you do not use the **–length** option, the CLI uses the length attribute from the program's symbol table. This means that the watchpoint applies to the data object named; that is, specifying the name of an array lets you watch all elements of the array. Alternatively, you can tell TotalView to watch a certain number of bytes, starting at the named location.

In all cases, the CLI watches addresses. If you specify a variable as the target of a watchpoint, the CLI resolves the variable to an absolute address. If you are watching a local stack variable, the position being watched is just where the variable happened to be when space for the variable was allocated.

The focus establishes the processes (not individual threads) for which the watchpoint is in effect.

The CLI prints a message showing the action point identifier, the location being watched, the current execution location of the triggering thread, and the identifier of the triggering threads.

One possibly confusing aspect of using expressions is that their syntax differs from that of Tcl. This is because you need to embed code written in Fortran, C, or assembler within Tcl commands. In addition, your expressions often include TotalView built-in functions.

Command alias:

Alias	Definition	Description
wa	dwatch	Defines a watchpoint

Examples: For these examples, assume that the current process set at the time of the **dwatch** command consists only of process 2, and that **ptr** is a global variable that is a pointer.

`dwatch *ptr`	Watches the address stored in pointer **ptr** at the time the watchpoint is defined, for changes made by process 2.

Only process 2 is stopped. The watchpoint location does not change when the value of **ptr** changes.

`dwatch {*ptr}`
Performs the same action as the previous example. Because the argument to the **dwatch** command contains a space, Tcl requires you to place the argument within braces.

`dfocus {p2 p3} wa *ptr`
Watches the address pointed to by **ptr** in processes 2 and 3. Because this example does not contain either a –**p** or –**g** option, the value of the **STOP_ALL** variable lets the CLI know if it should stop processes or groups.

`dfocus {p2 p3 p4} dwatch -p *ptr`
Watches the address pointed to by **ptr** in processes 2, 3, and 4. The –**p** option indicates that TotalView only stops the process triggering the watchpoint.

`wa * aString -length 30 -e {goto $447}`
Watches 30 bytes of data beginning at the location pointed to by **aString**. If any of these bytes change, execution control transfers to line 447.

```
wa my_vbl -type long
    -e {if ($newval == 0x11ffff38) $stop;}
```
Watches the **my_vbl** variable and triggers when **0x11ffff38** is stored in it.

`wa my_vbl -e {if (my_vbl == 0x11ffff38) $stop;}`
Performs the same function as the previous example. This example tests the variable directly rather than by using the **$newval** variable.

dwhat

Determines what a name refers to

Format: **dwhat** *symbol-name*

Arguments:

symbol-name	Fully or partially qualified name specifying a variable, procedure, or other source code symbol.

Description: The **dwhat** command tells the CLI to display information about a named entity in a program. The displayed information contains the name of the entity and a description of the name.

To view information on CLI *variables or aliases, you need to use the* **dset** *or* **alias** *commands.*

The focus constrains the query to a particular context.

The default width for this command is thread (t).

Command alias:

Alias	Definition	Description
wh	dwhat	Determines what a name refers to

Examples: The following examples show what the CLI displays when you enter one of the indicated commands.

```
dprint timeout
```

```
timeout = {
   tv_sec = 0xc0089540 (-1073179328)
   tv_usec = 0x000003ff (1023)
}
```

```
dwhat timeout
```

In thread 1.1:

```
Name: timeout; Type: struct timeval; Size:
8 bytes; Addr: 0x11fffefc0
   Scope: #fork_loop.cxx#snore \
       (Scope class: Any)
   Address class: auto_var \
      (Local variable)
```

```
wh timeval
```

In process 1:

```
Type name: struct timeval; Size: 8 bytes; \
         Category: Structure
   Fields in type:
   {
   tv_sec   time_t    (32 bits)
   tv_usec  int       (32 bits)
   }
```

```
dlist
```

```
20     float field3_float;
21     double field3_double;
22 en_check en1;
23
24 };
25
26 main ()
27 {
```

```
28    en_check vbl;
29    check_struct s_vbl;
30 >  vbl = big;
31    s_vbl.field2_char = 3;
32    return (vbl + s_vbl.field2_char);
33 }
```

`p vbl`

```
vbl = big (0)
```

`wh vbl`

In thread 2.3:

```
Name: vbl; Type: enum en_check; \
      Size: 4 bytes; Addr: Register 01
   Scope: #check_structs.cxx#main \
      (Scope class: Any)
   Address class: register_var (Register \
      variable)
```

`wh en_check`

In process 2:

```
Type name: enum en_check; Size: 4 bytes; \
      Category: Enumeration
   Enumerated values:
      big    = 0
      little= 1
      fat    = 2
      thin   = 3
```

`p s_vbl`

```
s_vbl = {
   field1_int = 0x800164dc (-2147392292)
   field2_char = '\377' (0xff, or -1)
   field2_chars = "\003"
   <padding> = '\000' (0x00, or 0)
   field3_int = 0xc0006140 (-1073716928)
   field2_uchar = '\377' (0xff, or 255)
   <padding> = '\003' (0x03, or 3)
   <padding> = '\000' (0x00, or 0)
   <padding> = '\000' (0x00, or 0)


   field_sub = {
      field1_int = 0xc0002980 (-1073731200)
      <padding> = '\377' (0xff, or -1)
      <padding> = '\003' (0x03, or 3)
      <padding> = '\000' (0x00, or 0)
      <padding> = '\000' (0x00, or 0)
      field2_long = 0x0000000000000000 (0)
   ...
}
```

`wh s_vbl`

In thread 2.3:

```
Name: s_vbl; Type: struct check_struct; \
      Size: 80 bytes; Addr: 0x11ffff240
   Scope: #check_structs.cxx#main \
      (Scope class: Any)
   Address class: auto_var (Local variable)
```

```
wh check_struct
```

In process 2:

```
Type name: struct check_struct; \
     Size: 80 bytes; Category: Structure
  Fields in type:
  {
  field1_int        int           (32 bits)
  field2_char       char          (8 bits)
  field2_chars      $string[2]    (16 bits)
  <padding>         $char         (8 bits)
  field3_int        int           (32 bits)
  field2_uchar      unsigned char (8 bits)
  <padding>         $char[3]      (24 bits)
  field_sub         struct sub_st (320 bits){
     field1_int     int           (32 bits)
     <padding>      $char[4]      (32 bits)
     field2_long    long          (64 bits)
     field2_ulong   unsigned long (64 bits)
     field3_uint    unsigned int  (32 bits)
     en1            enum en_check  (32 bits)
     field3_double  double        (64 bits)
  }
  ...
}
```

dwhere

Displays the current execution location and call stack

Format: Displays locations in the call stack

```
dwhere [ –level level-num ] [ num-levels ] [ –args ] [ –locals ] [ –registers ]
        [ –noshow_pc ][ –noshow_fp ][ –show_image ]
```

Displays all locations in the call stack

```
dwhere –all [ –args ] [ –locals ] [–registers ]
        [ –noshow_pc ][ –noshow_fp ][ –show_image ]
```

Arguments:

–level *level-num*	Set the level at which **dwhere** starts displaying information.
num-levels	Restricts output to this number of levels of the call stack. If you omit this option, the CLI shows all levels in the call stack.
–args	Displays argument names and values in addition to program location information. If you omit this option, the CLI does not show argument names and values.
–locals	Display each frame's local variables.
–noshow_pc	Do not show the PC.
–noshow_fp	Do not show the FP.
–registers	Displays each frame's registers.
–show_image	Show the executable name as well as the file name.
–all	Shows all levels of the call stack.

Description: The **dwhere** command prints the current execution locations and the call stacks—or sequences of procedure calls—that led to that point. The CLI shows information for threads in the current focus, with the default being to show information at the thread level.

Arguments control the amount of command output in two ways:

- The *num-levels* argument lets you control how many levels of the call stacks are displayed, counting from the uppermost (most recent) level. If you omit this argument, the CLI shows all levels in the call stack, which is the default.
- The **–a** option tells the CLI that it should also display procedure argument names and values for each stack level.

A **dwhere** command with no arguments or options displays the call stacks for all threads in the target set.

The **MAX_LEVELS** variable contains the default maximum number of levels the CLI displays when you do not use the *num-levels* argument.

Output is generated for each thread in the target focus.

Command alias:

Alias	Definition	Description
w	dwhere	Displays the current location in the call stack

Examples:	`dwhere`	Displays the call stacks for all threads in the current focus.
	`dfocus 2.1 dwhere 1`	Displays just the most recent level of the call stack corresponding to thread 1 in process 2. This shows just the immediate execution location of a thread or threads.
	`f p1.< w 5`	Displays the most recent five levels of the call stacks for all threads involved in process 1. If the depth of any call stack is less than five levels, all of its levels are shown. This command is a slightly more complicated way of saying **f p1 w 5** because specifying a process width tells the **dwhere** command to ignore the thread indicator.
	`w 1 -a`	Displays the current execution locations (one level only) of threads in the current focus, together with the names and values of any arguments that were passed into the current process.

dworker

Adds or removes a thread from a workers group

Format: dworker { *number* | *boolean* }

Arguments:

number	If positive, marks the thread of interest as a worker thread by inserting it into the workers group.
boolean	If **true**, marks the thread of interest as a worker thread by inserting it into the workers group. If **false**, marks the thread as being a nonworker thread by removing it from the workers group.

Description: The **dworker** command inserts or removes a thread from the workers group.

If *number* is **0** or **false**, this command marks the thread of interest as a nonworker thread by removing it from the workers group. If *number* is **true** or is a positive value, this command marks the thread of interest as a worker thread by inserting it in the workers group.

Moving a thread into or out of the workers group has no effect on whether the thread is a manager thread. Manager threads are threads that are created by the pthreads package to manage other threads; they never execute user code, and cannot normally be controlled individually. TotalView automatically inserts all threads that are not manager threads into the workers group.

Command alias:

Alias	Definition	Description
wof	{dworker false}	Removes the focus thread from the workers group
wot	{dworker true}	Inserts the focus thread into the workers group

exit

Terminates the debugging session

Format:	exit [–force]	
Arguments:	–force	Tells TotalView to exit without asking permission. This is most often used in scripts.

Description: The **exit** command ends the debugging session.

After you enter this command, the CLI asks to exit. If you answer yes, TotalView exits. If you entered the CLI from the TotalView GUI, this command also closes the GUI window.

If you had invoked the CLI *from within the* TotalView GUI, *pressing* Ctrl+D *closes the* CLI *window without exiting from* TotalView.

TotalView destroys all processes and threads that it makes. Any processes that existed prior to the debugging session (that is, TotalView attached to them because you used the **dattach** command) are detached and left executing.

The **exit** and **quit** commands are interchangeable; they do the same thing.

Examples:	`exit`	Exits TotalView, leaving any attached processes running.

help

Displays help information

Format: help [*topic*]

Arguments:

topic	A CLI topic or command.

Description: The **help** command prints information about the specified topic or command. If you do no use an argument, the CLI displays a list of the topics for which help is available.

If the CLI needs more than one screen to display the help information, it fills the screen with data and then displays a *more* prompt. You can press Enter to see more data or enter **q** to return to the CLI prompt.

When you type a topic name, the CLI attempts to complete what you type. The **help** command also allows you to enter one of the CLI built-in aliases; for example:

```
d1.<> he a
Ambiguous help topic "a". Possible matches:
    alias accessors arguments addressing_expressions
d1.<> he ac
"ac" has been aliased to "dactions":
dactions [ bp-ids ... ] [ -at <source-loc> ] [ -disabled | \
              -enabled ]
   Default alias: ac
...
d1.<> he acc
   The following commands provide access to the properties
   of TotalView objects:
   ...
```

You can use the **capture** command to place help information into a variable.

Command alias:

Alias	Definition	Description
he	help	Displays help information

Examples:

`help help`	Prints information about the **help** command.

quit

Terminates the debugging session

Format:	quit [–force]	
Arguments:	–force	Tells the CLI that it should close all TotalView processes without asking permission.

Description: The **exit** command terminates the TotalView session.

After you enter this command, the CLI asks to exit. If you answer yes, TotalView exits. If you entered the CLI from the TotalView GUI, this command also closes the GUI window.

If you had invoked the CLI *from within the* TotalView GUI, *pressing* Ctrl+D *closes the* CLI *window without exiting from* TotalView*.*

TotalView destroys all processes and threads that it makes. Any processes that existed prior to the debugging session (that is, TotalView attached to them because you used the **dattach** command) are detached and left executing.

The **exit** and **quit** commands are interchangeable; they do the same thing.

Examples:	`quit`	Exits TotalView, leaving any attached processes running.

spurs

Returns information on the spurs library use

Format: Displays help information

spurs [help]

Add directories to the OBJECT_SEARCH_PATH variable

spurs add [directory *directory-list* **...]**

Creates an image-qualified breakpoint

spurs break [*spu-image-name source-location-expression*]

Deletes breakpoints

spurs delete *breakpoint-id-list*

Shows the directories in which TotalView searches for SPURS SPU ELF executables.

spurs info [directory | break]

Prints information about the kernel, the taskset, and tasks

spurs print [kernel [*eaSpurs*] **|**
taskset *eaTaskset* **|**
task *eaTaskset task*ID]

Scans for information—this is a no-op

spurs scan

Arguments:

directories	Names the directories that are to be added to the CLI's OBJECT_SEARCH_PATH variable. Here's an example of how you would use this argument: `spurs add directory directory1 directory2` Notice that the directory names are separated by space characters.
spu-image-name	The name of the image that is or will be loaded by TotalView
spu-source-location-expression	An expression that resolves into a particular line in the image. For more information on location expressions, see **dbreak** on page 31.
eaSpurs	The kernel context at this PPU address.
eaTaskset	

Description: Modeled after the GDB "spurs" command, the **spurs** command was created so that developers who are familiar the GDB command have a similar set of commands in TotalView. However, not all GDB "spurs" commands are implemented within the CLI.

TotalView supports the SPURS library. Here's this library's description in the SPURS documentation:

> libspurs is a user-level thread library for SPUs. In a SPURS environment (SPU Runtime System), SPU threads are managed by SPUs. For this reason, thread switching is more efficient than under PPU management and requires no PPU resources. Using SPURS also makes

it easier to synchronize threads and adjust the load balance on multiple SPUs. SPURS is furthermore highly extensible and allows users to define their own thread models as necessary.

spurs [help]

You can obtain information on how to use the spurs command in two ways:

- Typing **spurs** tells the CLI to write a one line description of its commands.
- Typing **spurs help** displays more information about all **spurs** commands.

spurs add [directory *directory-list*]

Either displays a one-line description of this command or adds directories to be searched when TotalView looks for SPURS SPU executables.

- Typing **spurs add** tells the CLI to write a one line description of this command.
- Typing **spurs add directory** *directory-list* adds directories to the CLI's OBJECT_SEARCH_PATH variable. This variable contains the search path TotalView uses when it searches for SPU ELF executable files. The directories you are adding are placed at the beginning of the list in the order in which you name them. If a directory is already within the list, the previously named directory is removed.

  This command returns the modified OBJECT_SEARCH_PATH variable.

spurs break [*spu-image-name source-location-expression*]

Either displays a one-line description of this command or adds

- Typing **spurs break** tells the CLI to write a one line description of this command.
- Typing **spurs break** *spu-image-name source-loc-expr* creates an image-qualified breakpoint path. This is identical to typing the following CLI command:

  dbreak -pending ##*spu-image-name*#*source-loc-expr*

  This command a pending breakpoint that is only located with the image you name. However, if the image has already been loaded, TotalView will set an ordinary breakpoint rather than a pending breakpoint. When entering this command, the focus must be on a SPU thread.

  This command returns the action point ID of the created breakpoint. You can use this ID with other CLI commands that act upon breakpoints; for example, dactions. ddelete, ddisable, denable, and others.

spurs delete *breakpoint-id-list*

Permanently removes one or more action points. The argument to this command tells the CLI which action points to delete. Unlike **spurs break**, this command does not require that the command focus be set to an SPU thread.

spurs info [directory | break]

Either prints information of the contents of the OBJECT_SEARCH_PATH variable or prints action point information about action points in the thread in the current focus. If you do not use an argument, the CLI prints a line explaining what you can type.

spurs print

The spurs print command can be used in the following ways:

spurs print [kernel [*eaSpurs*] |
taskset *eaTaskset* |
task *eaTaskset taskID*]

spurs print	Displays one line of information on using this command.
spurs print kernel	Displays the kernel context for the SPU threads in the current or named focus. This focus must be one or more SPU threads. **cell_spurs_print_kernel** is an alias for this command.
spurs print [kernel [*eaSpurs*]]	Displays the kernel context at PPU address *eaSpurs*. The command focus must be one or more PPU threads. **cell_spurs_print_kernel** is an alias for this command.
spurs print task *eaTasket*	Prints the task at index *taskId* in the taskset at PPU address *eaTaskset*. The command focus must be one or more PPU threads. **cell_spurs_print_task** is an alias for this command.
spurs print taskset	Prints the taskset for the focus SPU threads. The command focus must be one or more SPU threads. **cell_spurs_print_taskset** is an alias for this command
print taskset *eaTaskset*	Prints the taskset at PPU address *eaTaskset*. The command focus must be one or more PPU threads. **cell_spurs_print_taskset** is an alias for this command.

spurs scan

This command is for compatibility with GDB. Unlike the GDB command, this command is a no-op as TotalView has no need for scanning for SPU executables because searches for SPU executables happen dynamically.

stty

Sets terminal properties

Format: **stty** [*stty-args*]

Arguments:

stty-args	One or more UNIX **stty** command arguments as defined in the **man** page for your operating system.

Description: The CLI **stty** command executes a UNIX **stty** command on the **tty** associated with the CLI window. This lets you set all of your terminal's properties. However, this is most often used to set erase and kill characters.

If you start the CLI from a terminal by using the **totalviewcli** command, the **stty** command alters this terminal's environment. Consequently, the changes you make using this command are retained in the terminal after you exit.

If you omit the *stty-args* argument, the CLI displays information about your current settings.

The output from this command is returned as a string.

Examples:

`stty`	Prints information about your terminal settings. The information it prints is the same as if you had entered **stty** while interacting with a shell.
`stty -a`	Prints information about all of your terminal settings.
`stty erase ^H`	Sets the *erase* key to Backspace.
`stty sane`	Resets the terminal's settings to values that the shell thinks they should be. If you are having problems with command-line editing, use this command. (The **sane** argument is not available in all environments.)

unalias

Removes a previously defined alias

Format:

Removes an alias

unalias *alias-name*

Removes all aliases

unalias –all

Arguments:

alias-name	The name of the alias to delete.
–all	Tells the CLI to remove all aliases.

Description:

The **unalias** command removes a previously defined alias. You can delete all aliases by using the **–all** option. Aliases defined in the **tvdinit.tvd** file are also deleted.

Examples:

`unalias step2`	Removes the **step2** alias; **step2** is undefined and can no longer be used. If **step2** was included as part of the definition of another command, that command no longer works correctly. However, the CLI only displays an error message when you try to execute the alias that contains this removed alias.
`unalias -all`	Removes all aliases.

chapter 3

CLI Namespace Commands

This chapter contains detailed descriptions of the CLI commands that are not in the top-level mainspace.

Command Overview

This section lists all of CLI commands that are not in the top-level mainspace. It also contains a short explanation of what each command does.

Accessor Functions

The following functions, all within the **TV::** namespace, access and set TotalView properties:

- **actionpoint:** Accesses and sets action point properties.
- **expr**: Manipulates values created by the **dprint –nowait** command.
- **focus_groups:** Returns a list containing the groups in the current focus.
- **focus_processes:** Returns a list of processes in the current focus.
- **focus_threads:** Returns a list of threads in the current focus.
- **group**: Accesses and sets group properties.
- **process:** Accesses and sets process properties.
- **scope**: Accesses and sets scope properties.
- **symbol**: Accesses and sets symbol properties.
- **thread:** Accesses and sets thread properties.
- **type:** Accesses and sets data type properties.
- **type_transformation**: Accesses and defines type transformations.

Helper Functions

The following functions, all within the TV:: namespace, are most often used in scripts:

- **dec2hex:** Converts a decimal number into hexadecimal format.
- **dll**: Manages shared libraries.
- **errorCodes:** Returns or raises TotalView error information.
- **hex2dec:** Converts a hexadecimal number into decimal format.
- **read_symbols**: Reads shared library symbols.
- **respond**: Sends a response to a command.
- **source_process_startup**: Reads and executes a .tvd file when TotalView loads a process.

actionpoint

Sets and gets action point properties

Format: TV::actionpoint *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. There are no arguments to this subcommand.
- **get** — Retrieves the values of one or more action point properties. The *other-args* argument can include one or more property names. The CLI returns values for these properties in a list whose order is the same as the names you enter.

  If you use the **–all** option instead of the *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists the action point properties that TotalView can access. There are no arguments to this subcommand.
- **set** — Sets the values of one or more properties. The *other-args* argument contains property name and value pairs.

object-id — An identifier for the action point.

other-args — Arguments that the **get** and **set** actions use.

Description: The **TV::actionpoint** command lets you examine and set the following action point properties and states:

address — The address of the action point.

block_count — The number of addresses associated with an actionpoint.

A single line of code can generate multiple instruction sequences. For example, there may be several entry points to a subroutine, depending on where the caller is. This means that an actionpoint can be set at many addresses even if you are placing it on a single line.

Internally, a block represents one of these addresses.

block_enabled — Each individual actionpoint block is an instruction that TotalView may replace with a trap instruction. (When a trap instruction is encountered, the operating system passes control to the debugger.)

Each block can be enabled or disabled separately. This property type returns a list with in which 1 indicates if the block is enabled and 0 if it is disabled.

This is the only property that can be set from within TotalView. All others are read-only

conflicted — Indicates that another action point shares at least one of the action point blocks. If this condition exists, the block is conflicted. If a block is conflicted, TotalView completely disables the action point.

	The conflicted property is 1 if the actionpoint is conflicted, and 0 if it is not.
context	A string that totally identifies an action point. The location of every action point is represented by a string. Even action points set by clicking on a line number are represented by strings. (In this case, the string is the line number.) Sometimes, this string is all that is needed. Usually, however, more context is needed. For example, a line number needs a file name.
enabled	A value (either 1 or 0) indicating if the action point is enabled. A value of 1 means enabled. (settable)
expression	The expression to execute at an action point. (settable)
id	The ID of the action point.
language	The language in which the action point expression is written.
length	The length in bytes of a watched area. This property is only valid for watchpoints. (settable)
line	The source line at which the action point is set. This property is not valid for watchpoints.
satisfaction_group	The group that must arrive at a barrier for the barrier to be *satisfied*. (settable)
share	A value (either 1 or 0) indicating if the action point is active in the entire share group. A value of 1 means that it is. (settable)
stop_when_done	A value that indicates what is stopped when a barrier is satisfied (in addition to the satisfaction set). Values are **process**, **group**, or **none**. (settable)
stop_when_hit	A value that indicates what is stopped when an action point is hit (in addition to the thread that hit the action point). Values are **process**, **group**, or **none**. (settable)
type	The object's type. (See **type_values** for a list of possible types.)
type_values	Lists values that can TotalView can assign to the **type** property: **break**, **eval**, **process_barrier**, **thread_barrier**, and **watch**.

Examples:

```
TV::actionpoint set 5 share 1 enable 1
```

Shares and enables action point 5.

```
f p3 TV::actionpoint set -all enable 0
```

Disables all the action points in process 3.

```
foreach p [TV::actionpoint properties] {
               puts [format “%20s %s” $p: \
[TV::actionpoint get 1 $p]]
```

Dumps all the properties for action point 1. Here is what your output might look like:

```
             address: 0x1200019a8
             enabled: 0
          expression:
                  id: 1
            language:
              length:
                line: /temp/arrays.F#84
  satisfaction_group:
satisfaction_process:
  satisfaction_width:
               share: 1
      stop_when_done:
       stop_when_hit: group
                type: break
         type_values: break eval
                      process_barrier
                      thread_barrier
                      watch
```

dec2hex

Converts a decimal number into hexadecimal

Format:	TV::dec2hex *number*	
Arguments:	*number*	A decimal number to convert.

Description: The **TV::dec2hex** command converts a decimal number into hexadecimal. This command correctly manipulates 64-bit values, regardless of the size of a **long** value on the host system.

dll

Manages shared libraries

Format: **TV::dll** *action* [*dll-id-list*] [**–all**]

Arguments:

action — The action to perform, as follows:

- **close** — Dynamically unloads the shared object libraries that were dynamically loaded by the **ddlopen** commands corresponding to the list of *dll-id*s.

  If you use the **–all** option, TotalView closes all of the libraries that it opened.
- **commands** — Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. There are no arguments to this subcommand.
- **get** — Retrieves the values of one or more **TV::dll** properties. The *other-args* argument can include one or more property names.

  If you use the **–all** option as the *dll-id-list*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists the **TV::dll** properties that TotalView can access. There are no arguments to this subcommand.
- **resolution_urgency_values** — Returns a list of values that this property can take. This list is operating-system specific, but always includes **{lazy now}**.
- **symbol_availability_values** — Returns a list of values that this property can take. This list is operating system specific, but always includes **{lazy now}**.

dll-id-list — A list of one or more dll-ids. There are the IDs returned by the **ddlopen** command.

–all — Closes all shared libraries that you opened using the **ddlopen** command.

Description: The **TV::dll** command either closes shared libraries that were dynamically loaded with the **ddlopen** command or obtains information about loaded shared libraries.

Examples:

```
TV::dll close 1
```

Closes the first shared library that you opened.

errorCodes

Returns or raises TotalView error information

Format: Returns a list of all error code tags

TV::errorCodes

Returns or raises error information

TV::errorCodes *number_or_tag* [–raise [*message*]]

Arguments:

number_or_tag	An error code mnemonic tag or its numeric value.
–raise	Raises the corresponding error. If you append a *message*, TotalView returns this string. Otherwise, TotalView uses the human-readable string for the error.
message	An optional string used when raising an error.

Description: The **TV::errorCodes** command lets you manipulate the TotalView error code information placed in the Tcl **errorCodes** variable. The CLI sets this variable after every command error. Its value is intended to be easy to parse in a Tcl script.

When the CLI or TotalView returns an error, **errorCodes** is set to a list with the following format:

TOTALVIEW *error-code subcodes... string*

where:

- The first list element is always TOTALVIEW.
- The second list element is always the error code.
- The *subcodes* argument is not used at this time.
- The last list element is a string describing the error.

With a tag or number, this command returns a list containing the mnemonic tag, the numeric value of the tag, and the string associated with the error.

The –**raise** option tells the CLI to raise an error. If you add a message, that message is used as the return value; otherwise, the CLI uses its textual explanation for the error code. This provides an easy way to return errors from a script.

Examples:

```
foreach e [TV::errorCodes] {
            puts [eval format {"%20s %2d %s"} \
            [TV::errorCodes $e]]}
```

Displays a list of all TotalView error codes.

expr

Manipulates values created by the dprint –nowait command

Format: **TV::expr** *action* [*susp-eval-id*] [*other-args*]

Arguments:

action	The action to perform, as follows:
commands	Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.
delete	Deletes all data associated with a suspended ID. If you use this command, you can specify an *other-args* argument. If you use the –**done** option, the CLI deletes the data for all completed expressions; that is, those expressions for which **TV::expr get** *susp-eval-id* **done** returns 1. If you specify –**all**, the CLI deletes all data for all expressions.
get	Gets the values of one or more **expr** properties. The *other-args* argument can include one or more values. The CLI returns these values in a list whose order is the same as the property names. If you use the –**all** option instead of *susp-eval-id*, the CLI returns a list containing one (sublist) element for each object.
properties	Displays the properties that the CLI can access. Do not use additional arguments with this option.
susp-eval-id	The ID returned or thrown by the **dprint** command, or printed by the **dwhere** command.
other-args	Arguments required by the **delete** subcommand.

Description: The **TV::expr** command, in addition to showing you command information, returns and deletes values returned by a **dprint** –**nowait** command. You can use the following properties for this command:

done	**TV::expr** returns 1 if the process associated with *susp-eval-id* has finished in all focus threads. Otherwise, it returns 0.
expression	The expression to execute.
focus_threads	A list of *dpid.dtid* values in which the expression is being executed.
id	The *susp-eval-id* of the object.
initially_suspended_process	A list of dpid IDs for the target processes that received control because they executed the function calls or compiled code. You can wait for processes to complete by entering the following:

```
dfocus p dfocus [TV::expr get \
    susp-eval-id \
    initially_suspended_processes] dwait
```

result	A list of pairs for each thread in the current focus. Each pair contains the thread as the first element and that thread's result string as the second element; for example:

```
d1.<> dfocus {1.1 2.1} TV::expr \
   get susp-eval-id result
{{1.1 2} {2.1 3}} d1.<>
```

The result of expression *susp-eval-id* in thread 1.1 is 2, and in thread 2.1 is 3.

status	A list of pairs for each thread in the current focus. Each pair contains the thread ID as the first element and that thread's status string as the second element. The possible status strings are **done**, **suspended**, and **{error** *diag***}.**

For example, if expression *susp-eval-id* finished in thread 1.1, suspended on a breakpoint in thread 2.1, and received a syntax error in thread 3.1, that expression's status property has the following value when **TV::expr** is focused on threads 1.1, 2.1, and 3.1:

```
d1.<> dfocus {t1.1 t2.1 t3.1} \
   TV::expr get 1 status
{1.1 done} {2.1 suspended} {3.1 {error
{Symbol nothing2 not found}}}
d1.<>
```

focus_groups

Returns a list of groups in the current focus

Format: TV::focus_groups

Arguments: This command has no arguments

Description: The **TV::focus_groups** command returns a list of all groups in the current focus.

Examples:

```
f d1.< TV::focus_groups
```

Returns a list containing one entry, which is the ID of the control group for process 1.

focus_processes

Returns a list of processes in the current focus

Format: TV::focus_processes [–all | –group | –process | –thread]

Arguments:

–all	Changes the default width to **all**.
–group	Changes the default width to **group**.
–process	Changes the default width to **process**.
–thread	Changes the default width to **thread**.

Description: The **TV::focus_processes** command returns a list of all processes in the current focus. If the focus width is something other than **d** (default), the focus width determines the set of processes returned. If the focus width is **d**, the **TV::focus_processes** command returns process width. Using any of the options changes the default width.

Examples:

```
f g1.< TV::focus_processes
```

Returns a list containing all processes in the same control as process 1.

focus_threads

Returns a list of threads in the current focus

Format: TV::focus_threads [–all | –group | –process | –thread]

Arguments:

–all	Changes the default width to **all**.
–group	Changes the default width to **group**.
–process	Changes the default width to **process**.
–thread	Changes the default width to **thread**.

Description: The **TV::focus_threads** command returns a list of all threads in the current focus. If the focus width is something other than **d** (default), the focus width determines the set of threads returned. If the focus width is **d**, the **TV::focus_threads** command returns thread width. Using any of the options changes the default width.

Examples:

```
f p1.< TV::focus_threads
```

Returns a list containing all threads in process 1.

group

Sets and gets group properties

Format: TV::group *action* [*object-id*] [*other-args*]

Arguments:

action	The action to perform, as follows:
commands	Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. Do not use additional arguments with this subcommand.
get	Gets the values of one or more group properties. The *other-args* argument can include one or more property names. The CLI returns the values for these properties in a list in the same order as you entered the property names. If you use the –**all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for each group.
properties	Displays the properties that the CLI can access. Do not use additional arguments with this option.
set	Sets the values of one or more properties. The *other-args* argument is a sequence of property name and value pairs.
object-id	The group ID. If you use the –**all** option, TotalView executes this operation on all groups in the current focus.
other-args	Arguments required by the **get** and **set** subcommands.

Description: The **TV::group** command lets you examine and set the following group properties and states:

actionpoint_count	The number of shared action points planted in the group. This is only valid for share groups and shared action points that are associated with the share group containing the process, rather than with the process itself. When you obtain the results of this read-only value, the number may not look correct as this number also includes “magic breakpoints”. These are breakpoints that TotalView sets behind the scene; they are not usually visible. In addition, these magic breakpoints seldom appear when you use the dactions command.
canonical_execution_name	The absolute file name of the program being debugged. If you had entered a relative name, TotalView finds this absolute name.
count	The number of members in a group.
executable	Like **canonical_execution_name**, this is the absolute file name of the program being debugged. It differs in that it contains symbolic links and the like that exist for the program.
id	The ID of the object.

member_type	The type of the group's members, either **process** or **thread**.
member_type_values	A list of all possible values for the **member_type** property. For all groups, this is a two-item list with the first being the number of proess groups and the second being the number of thread groups. In many ways, this is related to the **type_values** property, which is a list values the **type** property may take.
members	A list of a group's processes or threads.
type	The group's type. Possible values are **control**, **lockstep**, **share**, **user**, and **workers**.
type_values	A list of all possible values for the **type** property.

Examples:

```
TV::group get 1 count
```

Returns the number of objects in group 1.

hex2dec

Converts a hexadecimal number to decimal

Format:	TV::hex2dec *number*	
Arguments:	*number*	A hexadecimal number to convert.

Description: The **TV::hex2dec** command converts a hexadecimal number to decimal. You can type **0x** before this value. The CLI correctly manipulates 64-bit values, regardless of the size of a **long** value.

process

Sets and gets process properties

Format: **TV::process** *action* [*object-id*] [*other-args*]

Arguments:

action	The action to perform, as follows:
commands	Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. Do not use other arguments with this subcommand.
get	Gets the values of one or more process properties. The *other-args* argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the names you enter. If you use the **–all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for each object.
properties	Displays the properties that the CLI can access. Do not use other arguments with this subcommand.
set	Sets the values of one or more properties. The *other-args* arguments contains pairs of property names and values.
object-id	An identifier for a process. For example, **1** represents process 1. If you use the **–all** option, the operation executes upon all objects of this class in the current focus.
other-args	Arguments required by the **get** and **set** subcommands.

Description: The **TV::process** command lets you examine and set process properties and states, as the following list describes:

cannonical_executable_name	The full pathname of the current executable.
clusterid	The ID of the cluster containing the process. This is a number uniquely identifying the TotalView server that owns the process. The ID for the cluster TotalView is running in is always **0** (zero).
data_size	The size of the process's data segment.
duid	The internal unique ID associated with an object.
executable	Like **canonical_execution_name**, this is the absolute file name of the program being debugged. It differs in that it contains an symbolic links and the like that exist for the program.
heap_size	The amount of memory currently being used for data created at runtime. Stated in a different way, the heap is an area of memory that your program uses when it needs to dynamically allocate memory. For example, calls to the **malloc()** function allocate space on the heap while the **free()** function releases the space.
held	A Boolean value (either **1** or **0**) indicating if the process is held. (**1** means that the process is held.)
hia_guard_max_size	The value set for the maximum size for guard blocks that surround a memory allocation. See the "*Debugging*

	Memory Problems Using TotalView Guide" for information on what this size represents.
hia_guard_payload_alignment	The number of bits the guard block is aligned to.
hia_guard_pre_pattern	The numerical value of the bit pattern written into the guard block preceding an allocated memory block.
hia_guard_pre_size	The number of bits into which the guard block preceding an allocated memory block is written.
hia_guard_post_pattern	The numerical value of the bit pattern written into the guard block following an allocated memory block.
hia_guard_post_size	The number of bits into which the guard block following an allocated memory block is written.
hia_paint_pattern_width	Deprecated
hostname	A name of the process's host computer and operating system (if needed); for example, linux-x86 would be returned if the program is running on a 32-bit linux system.
is_parallel	Contains a value indicating if the current process is a parallel process. If it is, its value is 1. Otherwise, its value is 0.
id	The process ID.
image_ids	A list of the IDs of all the images currently loaded into the process both statically and dynamically. The first element of the list is the current executable.
is_parallel	Contains a value indicating if the current process is a parallel process. If it is, its value is 1. Otherwise, its value is 0.
nodeid	The ID of the node upon which the process is running. The ID of each processor node is unique within a cluster.
proc_name	The name of the process currently being executed.
rank	The rank of the currently selected process.
stack_size	The amount of memory used by the currently executing block or routines, and all the routines that have invoked it. For example, if your main routines invokes the **foo()** function, the stack contains two groups of information—these groups are called frames. The first frame contains the information required for the execution of your main routine and the second, which is the current frame, contains the information needed by the **foo()** function. If **foo()** invokes the **bar()** function, the stack contains three frames. When **foo()** finishes executing, the stack only contains one frame.

stack_vm_size	The logical size of the stack is the difference between the current value of the stack pointer and the address from which the stack originally grew. This value can be different from the size of the virtual memory mapping in which the stack resides. For example, the mapping can be larger than the logical size of the stack if the process previously had a deeper nest of procedure calls or made memory allocations on the stack, or it can be smaller if the stack pointer has advanced but the intermediate memory has not been touched. The **stack_vm_size** value is this difference in size.
state	Current state of the process. See **state_values** for a list of states.
state_values	A list of all possible values for the **state** property: **break**, **error**, **exited**, **running**, **stopped**, or **watch**.
syspid	The system process ID.
target_architecture	The machine architecture upon which the current process is executing.
target_byte_ordering	The bit ordering of the current machine. This is either **little_endian** or **big_endian**.
target_processor	The kind of processor upon which the program is executing. For example, this could be **x86** or **x86-64**.
text_size	The amount of memory used to store your program's machine code instructions. The text segment is sometimes called the code segment.
threadcount	The number of threads in the process.
threads	A list of threads in the process.
vm_size	The sum of the sizes of the mappings in the process's address space.

Examples:

```
f g TV::process get -all id threads
```

For each process in the group, creates a list with the process ID followed by the list of threads; for example:

```
{1 {1.1 1.2 1.4}} {2 {2.3 2.5}} {3 {3.1
3.7 3.9}}
```

```
TV::process get 3 threads
```

Gets the list of threads for process 3; for example:

```
1.1 1.2 1.4
```

```
TV::process get 1 image_ids
```

Returns a list of image IDs in process 1; for example:

```
1|1 1|2 1|3 1|4
```

read_symbols

Reads shared library symbols

Format:

Reads symbols from libraries

TV::read_symbols –lib *lib-name-list*

Reads symbols from libraries associated with a stack frame

TV::read_symbols –frame [*number*]

Reads symbols for all stack frames in the backtrace

TV::read_symbols –stack

Arguments:

–lib [*lib-name-list*]	Tells TotalView to read symbols for all libraries whose names are contained within the *lib-name-list* argument. Each name can include the asterisk (*) and question mark (**?**) wildcard characters. This command ignores the current focus; libraries for any process can be affected.
–frame [*number*]	Tells TotalView to read the symbols for the library associated with the current stack frame. If you also enter a frame number, TotalView reads the symbols for the library associated with that frame.
–stack	Reads the symbols for every frame in the backtrace. This is the same as right-clicking in the Stack Trace Pane and selecting the **Load All Symbols in Stack** command. If, while reading in a library, TotalView may also need to read in the symbols from additional libraries.

Description:

The **TV::read_symbols** command reads debugging symbols from one or more libraries that TotalView has already loaded but whose symbols have not yet been read. They are not yet read because the libraries were included within either the **TV::dll_read_loader_symbols_only** or **TV::dll_read_no_symbols** lists.

For more information, see "*Preloading Shared Libraries*" in the "*Debugging Programs*" chapter of the *TotalView Users Guide*.

respond

Provides responses to commands

Format: TV::respond *response command*

Arguments:		
	response	The response to one or more commands. If you include more than one response, separate the responses with newline characters.
	command	One or more commands that the CLI executes.

Description: The **TV::respond** command executes a command. The *command* argument can be a single command or a list of commands. In most cases, you place this information in braces (**{}**). If the CLI asks questions while *command* is executing, you are not asked for the answer. Instead, the CLI uses the characters in the *response* string for the argument. If more than one question is asked and strings within the *response* argument have all been used, The **TV::respond** command starts over at the beginning of the *response* string. If *response* does not end with a newline, the **TV::respond** command appends one.

Do not use this command to suppress the **MORE** prompt in macros. Instead, use the following command:

```
dset LINES_PER_SCREEN 0
```

The most common values for *response* are **y** and **n**.

If you are using the TotalView GUI and the CLI at the same time, your CLI command might cause dialog boxes to appear. You cannot use the **TV::respond** *command to close or interact with these dialog boxes.*

Examples:

```
TV::respond {y} {exit}
```

Exits from TotalView. This command automatically answers the "Do you really wish to exit TotalView" question that the **exit** command asks.

```
set f1 y
set f2 exit
TV::respond $f1 $f2
```

A way to exit from TotalView without seeing the "Do you really wish to exit TotalView" question. This example and the one that preceded are not really what you would do as you would use the **exit –force** command.

scope

Sets and gets internal scope properties

Format: TV::scope *action* [*object-id*] [*other-args*]

Arguments: *action* The action to perform, as follows:

cast Attempts to find or create the type named by the *other-args* argument in the given scope.

code_unit_by_soid Look up loader symbols by using a regular expression to match the base name. For example:

```
TV::scope lookup $scope_id \
     loader_sym_by_regexp \
     "regular expression"
```

commands Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

create Allows you to create blocks, enum_type, named_constant, typedef, upc_shared_type, and variable symbols. The type of symbol determines the properties you meed to specify. In all cases, you must specify the **kind** property. If you are creating a located symbol such as a block, you need to provide a location. If you are creating a upc_shared_type, you need a target_type index.

dump Dump all properties of all symbols in the scope and in the enclosed scope.

get Returns properties of the symbols whose soids are specified. Specify the kinds of properties using the *other-args* argument.

If you use the –**all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for each object.

lookup Look up a symbol by name. Specify the kind of lookup using the *other-args* argument. The values you can enter are:

by_language_rules: Use the language rules of the language of the scope to find a single name.

by_path: Look up a symbol using a pathname.

by_properties [proptery_regexp_pair]: TotalView recurses down the scope tree after it visits a symbol. This means TotalView will search for matching symbols in the specified scope and any nested scope. The **walk** property shows an example.

by_type_index: Look up a symbol using a type index.

in_scope: Look up a name in the given scope and in all enclosing scopes, and in the global scope.

lookup_keys
Displays the kinds of lookup operations that you can perform.

properties Displays the properties that the CLI can access. Do not use additional arguments with this option. The arguments displayed are those that are displayed for the scope of all types. Additional properties also exist but are not shown.(Only the ones used by all are visible.) For more information, see **TV::symbol**.

walk Walk the scope, calling Tcl commands at particular points in the walk. The commands are named using the following options:

by_properties [proptery_regexp_pair]: TotalView recurses down the scope tree after it visits a symbol. This means TotalView will search for matching symbols in the specified scope and any nested scope. For example:

```
TV::scope walk $scope_id by_properties \
      kind typedef base_name "^__BMN_.*$"
```

–**pre_scope** *tcl_cmd*: Names the commands called before walking a scope.

–**pre_sym** *tcl_cmd*: Names the commands called before walking a symbol.

–**post_scope** *tcl_cmd*: Names the commands called after walking a scope.

–**post_symbol** *tcl_cmd*: Names the commands called after walking a symbol.

tcl_cmd: Names the commands called for each symbol.

object-id The ID of a scope.

other-args Arguments required by the **get** subcommand.

Description: The **TV::scope** command lets you examine and set a scope's properties and states.

Examples:

```
TV::scope create $scope kind [kind] \
    [required_property_regexp_pair]...
    [non-required_property_regexp_pair]...
```

This is the general specification for creating a symbol

```
TV::scope create 1|31 kind block location {ldam 0x12}
```

Create a block. A block should have a length. However, you can set the length later using the **set** property.

source_process_startup

Reads, then executes a .tvd file when a process is loaded

Format:	**TV::source_proccess_startup** *process_id*	
Arguments:	*process_id*	The PID of the current process.
Description:	The **TV::source_process_startup** command loads and interprets the **.tvd** file associated with the current process. That is, if a file named *executable*.**tvd** exists, the CLI reads and then executes the commands in it.	

symbol

Gets and sets symbol properties

Format: **TV::symbol** *action* [*object-id*] [*other-args*]

Arguments: *action* The action to perform, as follows:

code_unit_by_soid
Returns the containing scope of a line number. For example:

```
TV::symbol code_unit_by_soid $start_line
```

commands Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

dump Dumps all properties of the symbol whose soid (symbol object ID) is named. Do not use additional arguments with this command.

get Returns properties of the symbols whose soids are specified here. The *other-args* argument names the properties to be returned.

properties Displays the properties that the CLI can access. Do not use additional arguments with this option. These properties are discussed later in this section.

read_delayed
Only global symbols are initially read; other symbols are only partially read. This command forces complete symbol processing for the compilation units that contain the named symbols.

resolve_final
Performs a sequence of **resolve_next** operations until the symbol is no longer undiscovered. If you apply this operation to a symbol that is not undiscovered, it returns the symbol itself.

resolve_next
Some symbols only serve to hold a reference to another symbol. For example, a **typedef** is a reference to the aliased type, or a **const**-qualified type is a reference to the non-**const**s qualified type. These reference types are called *undiscovered symbols*. This operation, when performed on an undiscovered symbol, returns the symbol the type refers to. When this is performed on a symbol, it returns the symbol itself.

rebind Changes one or more structural properties of a symbol. These operations can crash TotalView or cause it to produce inconsistent results. The properties that you can change are:

address: the new address:

base_name: the new base name. The symbol must be a base name.

line_number: the new line number. The symbol must be a line number symbol.

loader_name: the new loader name and a file name.

scope: the soid of a new scope owner.

type_index: the new type index, in the form **<n, m, p>**. The symbol must be a type.

set Sets a symbol's property. Not all properties can be set. Determine which properties can be set using the **writable_properties** property. For example,

```
TV::symbol set $new_upc_type \
     type_index $old_idx
```

writable_properties
Returns a list of writable properties. For example:

```
TV::symbol writable_properties $symbol_id
```

object-id The ID of a symbol.

other-args Arguments required by the **get** subcommand.

Description: The **TV::symbol** command lets you examine and set the symbol properties and states.

Symbol Properties

The following table lists the properties associated with the symbols information that TotalView stores. Not all of this information will be useful when creating transformations. However, it is possible to come across some of these properties and this information will help you decide if you need to use it in your transformation. In general, the properties used in the transformation files that TotalView Technologies provided will be the ones that you will use.

Symbol Kind	Has base_name	Has type_index	Property		
aggregate_type	4	4	aggregate_kind artificial external_name	full_pathname id kind	length logical_scope_owner scope_owner
array_type	4	4	artificial data_addressing element_addressing external_name full_pathname id	index_type_index kind logical_scope_owner lower_bound scope_owner stride_bound	submembers target_type_index upper_bound validator
block	4		address_class artificial full_pathname	id kind length	location logical_scope_owner scope_owner
char_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner target_type_index
code_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner

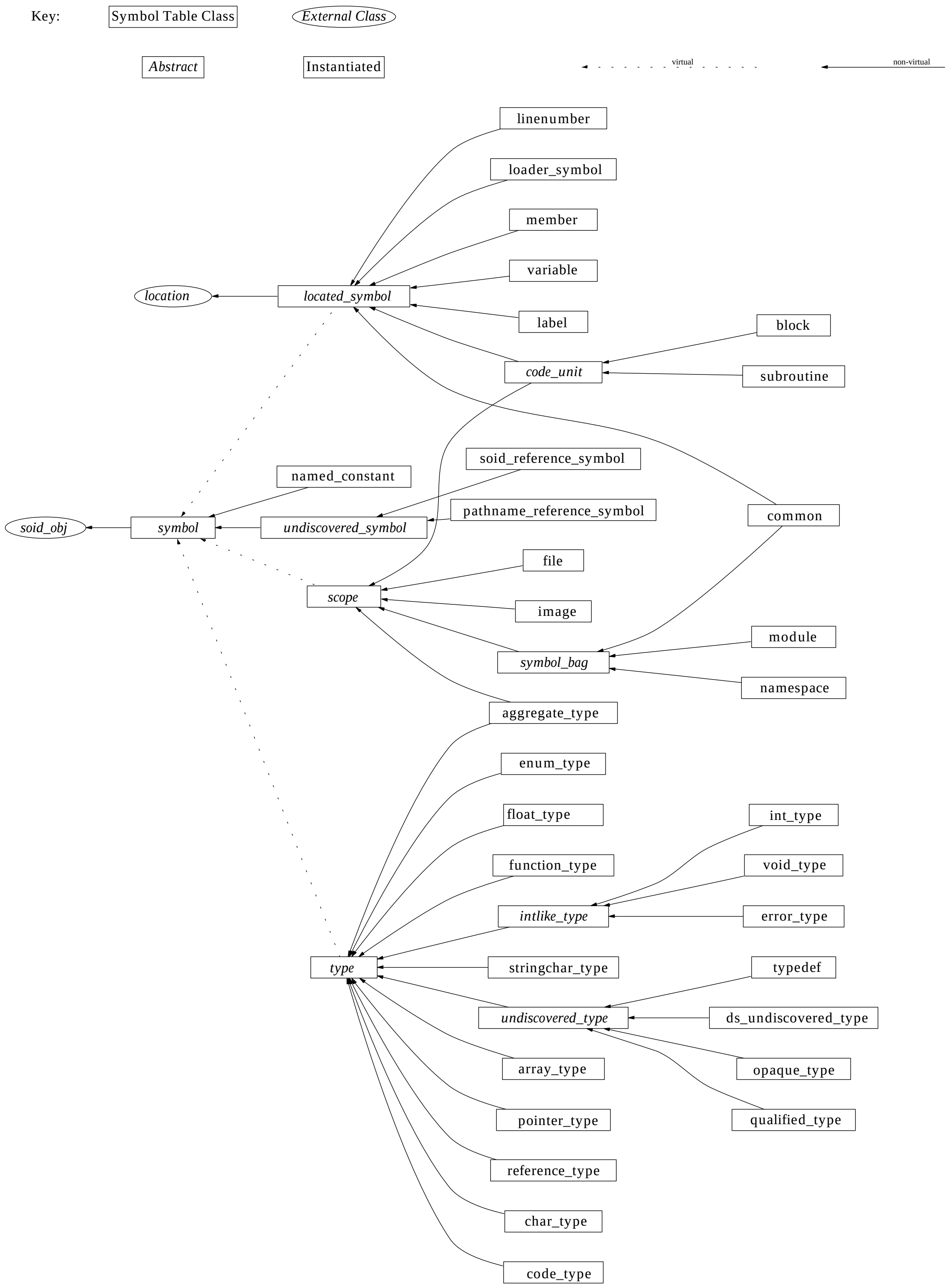
Symbol Kind	Has base_name	Has type_index	Property		
common	4		address_class artificial full_pathname	id kind location	logical_scope_owner scope_owner
ds_ undiscovered_ type	4	4	artificial full_pathname id	kind logical_scope_owner scope_owner	target_type_index
enum_type	4	4	artificial enumerators external_name	full_pathname id kind	logical_scope_owner scope_owner value_size
error_type	4	4	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
file	4		artificial compiler_kind delayed_symbol demangler	full_pathname id kind language	logical_scope_owner scope_owner
float_type	4	4	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
function_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
image	4		artificial full_pathname	id	kind
int_type	4	4	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
label	4		address_class artificial full_pathname	id kind location	logical_scope_owner scope_owner
linenumber			address_class artificial full_pathname	id kind location	logical_scope_owner scope_owner
loader_symbol			address_class artificial full_pathname	id kind length	location logical_scope_owner scope_owner
member	4		address_class artificial full_pathname id	inheritance kind location logical_scope_owner	ordinal scope_owner type_index
module	4		artificial full_pathname	id kind	logical_scope_owner scope_owner
named_constant	4		artificial full_pathname id	kind length logical_scope_owner	scope_owner type_index value
namespace	4		artificial full_pathname	id kind	logical_scope_owner scope_owner

Symbol Kind	Has base_name	Has type_index	Property		
common	4		address_class	id	logical_scope_owner
			artificial	kind	scope_owner
			full_pathname	location	
ds_ undiscovered_ type	4	4	artificial	kind	target_type_index
			full_pathname	logical_scope_owner	
			id	scope_owner	
enum_type	4	4	artificial	full_pathname	logical_scope_owner
			enumerators	id	scope_owner
			external_name	kind	value_size
error_type	4	4	artificial	id	logical_scope_owner
			external_name	kind	scope_owner
			full_pathname	length	
file	4		artificial	full_pathname	logical_scope_owner
			compiler_kind	id	scope_owner
			delayed_symbol	kind	
			demangler	language	
float_type	4	4	artificial	id	logical_scope_owner
			external_name	kind	scope_owner
			full_pathname	length	
function_type	4	4	artificial	id	scope_owner
			external_name	kind	
			full_pathname	logical_scope_owner	
image	4		artificial	id	kind
			full_pathname		
int_type	4	4	artificial	id	logical_scope_owner
			external_name	kind	scope_owner
			full_pathname	length	
label	4		address_class	id	logical_scope_owner
			artificial	kind	scope_owner
			full_pathname	location	
linenumber			address_class	id	logical_scope_owner
			artificial	kind	scope_owner
			full_pathname	location	
loader_symbol			address_class	id	location
			artificial	kind	logical_scope_owner
			full_pathname	length	scope_owner
member	4		address_class	inheritance	ordinal
			artificial	kind	scope_owner
			full_pathname	location	type_index
			id	logical_scope_owner	
module	4		artificial	id	logical_scope_owner
			full_pathname	kind	scope_owner
named_constant	4		artificial	kind	scope_owner
			full_pathname	length	type_index
			id	logical_scope_owner	value
namespace	4		artificial	id	logical_scope_owner
			full_pathname	kind	scope_owner

Symbol Kind	Has base_name	Has type_index	Property		
opaque_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
pathname_-reference_symbol	4		artificial id full_pathname	kind lookup_scope logical_scope_owner	resolved_symbol_-pathname scope_owner
pointer_type		4	artificial external_name full_pathname id	kind length logical_scope_owner scope_owner	target_type_index validator
qualified_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	qualification scope_owner target_type_index
soid_reference_-symbol	4		artificial full_pathname id	kind logical_scope_owner resolved_symbol_id	scope_owner
stringchar_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner target_type_index
subroutine	4		address_class artificial full_pathname id	kind length location logical_scope_owner	return_type_index scope_owner static_chain static_chain_height
typedef	4	4	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner target_type_index
variable	4		address_class artificial full_pathname id	is_argument kind location logical_scope_owner	ordinal scope_owner type_index
void_type	4	4	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
wchar_type	4	4	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner target_type_index

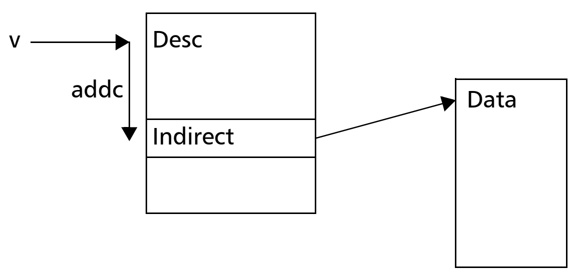
The figure on the following page shows how these symbols are related. Here are definitions of the properties associated with these symbols.

address_class	contains the location for a variety of objects such as a **func**, **global_var**, and a **tls_global**.
aggregate_kind	One of the following: **struct**, **class**, or **union**.
artificial	A Boolean (0 or 1) value where true indicates that the compiler generated the symbol.
compiler_kind	The compiler or family of compiler used to create the file; for example, **gnu**, **xlc**, **intel**, and so on.


Key:
Symbol Table Class
External Class
Abstract
Instantiated
virtual
non-virtual
linenumber
loader_symbol
member
variable
location
located_symbol
label
block
code_unit
subroutine
soid_reference_symbol
named_constant
pathname_reference_symbol
common
soid_obj
symbol
undiscovered_symbol
file
scope
image
module
symbol_bag
namespace
aggregate_type
enum_type
float_type
int_type
function_type
void_type
intlike_type
error_type
type
stringchar_type
typedef
undiscovered_type
ds_undiscovered_type
array_type
opaque_type
pointer_type
qualified_type
reference_type
char_type
code_type

data_addressing	Contains additional operands to get from the base of an object to its data; for example, a Fortran by-desc array contains a descriptor data structure. The variable points to the descriptor. If you do an **addc** operation on the descriptor, you can then do an **indirect** operation to locate the data.

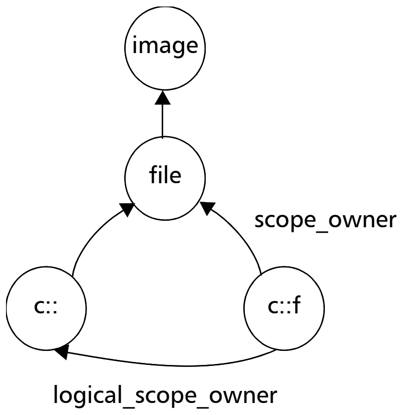
Figure 1: Data Addressing



delayed_symbol	Indicates if a symbol has been full or partially read-in. The following constants are or'd and returned: **skim**, **index**, **line**, and **full**.
demangler	The name of demangler used by your compiler.
element_addressing	The location containing additional operands that let you go from the data's base location to an element.
enumerators	Name of the enumerator tags. For example, if you have something like **enum[R,G,B]**, the tags would be **R**, **G**, and **B**.
external_name	When used in data types, it translates the object structure to the type name for the language. For example, if you have a pointer that points to an **int**, the external name is **int ***.
full_pathname	This is the # separated static path to the variable; for example, **##image#file#externalname**....
id	The internal object handle for the symbol. These symbols always take the form *number*|*number*.
index_type_index	The array type's index **type_index**; for example, this indicates if the index is a 16-, 32-, 64-bit, and so on.
inheritance	For C++ variables, this string is as follows: **[virtual] [{ private | protected | public }] [base class]**
is_argument	A true/false value indicating if a variable was a parameter (dummy variable) passed into the function.
kind	One of the symbol types listed in the first column of the previous table.
language	A string containing a value such as C, C++, or Fortran.

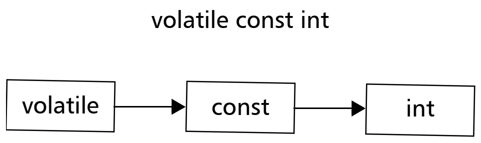
length	The byte size of the object. For example, this might represent the size of an array or a subroutine.
location	The location in memory where an object's storage begins.
logical_scope_owner	The current scope's owner as defined by the language's rules.

Figure 2: Logical Scope Owner



lookup_scope	This is a pathname reference symbol that refers to the scope in which to look up a pathname.
lower_bound	The location containing the array's lower bound. This is a numeric value, not the location of the first array item.
ordinal	The order in which a member or variable occurred within a scope.
qualification	A qualifier to a data type such as **const** or **volatile**. These can be chained together if there is more than one qualifier.

Figure 3: Qualification



resolved_symbol_id	The soid to lookup in a soid reference symbol.
resolved_symbol_pathname	The pathname to lookup in a Fortran reference symbol.
return_type_index	The data type of the value returned by a function.
scope_owner	The ID of the symbol's scope owner. (This is illustrated by the figure within the **logical_scope_owner** definition.)
static_chain	The location of a static link for nested subroutines.

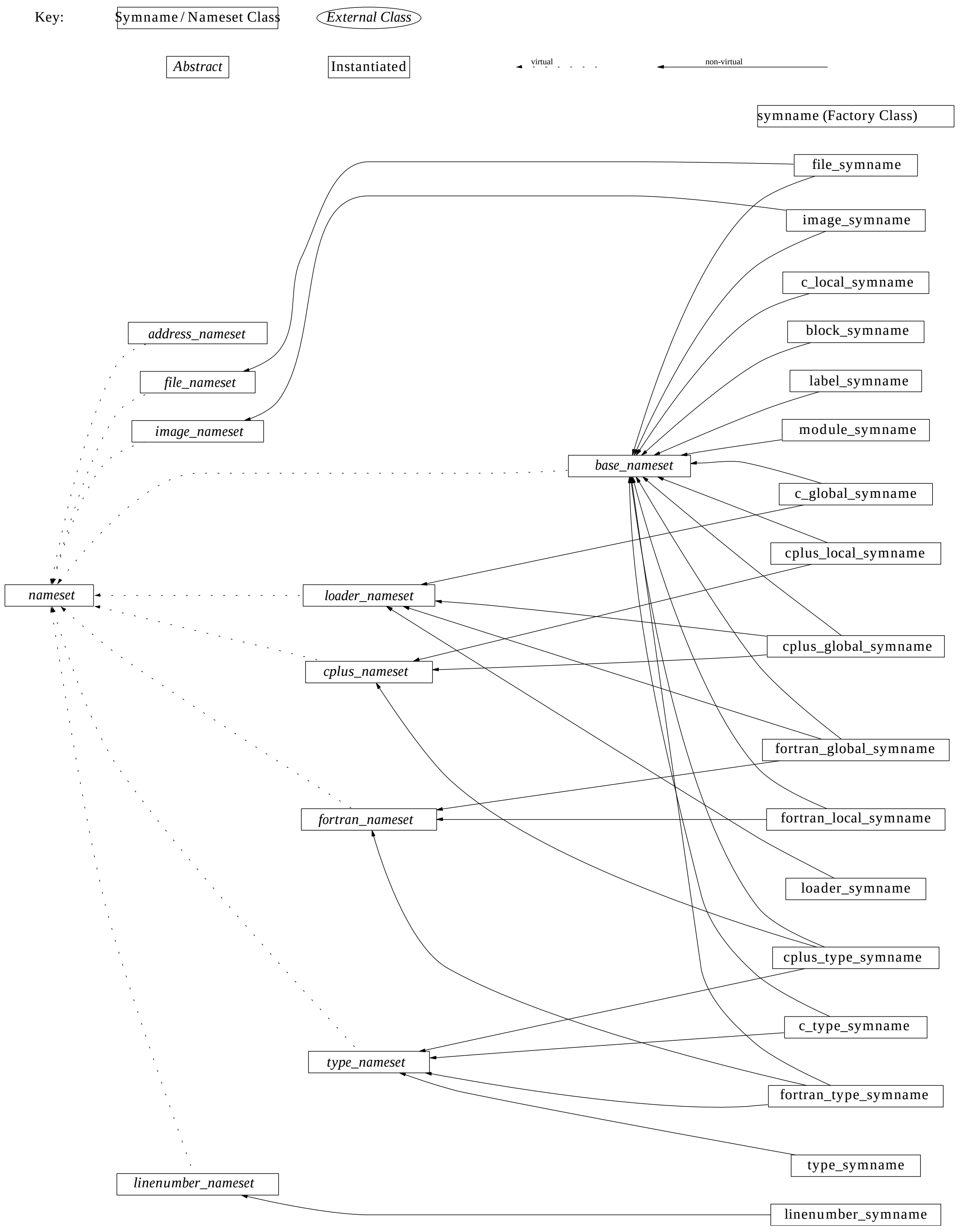
static_chain_height	For nested subroutines, this indicates the nesting level.
stride_bound	Location of the value indicating an array's stride.
submembers	If you have an array of aggregates or pointers and you have already dived on it, this property gives you a list of {*name type*} tuples where **name** is the name of the member of the array (or * if it's an array of pointers), and **type** is the soid of the type that should be used to dive in all into that field.
target_type_index	The type of the following entities: **array**, **ds_undiscovered_type**, **pointer**, and **typedef**.
type_index	One of the following: **member**, **variable**, or **named_constant**.
upper_bound	The location of the value indicating an array's upper bound or extent.
validator	The name of an array or pointer validator. This looks at an array descriptor or pointer to determine if it is allocated and associated.
value	For enumerators, this indicates the item's value in hexadecimal bytes.
value_size	For enumerators, this indicates the length in bytes

Symbol Namespaces

The symbols described in the previous section all reside within namespaces. Like symbols, namespaces also have properties. The figure on the next page illustrates how these namespaces are related.

The following table lists the properties associated with a namespace.

Symbol Namespaces	Properties	
block_symname	base_name	
c_global_symname	base_name	loader_name
	loader_file_path	
c_local_symname	base_name	
c_type_symname	base_name	type_index
cplus_global_symname	base_name	cplus_template_types
	cplus_class_name	cplus_type_name
	cplus_local_name	loader_file_path
	cplus_overload_list	loader_name
cplus_local_symname	base_name	cplus_overload_list
	cplus_class_name	cplus_template_types
	cplus_local_name	cplus_type_name
cplus_type_symname	base_name	cplus_template_types
	cplus_class_name	cplus_type_name
	cplus_local_name	type_index
	cplus_overload_list	
file_symname	base_name	directory_path
	directory_hint	


Key:
Symname / Nameset Class
External Class
Abstract
Instantiated
virtual
non-virtual
symname (Factory Class)
file_symname
image_symname
c_local_symname
block_symname
label_symname
module_symname
c_global_symname
cplus_local_symname
cplus_global_symname
fortran_global_symname
fortran_local_symname
loader_symname
cplus_type_symname
c_type_symname
fortran_type_symname
type_symname
linenumber_symname
address_nameset
file_nameset
image_nameset
base_nameset
nameset
loader_nameset
cplus_nameset
fortran_nameset
type_nameset
linenumber_nameset

Symbol Namespaces	Properties	
fortran_global_symname	base_name	loader_file_path
	fortran_module_name	loader_name
	fortran_parent_function_name	
fortran_local_symname	base_name fortran_parent_function_name	
	fortran_module_name	
fortran_type_symname	base_name	fortran_parent_function_name
	fortran_module_name	type_index
image_symname	base_name	member_name
	directory_path	node_name
label_symname	base_name	
linenumber_symname	linenumber	
loader_symname	loader_file_path	loader_name
module_symname	base_name	
type_symname	type_index	

Many of the following properties are used in more than one namespace. The explanations for these properties will assume a limited context as their use is similar. Some of these definitions assume that you're are looking at the following function prototype:

```
void c::foo<int>(int &)
```

base_name The name of the function; for example, **foo**.

cplus_class_name
The C++ class name; for example, c.

cplus_local_name
Not used.

cplus_overload_list
The function's signature; for example, **int &**.

cplus_template_types
The template used to instantiate the function; for example: **<int>**.

cplus_type_name
The data type of the returned value; for example, *void*.

directory_hint The directory to which you were attached when you started TotalView.

directory_path Your file's pathname as it is named within your program.

fortran_module_name
The name of your module. Typically, this looks like **module'var** or **module'subr'var**.

fortran_parent_function_name
The parent of the subroutine. For example, the parent is **module** in a reference such as **module'subr**. If you have an inner subroutine, the parent is the outer subroutine.

linenumber The line number at which something occurred.

loader_file_path	The file's pathname.
loader_name	The mangled name.
member_name	In a library, you might have an object reference; for example, libC.a(foo.so). foo.so is the member name.
node_name	Not used.
type_index	A handle that points to the type definition. It's format is <number,number,number>.

thread

Gets and sets thread properties

Format: **TV::thread** *action* [*object-id*] [*other-args*]

Arguments:

action	The action to perform, as follows:
commands	Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. Do not use other arguments with this option.
get	Gets the values of one or more thread properties. The *other-args* argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the names you enter. If you use the **–all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for each object.
properties	Lists an object's properties. Do not use other arguments with this option.
set	Sets the values of one or more properties. The *other-args* argument contains paired property names and values.
object-id	A thread ID. If you use the **–all** option, the operation is carried out on all threads in the current focus.
other-args	Arguments required by the **get** and **set** subcommands.

Description: The **TV::thread** command lets you examine and set the following thread properties and states:

canonical_executable_name	The absolute file name of the program being debugged. If you had entered a relative name, TotalView find this absolute name.
continue_sig	The signal to pass to a thread the next time it runs. On some systems, the thread receiving the signal might not always be the one for which this property was set.
current_ap_id	The ID of the action point at which the current thread is stopped.
dpid	The ID of the process associated with a thread.
duid	The internal unique ID associated with the thread.
held	A Boolean value (either **1** or **0**) indicating if the thread is held. (**1** means that the thread is held.) (settable)
id	The ID of the thread.
manager	A Boolean value (either **1** or **0**) indicating if this is a system manger thread. (**1** means that it is a system manager thread.)
pc	The current PC at which the target is executing. (settable)
sp	The value of the stack pointer.
state	The current state of the target. See **state_values** for a list of states.

state_values	A list of values for the **state** property: **break**, **error**, **exited**, **running**, **stopped**, and **watch**.
stop_reason_message	The reason why the current thread is stopped; for example, **Stop Signal**.
systid	The system thread ID.
target_architecture	The machine architecture upon which the current thread is executing.
target_byte_ordering	The bit ordering of the current machine. This is either **little_endian** or **big_endian**.
target_processor	The kind of processor upon which the current thread is executing. For example, this could be **x86** or **x86-64**.

Examples:

```
f p3 TV::thread get -all id
```

Returns a list of thread IDs for process 3; for example:

```
1.1 1.2 1.4
```

```
proc set_signal {val} {
    TV::thread set \
        [f t TV::focus_threads] continue_sig $val
}
```

Set the starting signal for the focus thread.

```
proc show_signal {} {
    foreach th [TV::focus_threads] {
        puts "Continue_sig ($th): \
            [TV::thread get $th continue_sig]";
    }
}
```

Show all starting signals

type

Gets and sets type properties

Format: **TV::type** *action* [*object-id*] [*other-args*]

Arguments:

action	The action to perform, as follows:
commands	Displays the subcommands that you can use. The CLI responds by displaying these four *action* subcommands. Do not use other arguments with this option.
get	Gets the values of one or more type properties. The *other-args* argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the names you enter. If you use the –**all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for each object.
properties	Lists a type's properties. Do not use other arguments with this option.
set	Sets the values of one or more type properties. The *other-args* argument contains paired property names and values.
object-id	An identifier for an object; for example, **1** represents process 1, and **1.1** represents thread 1 in process 1. If you use the –**all** option, the operation is carried out on all objects of this class in the current focus.
other-args	Arguments required by the **get** and **set** subcommands.

Description: The **TV::type** command lets you examine and set the following type properties and states:

enum_values	For an enumerated type, a list of **{name value}** pairs giving the definition of the enumeration. If you apply this to a non-enumerated type, the CLI returns an empty list.
id	The ID of the object.
image_id	The ID of the image in which this type is defined.
language	The language of the type.
length	The length of the type.
name	The name of the type; for example, **class foo**.
prototype	The ID for the prototype. If the object is not prototyped, the returned value is **{}**.
rank	(array types only) The rank of the array.
struct_fields	(**class**/**struct**/**union** types only). A list of lists that contains descriptions of all the type's fields. Each sublist contains the following fields: { *name type_id addressing properties* }

	where: *name* is the name of the field. *type_id* is simply the *type_id* of the field. *addressing* contains additional addressing information that points to the base of the field. *properties* contains an additional list of properties in the following format: "[virtual] [public|private|protected] base class" If no properties apply, this string is null. If you use **get struct_fields** for a type that is not a **class**, **struct**, or **union**, the CLI returns an empty list.
target	For an array or pointer type, returns the ID of the array member or target of the pointer. If you do not apply a command that uses this argument to one of these types, the CLI returns an empty list.
type	Returns a string describing this type; for example, **signed integer**.
type_values	Returns all possible values for the **type** property.

Examples:

```
TV::type get 1|25 length target
```

Finds the length of a type and, assuming it is a pointer or an array type, the target type. The result might look something like:

4 1|12

The following example uses the **TV::type properties** command to obtain the list of properties. It begins by defining a procedure:

```
proc print_type {id} {
   foreach p [TV::type   properties] {
      puts [format "%13s  %s" $p [TV::type get $id $p]]
   }
}
```

You then display information with the following command:

```
print_type 1|6

enum_values
id                    1|6
image_id              1|1
language              f77
length                4
name                  <integer>
prototype
rank                  0
struct_fields
target
type                  Signed Integer
type_values           {Array} {Array of characters}
                      {Enumeration}...
```

type_transformation

Creates type transformations and examines properties

Format: TV::type_transformation *action* [*object-id*] [*other-args*]

Arguments:

action The action to perform, as follows:

commands Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

create Creates a new transformation object. The *object-id* argument is not used; *other-args* is **Array**, **List**, **Map**, or **Struct**, indicating the type of transformation being created. You can change a transformation's properties up to the time you install it. After being installed, you can longer change them.

get Gets the values of one or more transformation properties. The *other-args* argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the property names you entered.

If you use the **-all** option instead of *object-id*, the CLI returns a list containing one (sublist) element for the object.

properties Displays the properties that the CLI can access. Do not use additional arguments with this option. These properties are discussed later in this section.

set Sets the values of one or more properties. The *other-args* argument consists of pairs of property names and values. The argument pairs that you can set are listed later in this section.

object-id The type transformation ID. This value is returned when you crate a new transformation; for example, **1** represents process 1. If you use the **-all** option, the operation executes upon all objects of this class in the current focus.

other-args Arguments required by **get** and **set** subcommands.

Description: The **TV::type_transformation** command lets you define and examine properties of a type transformation. The states and properties you can set are:

addressing_callback

Names the procedure that locates the address of the start of an array. The call structure for this callback is:

addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

This callback defines a TotalView addressing expression that computes the starting address of an array's first element.

compiler	Reserved for future use.
id	The type transformation ID returned from a **create** operation.
language	The language property specifies source language for the code of the aggregate type (class) to transform. This is always C++.

list_element_count_addressing_callback

Names the procedure that determines the total number of elements in a list. The call structure for this callback is:

list_element_count_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the number of elements in the list.

If your data structure does not have this element, you still must use this callback. In this case, simply return **{nop}** as the addressing expression and the transformation will count the elements by following all the pointers. This can be very time consuming.

list_element_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the data member of a list element. The call structure for this callback is:

list_element_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

list_element_next_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the next element of a list. The call structure for this callback is:

list_element_next_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

list_element_prev_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the previous element of a list. The call structure for this callback is:

list_element_prev_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

This property is optional. For example, you would not use it in a singly linked list.

list_end_value	Specifies if a list is terminated by NULL or the head of the list. Enter one of the following: **NULL** or **ListHead**

list_first_element_addressing_callback

Names the procedure that defines an addressing expression that specifies how to go from the head element of the list to the first element of the list. It is not always the case that the head element of the list is the first element of the list. The call structure for this callback is:

list_element_first_element_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

list_head_addressing_callback

Names the procedure that defines an addressing expression to obtain the head element of the linked list. The call structure for this callback is:

list_head_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

lower_bounds_callback

Names the procedure that obtains a lower bound value for the array type being transformed. For C/C++ arrays, this value is always 0. The call structure for this callback is:

lower_bounds_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

name

Contains a regular expression that checks to see if a symbol is eligible for type transformation. This regular expression must match the definition of the aggregate type (class) being transformed.

type_callback

The **type_callback** property is used in two ways.

(1) When it is used within a list or vector transformation, it names the procedure that determines the type of the list or vector element. The callback procedure takes one parameter, the symbol ID of the symbol that was validated during the callback to the procedure specified by the **validate_callback**. The call structure for this callback is:

type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

(2) When it is used within a struct transformation, it names the procedure that specifies the data type to be used when displaying the struct.

type_transformation_description

A string containing a description of what is being transformed; for example, you might enter "GNU Vector".

upper_bounds_callback

Names the procedure that defines an addressing expression that computes the extent (number of elements) in an array. The call structure for this callback is:

upper_bounds_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

validate_callback

Names a procedure that is called when a data type matches the regular expression specified in the **name** property. The call structure for this callback is:

validate_callback *id*

where *id* is the symbol ID of the symbol being validated.

Your callback procedure check the symbol's structure to insure that it should be transformed. While not required, most users will extract symbol information such as its type and its data members while validating the data type. The callback procedure must return a Boolean value, where *true* means the symbol is valid and can be transformed.

chapter 4

Batch Debugging Using tvscript

TotalView can run unattended if you start it using the **tvscript** command. This is called batch debugging because, like all batch programs, you do not need to use them interactively. In addition, you can invoke **tvscript** using **cron** to schedule debugging in the evening. In this way, you'll have reports waiting for you in the morning.

The actions that **tvscript** can perform can be quite complicated if you include these actions in a script file because this script file can also contain CLI and Tcl commands.

Here, for example, is how **tvscript** is invoked on a program:

```
tvscript \
  -create_actionpoint "method1=>display_backtrace" \
     -show_arguments \
  -create_actionpoint “method2#37=>display_backtrace \
     -show_locals -level 1 \
  -display_specifiers “nowshow_pid,noshow_tid” \
  -maxruntime “00:00:30” \
     filterapp -a 20
```

You can also execute MPI programs using tvscript. Here is a small example:

```
tvscript -mpi “Open MP” -tasks 4 \
     -create_actionpoint \
      “hello.c#14=>display_backtrace” \
     ~/tests/MPI_hello
```

All these tvscript command-line options (and others) are explained in the remainder of this chapter.

tvscript Command Syntax

The syntax for the **tvscript** command is:

tvscript [*options*] [*filename*] [–a *program_args*]

options	TotalView and **tvscript** command-line options. You can use any of the options described in Chapter 7, "*TotalView Command Syntax*," on page 241. **tvscript** command-line options are described in the next section.
filename	The program being debugged.
–a *program_args*	Arguments that TotalView sends to your program.

The MPI compilers that you can run under tvscript control are those that you see when you select an MPI with the list in the Parallel page of the **File > New Program** dialog box. The command-line options that are most often used with tvscript are:

- –**mpi**
- –**starter_args**
- –**nodes**
- –**np** or –**procs** or –**tasks**

You will find more information on these command-line options in Chapter 7, "*TotalView Command Syntax*," on page 241.

You can run the following MPIs using tvscript:

- Intel MPI
- LAM
- MPICH2
- Open MPI
- OSU MVAPICH2
- poe on AIX and Linux
- Sun CT7
- Sun MPI

tvscript Options

–**create_actionpoint** "*source_location_expr* [=>*action*1 [, *action*2] ...]"

Creates an action point at a source location using an expression. (These expressions are discussed in "*Action Point API*" on page 181.) When this action point is hit, **tvscript** can trigger one or more actions. You will need to add one –**create_watchpoint** command-line option for each action point being set.

See –**event_action** for information about actions.

–**event_action** "*event_action_list*"

Performs an action when an event occurs. Events represent a condition that wasn't anticipated. For example, **free_not_allocated** in the Memory Debugger. You can use more than one –**event_action** command-line option on the command line that invokes **tvscript**.

Here is how you enter an *event_action_list* :

event1=action1,event2=action2

or

event1=>action1,action2,action3

The following table lists the names of events that you can use:

Event Type	Event	Definition
General event	**any_event**	A generated event occurred.
Memory debugging event	**addr_not_at_start**	Program attempted to free a block using an incorrect address.
	alloc_not_in_heap	The memory allocator returned a block not in the heap; the heap may be corrupt.
	alloc_null	An allocation either failed or returned NULL; this usually means that the system is out of memory.
	alloc_returned_bad_-alignment	The memory allocator returned a misaligned block; the heap may be corrupt.
	any_memory_event	A memory event occurred.
	bad_alignment_argument	Program supplied an invalid alignment argument to the heap manager.
	double_alloc	The memory allocator returned a block currently being used; the heap may be corrupt.
	double_dealloc	Program attempted to free an already freed block.
	free_not_allocated	Program attempted to free an address that is not in the heap.
	guard_corruption	Program overwrote the guard areas around a block.
	hoard_low_memory_-threshold	Hoard low memory threshold crossed.

Event Type	Event	Definition
	realloc_not_allocated	Program attempted to reallocate an address that is not in the heap.
	rz_overrun	Program attempted to access memory beyond the end of an allocated block.
	rz_underrun	Program attempted to access memory before the start of an allocated block.
	rz_use_after_free	Program attempted to access a block of memory after it has been deallocated.
	rz_use_after_free_overrun	Program attempted to access memory beyond the end of a deallocated block.
	rz_use_after_free_underrun	Program attempted to access memory before the start of a deallocated block.
	termination_notification	The target is terminating.
Source code debugging event	actionpoint	A thread hit an action point.
	error	An error occurred.

When an event occurs, you can tell TotalView what actions it should perform:

Action Type	Action	Definition
Memory debugging actions	check_guard_blocks	Check all guard blocks and write violations into the log file.
	list_allocations	Write a list of all memory allocations into the log file.
	list_leaks	Write a list of all memory leaks into the log file.
	save_html_heap_status_source_view	Generate and save an HTML version of the Heap Status Source View Report.
	save_memory_debugging_file	Generate and save a memory debugging file.
	save_text_heap_status_source_view	Generate and save a text version of the Heap Status Source View Report.

Action Type	Action	Definition
Source code debugging actions	**display_backtrace** [**–level** *level-num*] [*num_levels*] [*options*]	Write the current stack backtrace into the log file. **–level** *level-num* set the level at which information starts being logged. *num-levels* restricts output to this number of levels in the call stack. f you do not set a level, **tvscript** displays all levels in the call stack. *options* is one or more of the following: **–[no]show_arguments** **–[no]show_fp** **–[no]show_fp_registers** **–[no]show_image** **–[no]show_locals** **–[no]show_pc** **–[no]show_registers**
	print [**–slice** {*slice_exp*}] {*variable* | *exp*}	Write the value of a variable or an expression into the log file. If the variable is an array, the **–slice** option tells TotalView to you limit the amount of data to that defined by *slice_exp*. A slice expression is a way to define the slice such as **var[100:130]** in C and C++. (This tells TotalView to display all values from var[100] to **var[130]**.) If you wanted to display every fourth value, you'd add an additional argument; for example, **var[100:130:4]**. For additional information, see Chapter 14 "*Examining Arrays*" in the *TotalView Users Guide*.

–display_specifiers "*display_specifiers_list*"

By default, **tvscript** writes all of the information in the following table to the log file. You can exclude information by using one of the following specifiers. You can eliminate

Type of Specifier	Specifier	Display ...
General display specifiers	**noshow_fp**	Do not show the frame pointer (FP)
	noshow_image	Do not show the process/library in backtrace
	noshow_pc	Do not show the program counter (PC)
	noshow_pid	Do not show the system process ID with process information
	noshow_rank	Do not show the rank of a process, which is only shown for a parallel process
	noshow_tid	Do not show the thread ID with process information
Memory debugging display specifiers	**noshow_allocator**	Do not show the allocator for the address space
	noshow_backtrace	Do not show the backtraces for memory blocks
	noshow_backtrace_id	Do not show the backtrace ID for memory blocks
	noshow_block_address	Do not show the memory block start and end addresses
	noshow_flags	Do not show the memory block flags
	noshow_guard_id	Do not show the guard ID for memory blocks
	noshow_guard_settings	Do not show the guard settings for memory blocks
	noshow_leak_stats	Do not show the leaked memory block statistics
	noshow_owner	Do not show the owner of the allocation
	noshow_red_zones_settings	Do not show the Red Zone entries for allocations (and deallocations) for the address space

–memory_debugging

Enables memory debugging and memory event notification. You must use this option if you use any of the options that begin with **–mem**. These options are TotalView command line options as they can be invoked directly by TotalView.

–mem_detect_leaks

Performs leak detection before generating memory information.

–mem_detect_use_after_free

Tests for use after memory is freed.

–mem_guard_blocks
: Adds guard blocks to an allocated memory block.

–mem_hoard_freed_memory
: Holds onto freed memory rather than return it to the heap.

–mem_hoard_low_memory_threshold *nnnn*
: Sets the low memory threshold amount. When memory falls below this amount an event will be fired.

–mem_paint_all
: Paints memory blocks with a bit pattern when a memory is allocated and when it is deallocated.

–mem_paint_on_alloc
: Paints memory blocks with a bit pattern when a memory block is allocated.

–mem_paint_on_dealloc
: Paints memory blocks with a bit pattern when a memory block is deallocated.

–mem_red_zones_overruns
: Turns on testing for Red Zone overruns.

–mem_red_zones_size_ranges min:max,min:max,...
: Defines the memory allocations ranges for which Red Zones are in effect. Ranges can be specified as follows:

x:y allocations from x to y

:y allocations from 1 to y

x: allocations of x and higher

x allocation of x

–mem_red_zones_underruns
: Turns on testing for Red Zone underruns.

–maxruntime "*hh:mm:ss*"
: Specifies how long the script can run.

–script_file *script_file*
: Names a file containing **tvscript** API calls and Tcl callback procedures that you create. See for more information.

tvscript Example: The following example is similar to shown in "*Batch Debugging Using tvscript*" on page 173.

```
tvscript \
   -create_actionpoint "method1=>display_backtrace    \
                   -show_arguments"                   \
   -create_actionpoint "method2#37=>display_backtrace \
                   -show_locals -level 1"             \
   -event_action "error=>display_backtrace            \
                   -show_arguments -show_locals"      \
   -display_specifiers "noshow_pid,noshow_tid"        \
   -maxruntime "00:00:30"                             \
   filterapp -a 20
```

This script tells **tvscript** to perform the following actions:

- Create an actionpoint at the beginning of **method1**. When **tvscript** reaches that breakpoint, it logs a backtrace and the method's arguments.
- Create an actionpoint at line 37 of **method2**. When **tvscript** reaches this line, it logs a backtrace and the local variables. The backtrace information starts at level 1.
- Log the backtrace, the current routine's arguments, and its local variables when an error event occurs.
- Exclude the process ID and thread ID from the information that **tvscript** logs.
- Limit **tvscript** execution time to 30 sections.
- Name the program being debugged and passes a value of 20 to the application.

tvscript External Script Files

The previous section discussed the command-line options that you use when invoking the **tvscript** command. A second method is to place commands in a file and feed these commands into **tvscript** using the **–script_file** command-line option. Within this script file, you can use Tcl to create more complex actions when events occur. The following sections describe the functions that you can use within a CLI file.

Logging Functions API

tvscript_log *msg*	Log a message to the log file set up by **tvscript**.
tvscript_slog *msg*	Log a message to the summary log file set up by **tvscript**.

Process Functions API

tvscript_get_process_property *process_id property*
Get the value of a property about the process.

The properties you can name are the same as those used with the **TV::process** command. See "*process*" on page 145 for more information.

Thread Functions API

tvscript_get_thread_property *thread_id property*
Get the value of a property about the thread.

The properties you can name are the same as those used with the **TV::thread** command. See "*thread*" on page 165 for more information.

Action Point API

tvscript_add_actionpoint_handler *actionpoint_id actionpoint_handler*

Registers a procedure handler that is called when the action point associated with *actionpoint_id* is hit. This *actionpoint_id* is the value that is returned from the **tvscript_create_actionpoint** routine. The value of *actionpoint_handler* is the string naming the procedure.

When **tvscript** calls an action point handler procedure, it passes one argument. This argument contains a list that you must convert into an array. The array indices are as follows:

event—The event that occurred, which is action point.

process_id—The ID of the process that hit the action point

thread_id—The ID of the thread that hit the action point.

actionpoint_id—The ID of the action point that was hit.

actionpoint_source_loc_expr—The initial source location expression used to create the action point.

tvscript_create_actionpoint *source_loc_expr*

Creates an action point using a source location expression.

This procedure returns an action point ID that you can use in a **tvscript_add_actionpoint_handler** procedure.

source_loc_expr Set a breakpoint at the line specified by *source_loc_expr* or an absolute address. For example:

- **[[##image#]filename#]line_number**

  Indicates all addresses at this line number.
- A function signature; this can be a partial signature.

  Indicates all addresses that are the addresses of functions matching *signature*. If parts of a function signature are missing, this expression can match more than one signature. For example, "**f**" matches "**f(void)**" and "**A::f(int)**". You cannot specify a return type in a signature.

You can also enter a source location expression that are sets of addresses using the class and virtual keywords. For example:

class *class_name*	Names a set containing the addresses of all member functions of class *class_name*.
virtual *class*::*signature*	Names the set of addresses of all virtual member functions that match *signature*, and which are in the classes or derived from the class.

If the expression evaluates to a function that has multiple overloaded implementations, TotalView sets a barrier on each of the overloaded functions.

Event API

tvscript_add_event_handler *event event_handler*

Registers a procedure handler that is called when the named event occurs. The event is either **error** or **actionpoint**.

When **tvscript** calls an event handler procedure, it passes one argument to it. This argument contains a list that you must convert into an array.

error When any error occurs, the array has the following indices:

event—The event, which is set to **error**.

process_id— The ID of the process that hit the action point.

thread_id—The ID of the thread that hit the action point.

error_message—A message describing the error that occurred.

actionpoint

When any action point is hit, the array has the following indices:

event—The event, which is set to **actionpoint**.

process_id—The ID of the process that hit the action point

thread_id—The ID of the thread that hit the action point.

actionpoint_id—The ID of the action point that was hit.

actionpoint_source_loc_expr—The initial source location expression used to create the action point.

Example tvscript Script File

The following information is passed to tvscript as follows:

tvscript –script_file *script_file*

This script installs an error handler and an actionpoint handler. When an error is encountered during execution, **tvscript** passes an array of information to the error handler. Similarly, when an action point is hit, it passes an array of information to the action point handler. These arrays are described in "Event API" on page 182.

```
# Get the process so we have some information about it
tvscript_log "PID: \
             [tvscript_get_process_property 1 "syspid"]";
tvscript_log "Status:  \
             [tvscript_get_process_property 1 "state"]";
tvscript_log "Executable:  \
             [tvscript_get_process_property 1 "executable"]";

##############################################################
proc error_handler {error_data} {
    tvscript_log "Inside error_handle: $error_data"

    # Change the incoming list into an array.
    # It contains the following indices:
    #    process_id
    #    thread_id
    #    error_message
    array set error_data_array $error_data

    # Get the process so we have some information about it
    temp = [tvscript_get_process_property \
                 $error_data_array(process_id) "syspid"];
    tvscript_log "             Process ID: $temp";

    temp = [tvscript_get_thread_property \
                 $error_data_array(thread_id) "systid"];
     tvscript_log "            Thread ID: $temp";

    temp = $error_data_array(error_message);
     tvscript_log "            Error Message: $temp";

}

##############################################################
# Action point handlers

proc l1_actionpoint_handler {event_data} {
    tvscript_log "Inside l1_actionpnt_handler:  $event_data"
    tvscript_slog "Inside l1_actionpnt_handler:  $event_data"

    # Change the incoming list into an array.
    # It contains the following indices:
    #   actionpoint_id
    #   actionpoint_source_loc_expr
    #   event
    #   process_id
    #   thread_id
    array set event_data_array $event_data

    # Get the process so we have some information about it
    temp = [tvscript_get_process_property \
             $event_data_array(process_id) "syspid"];
    tvscript_log "            Process ID: $temp";

    temp = [tvscript_get_thread_property \
             $event_data_array(thread_id) "systid"];
    tvscript_log "             Thread ID: $temp";

    temp = [tvscript_get_process_property \
             $event_data_array(process_id) "state"];
    tvscript_log "                Status: $temp";

    temp = [tvscript_get_process_property \
             $event_data_array(process_id) "executable"]
    tvscript_log "            Executable: $temp";

    temp = $event_data_array(actionpoint_source_loc_expr)
    tvscript_log "Action point Expression: $temp"

    tvscript_log "Value of i:"
```

```
   set output [capture "dprint i"]
   tvscript_log $output
}

########################################################
# Event handlers

proc generic_actionpoint_event_handler {actionpoint_data} {
   tvscript_log \
      "Inside generic_actionpoint_event_handler: “
   tvscript_log $actionpoint_data
   tvscript_slog "Inside generic_actionpoint_event_handler: “
   tvscript_slog $actionpoint_data

   # Change the incoming list into an array.
   # It contains the following indices:
   #   actionpoint_id
   #   actionpoint_source_loc_expr
   #   event
   #   process_id
   #   thread_id
   array set actionpnt_data_array $actionpoint_data

   temp = $actionpnt_data_array(process_id)
   tvscript_log "            Process ID: $temp"

   temp = $actionpnt_data_array(thread_id)
   tvscript_log "            Thread ID: $temp"

   temp = $actionpnt_data_array(actionpoint_id)
   tvscript_log "       Action Point ID: $temp"

   temp = $actionpnt_data_array(actionpoint_source_loc_expr)
   tvscript_log "Action Point Expression: "
}

##############################################################
# Add event handlers

# Setup action points and action point handlers
set actionpoint_id [tvscript_create_actionpoint "l1"]
tvscript_add_actionpoint_handler $actionpoint_id \
   "l1_actionpoint_handler"

# Setup a generic actionpoint handler
tvscript_add_event_handler "actionpoint" \
   "generic_actionpoint_event_handler"

########################################################
# Add error handler
tvscript_add_event_handler "error" "error_handler"
```

chapter 5

TotalView Variables

This chapter contains a list of all CLI and TotalView variables. This chapter has three sections, each corresponding to a CLI namespace, as follows:

- Top-Level (::) Namespace
- TV:: Namespace
- TV::MEMDEBUG:: Namespace
- TV::GUI:: Namespace

Top-Level (::) Namespace

ARGS(*dpmid*)

Contains the arguments that TotalView passes to the process with TotalView ID *dpid* the next time you start the process.

Permitted Values:	A string
Default:	None

ARGS_DEFAULT

Contains the argument passed to a new process when no **ARGS(*dpid*)** variable is defined.

Permitted Values:	A string
Default:	None

BARRIER_STOP_ALL

Contains the value for the "stop_when_done" property for newly created action points. This property tells TotalView what else it should stop when a barrier point is satisfied. This property also tells TotalView what else it should stop when a thread encounters this action point. You can also set this value using the **When barrier hit, stop** value in the **Action Points** Page of the **File > Preferences** Dialog Box. The values that you can use are as follows:

group	TotalView will stop all processes in a thread's control group when a thread reaches a barrier created using this as a default.
process	TotalView will stop the process in which the thread is running when a thread reaches a barrier created using this default.
none	TotalView just stops the thread that hit a barrier created using this default.

This variable is the same as the **TV::barrier_stop_all** variable.

Permitted Values:	**group**, **process**, or **thread**
Default:	**group**

BARRIER_STOP_WHEN_DONE

Contains the default value that TotalView uses when a barrier point is satisfied. You can also set this value if you use the **–stop_when_done** command-line option or the **When barrier done, stop** value in the **Action Points** Page of the **File > Preferences** Dialog Box. The values you can use are as follows:

group	When a barrier is satisfied, TotalView stops all processes in the control group.
process	When a barrier is satisfied, TotalView stops the processes in the satisfaction set.
none	TotalView only stops the threads in the satisfaction set; other threads are not affected. For process barriers, there is no difference between **process** and **none**.

In all cases, TotalView releases the satisfaction set when the barrier is satisfied.

This variable is the same as the **TV::barrier_stop_when_done** variable.

Permitted Values:	**group**, **process**, or **thread**
Default:	**group**

CGROUP(*dpid*)

Contains the control group for the process with the TotalView ID *dpid*. Setting this variable moves process *dpid* into a different control group. For example, the following command moves process 3 into the same group as process 1:

```
dset CGROUP(3) $CGROUP(1)
```

Permitted Values:	A number
Default:	None

COMMAND_EDITING

Enables some Emacs-like commands that you can use while editing text in the CLI. These editing commands are always available in the CLI window of the TotalView GUI. However, they are only available in the stand-alone CLI if the terminal in which you are running it supports cursor positioning and clear-to-end-of-line. The commands that you can use are:

^A: Moves the cursor to the beginning of the line.

^B: Moves the cursor one character backward.

^D: Deletes the character to the right of cursor.

^**E**: Moves the cursor to the end of the line.

^**F**: Moves the cursor one character forward.

^**K**: Deletes all text to the end of line.

^**N**: Retrieves the next entered command (only works after ^P).

^**P**: Retrieves the previously entered command.

^**R** or ^**L**: Redraws the line.

^**U**: Deletes all text from the cursor to the beginning of the line.

Rubout or **Backspace**: Deletes the character to the left of the cursor.

Permitted Values:	true or false
Default:	false

EXECUTABLE_PATH

Contains a colon-separated list containing the directories that TotalView searches when it looks for source and executable files.

Permitted Values:	Any directory or directory path. To include the current setting, use $EXECUTABLE_PATH.
Default:	. (dot)

EXECUTABLE_SEARCH_MAPPINGS

Contains pairs of regular expressions and replacement and replacement strings—these replacements are called *mappings*—separated by colons. TotalView applies these mappings to the search paths before it looks for source, object, and program files.

The syntax for mapping strings is:

+regular_exp+=+replacement+ :+regular_exp+=+replacement+

This example shows two pairs, each delimited by a colon (":"). Each element within a pair is delimited by any character except a colon. The first character you enter is the delimiter. This example uses a "+" as a delimiter. (Traditionally, forward slashes are used as delimiters. This is not a good idea as a forward slash is also used to separate components of a a pathname. For example, /home/my_dir contains forward slashes.)

When you enter information, be careful when you include special characters. When you do, these characters must follow standard Tcl rules and conventions.

Here is an example:

```
dset EXECUTABLE_SEARCH_MAPPINGS
   {+^/nfs/compiled/u2/(.*)$+ = +/nfs/host/u2/\1+ }
```

This expression tells TotalView that if it finds a directory named /nfs/compiled/u2/project/src1 in the expanded search path, it will become /nfs/host/u2/project/src1 after applying this mapping.

Default:	{}

EXECUTABLE_SEARCH_PATH

Contains a list of paths, with paths separated by a colon. that TotalView searches when it looks for your executables. For information, consult the online help.

Permitted Values:	Any directory or directory path.
Default:	${EXECUTABLE_PATH};${$PATH}:.

GROUP(*gid*)

Contains a list containing the TotalView IDs for all members in group *gid*.

The first element in the list indicates what kind of group it is, as follows:

control	The group of all processes in a program
lockstep	A group of threads that share the same PC
process	A user-created process group
share	The group of processes in one program that share the same executable image
thread	A user-created thread group
workers	The group of worker threads in a program

Elements that follow are either *pids* (for process groups) or *pid*.*tid* pairs (for thread groups).

The *gid* is a simple number for most groups. In contrast, a lockstep group's ID number is of the form *pid.tid*. Thus, **GROUP(2.3)** contains the lockstep group for thread 3 in process 2. Note, however, that the CLI will not display lockstep groups when you use **dset** with no arguments—they are hidden variables.

The **GROUP(*id*)** variable is read-only.

Permitted Values:	A Tcl array of lists indexed by the group ID. Each entry contains the members of one group.
Default:	None

GROUPS

Contains a list that contains all TotalView groups IDs. Lockstep groups are not contained in this list. This is a read-only value and cannot be set.

Permitted Values:	A Tcl list of IDs.

LINES_PER_SCREEN

Defines the number of lines shown before the CLI stops printing information and displays its **more** prompt. The following values have special meaning:

0	No *more* processing occurs, and the printing does not stop when the screen fills with data.
NONE	This is a synonym for 0.
AUTO	The CLI uses the tty settings to determine the number of lines to display. This may not work in all cases. For example, Emacs sets the **tty** value to 0. If **AUTO** works improperly, you will need to explicitly set a value.

Permitted Values:	A positive integer, or the **AUTO** or **NONE** strings
Default:	Auto

MAX_LEVELS Defines the maximum number of levels that the **dwhere** command will display.

Permitted Values:	A positive integer
Default:	512

MAX_LIST Defines the number of lines that the **dlist** command will display.

Permitted Values:	A positive integer
Default:	20

OBJECT_SEARCH_MAPPINGS Contains pairs of regular expressions and replacement and replacement strings—these replacements are called *mappings*—separated by colons. TotalView applies these mappings to the search paths before it looks for source, object, and program files. For more information, see EXECUTABLE_SEARCH_MAPPINGS.

Default:	{ }

OBJECT_SEARCH_PATH Contains a list of paths, with paths separated by a colon. that TotalView searches when it looks for your program's object files. For information, consult the online help.

Permitted Values:	Any directory or directory path.
Default:	${COMPILATION_DIRECTORY): ${EXECUTABLE_PATH}: ${EXECUTABLE_DIRECTORY}: $links{${EXECUTABLE_DIRECTORY}}: .:${TOTALVIEW_SRC}

PROCESS(*dpid*) Contains a list of information associated with a dpid. This is a read-only value and cannot be set.

Permitted Values:	An integer
Default:	None

PROMPT Defines the CLI prompt. If you use brackets ([ ]) in the prompt, TotalView assumes the information within the brackets is a Tcl command and evaluates this information before it creates the prompt string.

Permitted Values:	Any string. If you wish to access the value of PTSET, you must place the variable within brackets; that is, [dset PTSET].
Default:	{[dfocus]> }

PTSET Contains the current focus. This is a read-only value and cannot be set.

Permitted Values:	A string
Default:	d1.<

SGROUP(*pid*)

Contains the group ID of the share group for process *pid*. TotalView decides which share group this is by looking at the control group for the process and the executable associated with this process. You cannot directly modify this group.

Permitted Values:	A number
Default:	None

SHARE_ACTION_POINT

Indicates the scope in which TotalView places newly created action points. In the CLI, this is the **dbarrier**, **dbreak**, and **dwatch** commands. If this Boolean value is **true**, newly created action point are shared across the group. If it is **false**, a newly created action point is only active in the process in which it is set.

As an alternative to setting this variable, you can select the **Plant in share group** check box in the **Action Points** Page in the **File > Preferences** Dialog Box. You can override this value in the GUI by using selecting the **Plant in share group** checkbox in the **Action Point > Properties** Dialog Box.

Permitted Values:	**true** or **false**
Default:	**true**

SHARED_LIBRARY_SEARCH_MAPPINGS

Contains pairs of regular expressions and replacement strings (*mappings*), separated by colons. TotalView applies these mappings to the search paths before it looks for shared library files.

Default:	{}

SHARED_LIBRARY_SEARCH_PATH

Contains a list of paths, each separated by a colon, that TotalView searches when it looks for your program's shared library files.

Permitted Values:	Any directory or directory path.
Default:	${EXECUTABLE_PATH}:

SOURCE_SEARCH_MAPPINGS

Contains pairs of regular expressions and replacement strings—these replacements are called *mappings*—separated by colons. TotalView applies these mappings to the search paths before it looks for source, object, and program files. For more information, see **EXECUTABLE_SEARCH_MAPPINGS**.

Default:	{}

SOURCE_SEARCH_PATH

Contains a list of paths, with paths separated by a colon. that TotalView searches when it looks for your program's source files. For information, consult the online help.

Permitted Values:	Any directory or directory path.
Default:	${COMPILATION_DIRECTORY}: ${EXECUTABLE_PATH}: ${EXECUTABLE_DIRECTORY}: ${links{${EXECUTABLE_DIRECTORY}): .:${TOTALVIEW_SRC}

STOP_ALL Indicates a default property for newly created action points. This property tells TotalView what else it should stop when it encounters this action point. The values you can set are as follows:

group	Stops the entire control group when the action point is hit.
process	Stops the entire process when the action point is hit.
thread	Only stops the thread that hit the action point. Note that **none** is a synonym for **thread**.

Permitted Values:	**group**, **process**, or **thread**
Default:	**process**

TAB_WIDTH Indicates the number of spaces used to simulate a tab character when the CLI displays information.

Permitted Values:	A positive number. A value of –1 indicates that the CLI does not simulate tab expansion.
Default:	8

THREADS(*pid*) Contains a list of all threads in the process *pid*, in the form **{pid.1 pid.2 ...}**. This is a read-only variable and cannot be set.

Permitted Values:	A Tcl list
Default:	None

TOTALVIEW_ROOT_PATH Names the directory in which the TotalView executable is located. This is a read-only variable and cannot be set. This variable is exported as **TVROOT**, and is can be used in launch strings.

Permitted Values:	The location of the TotalView installation directory

TOTALVIEW_TCLLIB_PATH Contains a list containing the directories in which the CLI searches for TCL library components.

Permitted Values:	Any valid directory or directory path. To include the current setting, use **$TOTALVIEW_TCLLIB_PATH**.
Default:	The directory containing the CLI's Tcl libraries

TOTALVIEW_VERSION Contains the version number and the type of computer architecture upon which TotalView is executing. This is a read-only variable and cannot be set.

Permitted Values:	A string containing the platform and string
Default:	Platform-specific

VERBOSE Controls the error message information displayed by the CLI. The values for this variable can be:

info	Prints errors, warnings, and informational messages. Informational messages include data on dynamic libraries and symbols.
warning	Only print errors and warnings.
error	Only print error messages.

silent	Does not print error, warning, and informational messages. This also shuts off the printing of results from CLI commands. This should only be used when the CLI is run in batch mode.

Permitted Values: **info**, **warning**, **error**, and **silent**
Default: **info**

WGROUP(*pid*)

The group ID of the thread group of worker threads associated with the process *pid*. This variable is read-only.

Permitted Values: A number
Default: None

WGROUP(*pid.tid*)

Contains one of the following:

- The group ID of the workers group in which thread *pid.tid* is a member
- 0 (zero), which indicates that thread *pid.tid* is not a worker thread

Storing a nonzero value in this variable marks a thread as a worker. In this case, the returned value is the ID of the workers group associated with the control group, regardless of the actual nonzero value that you had assigned to it.

Permitted Values: A number representing the *pid.tid*
Default: None

TV:: Namespace

TV::aix_use_fast_ccw

This variable is defined only on AIX, and is a synonym for the platform-independent variable **TV::use_fast_wp**, for TotalView script backward compatibility. See **TV::use_fast_wp** for more information.

TV::aix_use_fast_trap

This variable is defined only on AIX, and is a synonym for the platform-independent variable **TV::use_fast_trap**, for TotalView script backward compatibility. See **TV::use_fast_trap** for more information.

TV::ask_on_cell_spu_image_load

If set to true, TotalView might ask you if you want to stop the process when a Cell SPU image is loaded.

If set to false, TotalView does not stop execution when a Cell SPU image is loaded.

Permitted Values: **true** or **false**
Default: **true**

TV::ask_on_dlopen

Setting this variable to **true** tells TotalView that it should ask you about stopping processes that use the **dlopen** or **load** (AIX only) system calls dynamically load a new shared library.

If this is set to **false**, TotalView will not ask about stopping a process that dynamically loads a shared library.

Permitted Values:	**true** or **false**
Default:	**true**

TV::auto_array_cast_bounds

Indicates the number of array elements that are displayed when the **TV::auto_array_cast_enabled** variable is set to **true**. This is the variable set by the **Bounds** field of the **Pointer Dive** Page in the **File > Preferences** Dialog Box.

Permitted Values:	An array specification
Default:	[10]

TV::auto_array_cast_enabled

When this is set to **true**, TotalView will automatically dereference a pointer into an array. The number of array elements is indicated in the **TV::auto_array_cast_bounds** variable. This is the variable set by the **Cast to array with bounds** checkbox of the **Pointer Dive** Page in the **File > Preferences** Dialog Box.

Permitted Values:	**true** or **false**
Default:	**false**

TV::auto_deref_in_all_c

Tells TotalView if and how it should dereference C and C++ pointers when you perform a **View > Dive in All** operation, as follows:

yes_dont_push	While automatic dereferencing will occur, you can't use the **Undive** command to see the undereferenced value when performing a **Dive in All** operation.
yes	You will be able to use the **Undive** control to see undereferenced values.
no	Do not automatically dereference values when performing a **Dive in All** operation.

This is the variable set when you select the **Dive in All** element in the **Pointer Dive** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**no**, **yes**, or **yes_dont_push**
Default:	**no**

TV::auto_deref_in_all_fortran

Tells TotalView if and how it should dereference Fortran pointers when you perform a **Dive in All** operation, as follows:

yes_dont_push	While automatic dereferencing will occur, you can't use the **Undive** command to see the undereferenced value when performing a **Dive in All** operation.
yes	You will be able to use the **Undive** control to see undereference values.
no	Do not automatically dereference values when performing a **Dive in All** operation.

This is the variable set when you select the **Dive in All** element in the **Pointer Dive** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**no**, **yes**, or **yes_dont_push**
Default:	**no**

TV::auto_deref_initial_c

Tells TotalView if and how it should dereference C pointers when they are displayed, as follows:

yes_dont_push	While automatic dereferencing will occur, you can't use the **Undive** command to see the undereferenced value.
yes	You will be able to use the **Undive** control to see undef-erenced values.
no	Do not automatically dereference values.

This is the variable set when you select the **initially** element in the **Pointer Dive** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**no**, **yes**, or **yes_dont_push**
Default:	**no**

TV::auto_deref_initial_fortran

Tells TotalView if and how it should dereference Fortran pointers when they are displayed, as follows:

yes_dont_push	While automatic dereferencing will occur, you can't use the **Undive** command to see the undereferenced value.
yes	You will be able to use the **Undive** control to see undef-erenced values.
no	Do not automatically dereference values.

This is the variable set when you select the **initially** element in the **Pointer Dive** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**no**, **yes**, or **yes_dont_push**
Default:	**no**

TV::auto_deref_nested_c

Tells TotalView if and how it should dereference C pointers when you dive on structure elements:

yes_dont_push	While automatic dereferencing will occur, you can't use the **Undive** command to see the undereferenced value.
yes	You will be able to use the **Undive** control to see undef-erenced values.
no	Do not automatically dereference values.

This is the variable set when you select the **from an aggregate** element in the **Pointer Dive** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**no**, **yes**, or **yes_dont_push**
Default:	**yes_dont_push**

TV::auto_deref_nested_fortran

Tells TotalView if and how it should dereference Fortran pointers when they are displayed, as follows:

yes_dont_push	While automatic dereferencing will occur, you can't use the **Undive** command to see the undereferenced value.
yes	You will be able to use the **Undive** control to see undef-erenced values.
no	Do not automatically dereference values.

This is the variable set when you select the **from an aggregate** element in the **Pointer Dive** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**no**, **yes**, or **yes_dont_push**
Default:	**yes_dont_push**

TV::auto_load_breakpoints

Setting this variable to **true** tells TotalView that it should automatically load action points from the file named *filename*.**TVD.v3breakpoints** where *filename* is the name of the file being debugged. If the variable is set to **false**, TotalView does not automatically load your breakpoints. If you set this to **false**, you can still load breakpoints if you use the **Action Point > Load All** or the **dactions –load** command.

Permitted Values:	**true** or **false**
Default:	**true**

TV::auto_read_symbols_at_stop

Setting this variable to **false** tells TotalView that it should not automatically read symbols if execution stops when the program counter is in a library whose symbols were not read. Setting it to **true** tells TotalView that it should read in loader and debugging symbols. You would set it to **false** if you have prevented symbol reading using either the **TV::dll_read_loader_symbols_only** or **TV::dll_read_no_symbols** variables (or the preference within the GUI) and reading these symbols is both unnecessary and would affect performance.

Permitted Values:	**true** or **false**
Default:	**true**

TV::auto_save_breakpoints

Setting this variable to **true** tells TotalView that it should automatically write information about breakpoints to a file named *filename*.**TVD.v3breakpoints** where *filename* is the name of the file being debugged. Information about watchpoints is not saved.

TotalView writes this information when you exit from TotalView. If this variable is set to **false**, you can explicitly save this information by using the **Action Point > Save All** or the **dactions –save** command.

Permitted Values:	**true** or **false**
Default:	**false**

TV::barrier_stop_all

Contains the value for the "stop_all" property for newly created action points. This property tells TotalView what else it should stop when a thread encounters this action point. You can also set this value using the **–stop_all** command-line option or the **When barrier hit, stop** value in the **Action Points** page of the **File > Preferences** dialog box. The values that you can use are as follows:

group	TotalView will stop all processes in a thread's control group when a thread reaches a barrier created using this as a default.
process	TotalView will stop the process in which the thread is running when a thread reaches a barrier created using this default.

thread	TotalView just stops the thread that hit a barrier created using this default.

This variable is the same as the **BARRIER_STOP_ALL** variable.

Permitted Values:	**group**, **process**, or **thread**
Default:	**group**

TV::barrier_stop_when_done

Contains the value for the "stop_when_done" property for newly created action points. This property tells TotalView what else it should stop when a barrier point is satisfied. You can also set this value if you use the **–stop_when_done** command-line option or the **When barrier done, stop** value in the **Action Points** page of the **File > Preferences** dialog box. The values you can use are as follows:

group	When a barrier is satisfied, TotalView stops all processes in the control group.
process	When a barrier is satisfied, TotalView stops the processes in the satisfaction set.
thread	TotalView only stops the threads in the satisfaction set; other threads are not affected. For process barriers, there is no difference between **process** and **none**.

In all cases, TotalView releases the satisfaction set when the barrier is satisfied.

This variable is the same as the **BARRIER_STOP_WHEN_DONE** variable.

Permitted Values:	**group**, **process**, or **thread**
Default:	**group**

TV::bluegene_io_interface

If the Bluegene front-end cannot resolve the network name, you must initialize this variable (or set it as a command-line option). By default, TotalView assumes that it can resolve the address as follows:

front_end_hostname-**io**

For example, if the front-end hostname is fred, TotalView assumes that the servers are connecting to **fred-io**.

Permitted Values:	A string
Default:	none

TV::bluegene_server_launch_string

Defines the launch string TotalView uses when it launches **tvdsvr** processes on I/O nodes.

Permitted Values:	A string
Default:	**–callback %L –set_pw %P –verbosity %V %F**

TV::bluegene__launch_timeout

Specifies the number of seconds that TotalView waits to hear back from the TotalView Debugger Server (**tvdsvr**) that it launches.

Permitted Values:	An integer from 1 to 3600 (1 hour)
Default:	240

TV::bulk_launch_base_timeout

Defines the base timeout period used when TotalView executes a bulk launch.

Permitted Values: A number from 1 to 3600 (1 hour)
Default: 20

TV::bulk_launch_enabled

When this is set to **true**, tells TotalView that it should use its bulk launch features when it automatically launches the TotalView Debugger Server (**tvdsvr**) for remote processes.

Permitted Values: **true** or **false**
Default: **false**

TV::bulk_launch_incr_timeout

Defines the incremental timeout period that TotalView waits for process to launch when it automatically launches the TotalView Debugger Server (**tvdsvr**) using the bulk server feature.

Permitted Values: A number from 1 to 3600 (1 hour)
Default: 10

TV::bulk_launch_string

Defines the command that will be used to launch the TotalView Debugger Server (**tvdsvr**) when remote processes are created. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values: A string, usually contained within braces **{}**
Default: The default value depends upon the platform—use the **dset** command to see what this default is

TV::bulk_launch_tmpfile1_header_line

Defines the header line used in the first temporary file when TotalView does a bulk server launch operation. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values: A string
Default: None

TV::bulk_launch_tmpfile1_host_lines

Defines the host line used in the first temporary file when TotalView performs a bulk server launch operation. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values: A string
Default: **%R**

TV::bulk_launch_tmpfile1_trailer_line

Defines the trailer line used in the first temporary file when TotalView performs a bulk server launch operation. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values: A string
Default: None

TV::bulk_launch_tmpfile2_header_line

Defines the header line used in the second temporary file when TotalView performs a bulk server launch operation. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values: A string
Default: None

TV::bulk_launch_tmpfile2_host_lines

Defines the host line used in the second temporary file when TotalView does a bulk server launch operation.For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values:	A string
Default:	{tvdsvr –working_directory %D –callback %L –set_pw %P –verbosity %V}

TV::bulk_launch_tmpfile2_trailer_line

Defines the trailer line used in the second temporary file when TotalView does a bulk server launch operation. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values:	A string
Default:	None

TV::c_type_strings

When set to **true**, TotalView uses C type string extensions when it displays character arrays. When **false**, TotalView uses its string type extensions.

Permitted Values:	**true** or **false**
Default:	**true**

TV::cell_spu_image_ignore_regexp

If set to a non-empty string and **TV::ask_on_cell_spu_image_load** is **true**, TotalView matches the SPU image's name with the regular expression. If it matches, TotalView does not ask if you want to stop the process and allows the process to continue running after loading the SPU image.

If the image name does not match this regular expression or the regular expression contained within **TV::cell_spu_images_stop_regexp**, TotalView asks if it should stop the process unless you've answered the stop to set breakpoint question by pressing **No** (or the equivalent from within the CLI).

Permitted Values:	A regular expression
Default:	{}

TV::cell_spu_images_stop_regexp

If set to a non-empty string and **TV::ask_on_cell_spu_image_load** is **true**, TotalView matches the SPU image's name with the regular expression. If it matches, TotalView asks if you want to stop the process.

If the image name does not match this regular expression or the regular expression contained within **TV::cell_spu_images_ignore_regexp**, TotalView asks if it should stop the process unless you've answered the stop to set breakpoint question by pressing **No** (or the equivalent from within the CLI).

Permitted Values:	A regular expression
Default:	{}

TV::cell_spurs_jm_name

Contains a strings that names the file that contain the symbols for the "jm" SPURS job policy module. When TotalView detects an embedded SPURS kernel image being loaded into an SPU context, it extracts the GUIDs of the policy modules from the kernel, and searches for either the default SPU ELF image file, which is **spurs_jm.elf** or the file named by this variable.

Permitted Values:	An ELF file name
Default:	**spurs_jm.elf**

TV::cell_spurs_kernel_dll_regexp

Defines a regular expression that matches the image path component name of the SPURS kernel SPU ELF image embedded in the **libspurs.so** DLL.

When TotalView sees a new image loaded into an SPU thread by **libspe** or **libspe2**, it checks if the image path component name matches this variable. If it does, TotalView handles the SPURS kernel library in a different way. You may need to change this regular expression to match the name of your SPURS kernel if it is embedded in a shared library other than **libspurs.so** or if the name of the SPURS kernel is different than **spurs_kernel.elf**.

Permitted Values:	A regular expression
Default:	**{/libspurs\.so\(spurs_kernel\.elf@[0-9]+\)$}**

TV::cell_spurs_ss_name

Contains a strings that names the file that contain the symbols for the "ss" SPURS system service policy module. When TotalView detects an embedded SPURS kernel image being loaded into an SPU context, it extracts the GUIDs of the policy modules from the kernel, and searches for either the default SPU ELF image file, which is **spurs_tss.elf** or the file named by this variable.

Permitted Values:	An ELF file name
Default:	**spurs_ss.elf**

TV::cell_spurs_tm_name

Contains a strings that names the file that contain the symbols for the "tm" SPURS task policy module. When TotalView detects an embedded SPURS kernel image being loaded into an SPU context, it extracts the GUIDs of the policy modules from the kernel, and searches for either the default SPU ELF image file, which is **spurs_tm.elf** or the file named by this variable.

Permitted Values:	An ELF file name
Default:	**spurs_tm.elf**

TV::checksum_libraries

Permitted Values:	
Default:	**auto**

TV::comline_patch_area_base

Allocates the patch space dynamically at the given *address*. See "*Allocating Patch Space for Compiled Expressions*" in Chapter 15 of the *TotalView Users Guide*.

Permitted Values:	A hexadecimal value indicating space accessible to TotalView
Default:	**0xffffffffffffffff**

TV::comline_patch_area_length

Sets the length of the dynamically allocated patch space to the specified *length*. See "*Allocating Patch Space for Compiled Expressions*" in Chapter 15 of the *TotalView Users Guide*.

Permitted Values:	A positive number
Default:	0

TV::command_editing

Enables some Emacs-like commands that you can use while editing text in the CLI. These editing commands are always available in the CLI window of TotalView GUI. However, they are only available within the stand-alone CLI if the terminal in which you are running it supports cursor positioning and clear-to-end-of-line. The commands that you can use are:

^A: Moves the cursor to the beginning of the line.

^B: Moves the cursor one character backward.

^D: Deletes the character to the right of cursor.

^E: Moves the cursor to the end of the line.

^F: Moves the cursor one character forward.

^K: Deletes all text to the end of line.

^N: Retrieves the next entered command (only works after **^P**).

^P: Retrieves the previously entered command.

^R or **^L**: Redraws the line.

^U: Deletes all text from the cursor to the beginning of the line.

Rubout or **Backspace**: Deletes the character to the left of the cursor.

Permitted Values:	**true** or **false**
Default:	**false**

TV::compile_expressions

When this variable is set to **true**, TotalView enables compiled expressions. If this is set to **false**, TotalView interprets your expression.

If you are running on an IBM AIX system, you can use the **–use_aix_fast_trap** command line option to speed up the performance of compiled expressions. Check the *TotalView Release Notes* to determine if your version of the operating system supports this feature.

Permitted Values:	**true** or **false**
Default:	HP Alpha and IBM AIX: **true**
	SGI IRIX: **false**
	Not settable on other platforms

TV::compiler_vars

(HP Alpha, HP, and SGI only) When this is set to **true**, TotalView shows variables created by your Fortran compiler as well as the variables in your program. When set to **false** (which is the default), TotalView does not show the variables created by your compiler.

Some Fortran compilers (HP f90/f77, SGI 7.2 compilers) write debugging information that describes variables that the compiler created to assist in some operations. For example, it could create a variable used to pass the length of **character*(*)** variables. You might want to set this variable to **true** if you are looking for a corrupted runtime descriptor.

You can override the value set to this variable in a startup file by using the following command-line options:

–compiler_vars: sets this variable to **true**

–no_compiler_vars: sets this variable to **false**

Permitted Values:	**true** or **false**
Default:	**false**

TV::control_c_quick_shutdown

When this variable is set to **true**, TotalView kills attached processes and exits. When set to false, TotalView can sometimes better manage the way it kills parallel jobs when it works with management systems. This has only been tested with SLURM. It may not work with other systems.

If you set the **TV::ignore_control_c** variable to **true**, TotalView ignores this variable.

Permitted Values:	**true** or **false**
Default:	true

TV::copyright_string

This is a read-only string containing the copyright information displayed when you start the CLI and TotalView.

TV::current_cplus_demangler

Setting this variable overrides the C++ demangler that TotalView uses. TotalView will ignore what you set the value of this variable to unless you also set the value of the **TV::force_default_cplus_demangler** variable. You can set this variable to the following values:

- **compaq:** HP cxx on running Linux-Alpha
- **dec:** HP Tru64 C++
- **gnu**: GNU C++ on Linux Alpha
- **gnu_dot**: GNU C++ Linux x86
- **gnu_v3**: GNU C++ Linux x86
- **hp**: HP aCC compiler
- **irix**: SGI IRIX C++
- **kai**: KAI C++
- **kai3_n**: KAI C++ version 3.n
- **kai_4_0**: KAI C++
- **spro**: SunPro C++ 4.0 or 5.2
- **spro5**: SunPro C++ 5.0 or later
- **sun**: Sun CFRONT C++
- **xlc**: IBM XLC/VAC++ compilers

Permitted Values:	A string naming the compiler
Default:	Derived from your platform and information within your program

TV::current_fortran_demangler

Setting this variable overrides the Fortran demangler that TotalView uses. TotalView will ignore what you set the value of this variable to unless you also set the value of the **TV::force_default_f9x_demangler** variable. You can set this variable to the following values:

- **xlf90**: IBM Fortran
- **dec**: HP Tru64 Fortran
- **decf90**: HP Tru64 Fortran 90
- **fujitsu_f9x**: Fujitsu Fortran 9x
- **hpux11_64_f9x**: HP Fortran 9x
- **intel**: Intel Fortran 9x
- **mipspro_f9x**: SGI IRIX Fortran
- **sunpro_f9x_4**: Sun ProFortran 4
- **sunpro_f9x_5**: Sun ProFortran 5

Permitted Values:	A string naming the compiler
Default:	Derived from your platform and information within your program

TV::data_format_double

Defines the format TotalView uses when displaying double-precision values. This is one of a series of variables that define how TotalView displays data. The format of each is similar and is as follows:

{*presentation format*-1 *format*-2 *format* 3}

presentation — Selects which format TotalView uses when displaying information. Note that you can display floating point information using **dec**, **hex**, and **oct** formats. You can display integers using **auto**, **dec**, and **sci** formats.

auto	Equivalent to the C language's **printf()** function's **%g** specifier. You can use this with integer and floating-point numbers. This format will either be **hexdec** or **dechex**, depending upon the programming language being used.
dec	Equivalent to the **printf()** function's **%d** specifier. You can use this with integer and floating-point numbers.
dechex	Displays information using the **dec** and **hex** formats. You can use this with integers.
hex	Equivalent to the **printf()** function's **%x** specifier. You can use this with integer and floating-point numbers.
hexdec	Displays information using the **hex** and **dec** formats. You can use this with integer numbers.
oct	Equivalent to the **printf()** function's **%o** specifier. You can use this with integer and floating-point numbers.
sci	Equivalent to the **printf()** function's **%e** specifier. You can use this with floating-point numbers.

format — For integers, *format*-1 defines the decimal format, *format*-2 defines the hexadecimal format, and *format*-3 defines the octal format.

For floating point numbers, *format*-1 defines the fixed point display format, *format*-2 defines the scientific format, and format-3 defines the auto (**printf()**'s **%g**) format.

The format string is a combination of the following specifiers:

%	A signal indicating the beginning of a format.
width	A positive integer. This is the same width specifier that is used in the **printf()** function.
. (period)	A punctuation mark separating the width from the precision.
precision	A positive integer. This is the same precision specifier that is used in the **printf()** function.

# (pound)	Tells TotalView to display a 0x prefix for hexadecimal and 0 for octal formats. This isn't used within floating-point formats.
0 (zero)	Tells TotalView to pad a value with zeros. This is ignored if the number is left-justified. If you omit this character, TotalView pads the value with spaces.
– (hyphen)	Tells TotalView to left-justify the value within the field's width.

Permitted Values: A value in the described format
Default: {auto %-1.15 %-1.15 %-20.2}

TV::data_format_ext

Defines the format TotalView uses when displaying extended floating point values such as long doubles.For a description of the contents of this variable, see TV::data_format_double.

Permitted Values: A value in the described format
Default: {auto %-1.15 %-1.15 %-1.15}

TV::data_format_int8

Defines the format TotalView uses when displaying 8-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

Permitted Values: A value in the described format
Default: {auto %1.1 %#4.2 %#4.3}

TV::data_format_int16

Defines the format TotalView uses when displaying 16-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

Permitted Values: A value in the described format
Default: {auto %1.1 %#6.4 %#7.6}

TV::data_format_int32

Defines the format TotalView uses when displaying 32-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

Permitted Values: A value in the described format
Default: {auto %1.1 %#10.8 %#12.11}

TV::data_format_int64

Defines the format TotalView uses when displaying 64-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

Permitted Values: A value in the described format
Default: {auto %1.1 %#18.16 %#23.22}

TV::data_format_int128

Defines the format TotalView uses when displaying 128-bit integer values. For a description of the contents of this variable, see TV::data_format_double.

Permitted Values: A value of the described format.
Default: {auto %1.1 %#34.32 %#44.43}

TV::data_format_long_stringlen

Defines the number of characters allowed in a long string.

Permitted Values: A positive, integer number
Default: **8000**

TV::data_format_single

Defines the format TotalView uses when displaying single precision, floating-point values. For a description of the contents of this variable, see **TV::data_format_double**.

Permitted Values: A value in the described format
Default: **{auto %-1.6 %-1.6 %-1.6}**

TV::data_format_stringlen

Defines the maximum number of characters displayed for a string.

Permitted Values: A positive integer number
Default: 100

TV::data_format_long_stringlen

Defines the maximum number of characters displayed for a long string.

Permitted Values: A positive integer number
Default: 8000

TV::dbfork

When this variable is set to **true**, TotalView catches the **fork()**, **vfork(),** and **execve()** system calls if your executable is linked with the **dbfork** library.

Permitted Values: **true** or **false**
Default: **true**

TV::default_stderr_append

When this variable is set to **true**, TotalView appends **stderr** information to the file set in the GUI or in the **TV::default_stderr_filename** variable. If no pathname is set, the value of this variable is ignored.

Permitted Values: **true** or **false**
Default: **false**

TV::default_launch_command

Names the compiled-in launch command appropriate for the platform.

Permitted Values: A string indicating the default compiled-in launch command value.
Default: SiCortex: **ssh -x**
HPUX: **remsh**
All other platforms: **rsh**

TV::default_stderr_filename

Names the file to which TotalView writes **stderr** information.

Permitted Values: A string indicating a pathname
Default: None

TV::default_stderr_is_stdout

When this variable is set to true, TotalView writes stderr information to the same location as it writes **stdout** information.

Permitted Values: **true** or **false**
Default: **false**

TV::default_stdin_filename

Names the file from which TotalView reads **stdin** information.

Permitted Values: A string indicating a pathname
Default: None

TV::default_stdout_append

When this variable is set to **true**, TotalView appends **stdout** information to the file set in the GUI or in the **TV::default_stdout_filename** variable. If no pathname is set, the value of this variable is ignored.

Permitted Values:	**true** or **false**
Default:	**false**

TV::default_stdout_filename

Names the file to which TotalView writes **stdout** information.

Permitted Values:	A string indicating a pathname
Default:	None

TV::display_assembler_symbolically

When this variable is set to **true**, TotalView displays assembler locations as **label+offset**. When it is set to **false**, these locations are displayed as hexadecimal addresses.

Permitted Values:	**true** or **false**
Default:	**false**

TV::dll_ignore_prefix

Defines a list of library files for which you are not asked to stop the process when it is loaded. This list contains a colon-separated list of prefixes. Also, TotalView will not ask if you would like to stop a process if:

- You also set the **TV::ask_on_dlopen** variable to **true**.
- The suffix of the library being loaded does *not* match a suffix contained in the **TV::dll_stop_suffix** variable.
- One or more of the prefixes in this list match the name of the library being loaded.

Permitted Values:	A list of path names, each item of which is separated from another by a colon
Default:	**/lib/:/usr/lib/:/usr/lpp/:/usr/ccs/lib/:/usr/dt/lib/:/tmp/**

TV::dll_read_all_symbols

TotalView will always read loader and debugging symbols of libraries named within this variable.

This variable is set to a colon-separated list of library names. A name can contain the * (asterisk) and ? (question mark) wildcard characters, which have their usual meaning:

- *: zero or more characters.
- ?: a single character.

As this is TotalView's default behavior, only include library names here that would be excluded because they are selected by a wildcard match within the **TV:dll_read_loader_symbols_only** and **TV::dll_read_no_symbols** variables.

Permitted Values:	One or more library names separated by colons
Default:	None

TV::dll_read_loader_symbols_only

When TotalView loads libraries named in this variable, it only reads loader symbols. Because TotalView checks and processes the names in **TV::dll_read_all_symbols** list before it processes this list, it will ignore names that are in that list and in this one.

This variable is set to a colon-separated list of strings. Any string can contain the * (asterisk) and ? (question mark) wildcard characters, which have their usual meaning:

- *: zero or more characters.
- ?: a single character.

If you are not interested in debugging most of your shared libraries, set this variable to * and then put the names of any libraries you wish to debug on the **TV::dll_read_all_symbols** list.

Permitted Values:	One or more library names separated by colons
Default:	None

TV::dll_read_no_symbols

When TotalView loads libraries named in this variable, it will not read in either loader or debugging symbols. Because TotalView checks and processes the names in the **TV::dll_read_loader_symbols_only** lists before it processes this list, it will ignore names that are in those lists and in this one.

This variable is set to a colon-separated list of strings. Any string can contain the * (asterisk) and ? (question mark) wildcard characters having their usual meaning:

- *, which means zero or more characters
- ?, which means a single character.

Because information about subroutines, variables, and file names will not be known for these libraries, stack backtraces may be truncated. However, if your program uses large shared libraries and it's time consuming to read even their loader symbols, you may want to put those libraries on this list.

Permitted Values:	One or more library names separated by colons
Default:	None

TV::dll_stop_suffix

Contains a colon-separated list of suffixes that tell TotalView that it should stop the current process when it loads a library file having this suffix.

TotalView will ask you if you would like to stop the process:

- If **TV::ask_on_dlopen** variable is set to **true**
- If one or more of the suffixes in this list match the name of the library being loaded.

Permitted Values:	A Tcl list of suffixes
Default:	None

TV::dpvm

When this is set to **true**, TotalView enables support for debugging HP Tru64 UNIX Parallel Virtual Machine applications. This value can only be set in a startup script. You can override this variable's value by using the following command-line options switches:

–dpvm sets this variable to **true**

–no_dpvm sets this variable to **false**

Permitted Values:	**true** or **false**
Default:	**false**

TV::dump_core

When this is set to **true**, TotalView will create a core file when an internal TotalView error occurs. This is only used when debugging TotalView problems. You can override this variable's value by using the following command-line options:

–dump_core sets this variable to **true**

–no_dumpcore sets this variable to **false**

Permitted Values:	**true** or **false**
Default:	**false**

TV::dwhere_ qualification_level

Controls the amount of information displayed when you use the **dwhere** command. Here are three examples:

```
dset TV::dwhere_qualification_level +overload_list
dset TV::dwhere_qualification_level -class_name
dset TV::dwhere_qualification_level -parent_function
```

You could combine these arguments into one command. For example:

```
dset TV::dwhere_qualification_level +overload_list \
      -class_name -parent_function
```

In these examples "+" means that the information should be displayed and "-" means the information should not be displayed.

The arguments to this command are as follows:

- all
- class_name
- file_directory
- hint
- image_directory
- loader_directory
- member
- module
- node
- overload_list
- parent_function
- template_args
- type_name

The **all** argument is often used as follows:

```
dset TV::dwhere_qualification_level all-parent_function
```

This is saying that all elements are displayed except for a parent function. For more information on these arguments, see "*symbol*" on page 153

Permitted Values:	One or more of the arguments listed above.
Default:	**class_name+template_args+module+ parent_function+member+node**

TV::dynamic

When this is set to **true**, TotalView loads symbols from shared libraries. This variable is available on all platforms supported by TotalView Technologies. (This may not be true for platforms ported by others. For example, this feature is not available for Hitachi computers.) Setting this value to **false** can cause the **dbfork** library to fail because TotalView might not find the **fork()**, **vfork()**, and **execve()** system calls.

Permitted Values:	**true** or **false**
Default:	**true**

TV::editor_launch_string

Defines the editor launch string command. The launch string substitution characters you can use are:

%E: The editor

%F: The display font

%N: The line number

%S: The source file

Permitted Values:	Any string value—as this is a Tcl variable, you'll need to enclose the string within **{}** (braces) if the string contains spaces
Default:	**{xterm –e %E +%N %S}**

TV::env

Names a variable that is already contained within your program's environment. This is a read-only variable. It is set by using the **–env** command-line option. For more information, see "***–env*** *variable=valiue*" on page 244.

To set this variable from within TotalView, use the **File > New Program** or **Process > Startup** dialog boxes.

Permitted Values:	None. The variable is read-only.
Default:	None

TV::follow_clone

When set to either to a value greater than 0, allows TotalView to pickup threads created using the clone() system call. The values can set are as follows:

0: TotalView does not follow clone() calls. This is most often used if problems occur.

1: TotalView follows clone() calls until the first pthread_create() call is made. This value is then set to 0.

2: TotalView follows clone() calls whenever they occur. Calls to clone() and pthread_create() can be interleaved. This may affect performance if the program has many threads.

3: (default) Like 2, TotalView follows clone() calls whenever they occur. However, TotalView uses a feature available on newer Linux systems to reduce the overhead.

Linuxthreads are not affected by this variable. This variable should be left set at 3 unless you have reason to believe it is malfunctioning on your system.

Permitted Values:	**0, 1, 2, or 3**
Default:	**3**

TV::force_default_cplus_demangler

When this is set to **true**, TotalView uses the demangler set in the **TV::current_cplus_demangler** variable. You would set this variable when TotalView uses the wrong demangler. TotalView can use the wrong demangler if you are using an unsupported compiler, and unsupported language preprocessor, or if your vendor has made changes to your compiler.

Permitted Values:	**true** or **false**
Default:	**false**

TV::force_default_f9x_demangler

When this is set to **true**, TotalView uses the demangler set in the **TV::current_fortran_demangler** variable. You would set this variable when TotalView uses the wrong demangler. TotalView can use the wrong demangler if you are using an unsupported compiler, and unsupported language preprocessor, or if your vendor has made changes to your compiler.

Permitted Values:	**true** or **false**
Default:	**false**

TV::global_typenames

When this is set to **true**, TotalView assumes that type names are globally unique within a program and that all type definitions with the same name are identical. This must be true for standard-conforming C++ compilers.

If you set this option to **true**, TotalView attempts to replace an opaque type (**struct foo *p;**) declared in one module with an identically named defined type (**struct foo { ... };**) in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, it will automatically display the variable by using the non-opaque type definition when displaying variables declared with the opaque type.

If you set this variable to **false**, TotalView does *not* assume that type names are globally unique within a program. Only use this variable if your code has different definitions of the same named type since TotalView can pick the wrong definition when it substitutes for an opaque type in this case.

Permitted Values:	**true** or **false**
Default:	**true**

TV::gnu_debuglink

When this is set to **true**, TotalView checks for a **.gnu_debuglink** section within your process. If it is found, it looks for the file named in this section. If it is set to false, TotalView ignores the contents of this section. This means that a gnu_debuglink file will not be loaded. For more information, see "*Using gnu_debuglink Files*" on page 268.

Permitted Values:	**true** or **false**
Default:	**true**

TV::gnu_debuglink_checksum

When this is set to **true**, TotalView compares the checksum of the gnu_debuglink file against the checksum contained within the **.gnu_debuglink** section. TotalView will only load the information from the gnu_debuglink file when the checksums match. For more information, see "*Using gnu_debuglink Files*" on page 268.

Permitted Values:	**true** or **false**
Default:	**true**

TV::gnu_debuglink_global_directory

Names the directory in which you store gnu_debuglink files. For more information, see "*Using gnu_debuglink Files*" on page 268.

Permitted Values:	A pathname within your file system. While this path can be relative, it is usually a full pathname.
Default:	**/usr/lib/debug**

TV::gnu_debuglink_global_search_path

Defines the search path that TotalView uses to search for **.gnu_debuglink** files. You can use two substituting variables when assigning values:

- **%D**: The directory containing the **.gnu_debuglink** file.
- **%G**: The contents of the **TV::gnu_debuglink_global_directory** variable.
- **%/**: The target directory delimiter; for example "/".

For more information, see "*Using gnu_debuglink Files*" on page 268.

Permitted Values:	A string containing directory paths.
Default:	**%D:%D.debug:%G%/%D**

TV::hpf

Deprecated.

TV::hpf_node

Deprecated.

TV::host_platform

This is a read-only value that returns the architecture upon which TotalView is running.

TV::ignore_control_c

When this is set to **true**, TotalView ignores Ctrl+C. This prevents you from inadvertently terminating the TotalView process. You would set this option to **true** when your program catches the Ctrl+C (**SIGINT**) signal. You may want to set **File > Signals** so that TotalView resends the **SIGINT** signal, instead of just stopping the program.

Permitted Values:	**true** or **false**
Default:	**false**

TV::image_load_callbacks

Contains a Tcl list of procedure names. TotalView invokes the procedures named in this list whenever it loads a new program. This could occur when:

- A user invokes a command such as **dload**.
- TotalView resolves dynamic library dependencies.
- User code uses **dlopen()** to load a new image.

TotalView invokes the functions in order, beginning at the first function in this list.

Permitted Values:	A Tcl list of procedure names
Default:	**{::TV::S2S::handle_image_load}**

TV::in_setup

Contains a **true** value if called while TotalView is being initialized. Your procedures would read the value of this variable so that code can be conditionally executed based on whether TotalView is being initialized. In most cases, this is used for code that should only be invoked while TotalView is being initialized. This is a read-only variable.

Permitted Values:	**true** or **false**
Default:	**false**

TV::ipv6_support

When you set this variable to **true**, you are enabling ipv6 support. If you set this to **false**, ipv6 support is disabled.

Permitted Values:	**true** or **false**
Default:	**false**

TV::jnibridge

Internal use only.

TV::kcc_classes

When this is set to **true**, TotalView converts structure definitions created by the KCC compiler into classes that show base classes and virtual base classes in the same way as other C++ compilers. When this is set to **false**, TotalView does not perform this conversion. In this case, TotalView displays virtual bases as pointers rather than as the data.

TotalView converts structure definitions by matching the names given to structure members. This means that TotalView may not convert definitions correctly if your structure component names look like KCC processed classes. However, TotalView never converts these definitions unless it believes that the code was compiled with KCC. (It does this when it sees one of the tag strings that KCC outputs, or when you use the KCC name demangler.) Because all of the recognized structure component names start with "_ _" and the C standard forbids this use, your code should not contain names with this prefix.

Under some circumstances, TotalView may not be able to convert the original type names because type definition are not available. For example, it may not be able to convert "**struct __SO_foo**" to "**struct foo**". In this case, TotalView shows the "**__SO_foo**" type. This is only a cosmetic problem. (The "**__SO__**" prefix denotes a type definition for the nonvirtual components of a class with virtual bases).

Since KCC output does not contain information on the accessibility of base classes (**private**, **protected**, or **public**), TotalView cannot provide this information.

Permitted Values:	**true** or **false**
Default:	**true**

TV::kernel_launch_string

This is not currently used.

TV::kill_callbacks

Names a TCL function that will be run before TotalView kills a process. The contents of this variable is a list of pairs. For example:

```
dset TV::kill_callbacks {
    {^srun$         TV::destroy_srun}
}
```

The first element in the pair is a regular expression and the second is the name of a TCL function. If the processes name matches the regular expression, TotalView runs the TCL procedure, giving it the DPID of the process as its argument. This procedure can do anything that needs to be done for orderly process termination.

If your Tcl procedure returns false, TotalView then kill your process as you would expect. If the procedure returns true, TotalView takes no further action to terminate the process.

Any slave processes are killed before the master process is killed. If there is a **kill_callback** for the master process, it is called after the slave processes are killed. If there are **kill_callbacks** for the slave processes, they will be called before the slave is killed.

Permitted Values:	List of one or more list of pairs
Default:	**{}**

TV::library_cache_directory

Specifies the directory into which TotalView writes library cache data.

Permitted Values:	A string indicating a path
Default:	**$USERNAME/.totalview/lib_cache**

TV::launch_command

Specifies the launch command.

Permitted Values:	A string indicating the launch command
Default:	The value of **TVDSVRLAUNCHCMD** if set, otherwise the value of **default_launch_command**. Note: changing the value of **TVDSVRLAUNCHCMD** in the environment after starting TotalView does not affect this variable or how **%C** is expanded.

TV::local_interface

Sets the interface name that the server uses when it makes a callback. For example, on an IBM PS2 machine, you would set this to css0. However, you can use any legal **inet** interface name. (You can obtain a list of the interfaces if you use the **netstat -i** command.)

Permitted Values:	A string
Default:	**{}**

TV::local_server

(Sun only) By default, TotalView finds the local server in the same place as the remote server. On Sun platforms, TotalView can launch a 32- and 64-bit version. This variable tells TotalView which local server it should launch.

Permitted Values:	A file or path name to the local server
Default:	**tvdsvr**

TV::local_server_launch_string

(Sun only) If TotalView will not be using the server contained in the same working directory as the TotalView executable, the contents of this string indicate the shell command that TotalView uses to launch this alternate server. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values:	A string enclosed with **{}** (braces) if it has embedded spaces
Default:	{%M –working_directory %D –local %U –set_pw %P –verbosity %V}

TV::message_queue

When this is set to **true**, TotalView displays MPI message queues when you are debugging an MPI program. When the variable is set to **false**, these queues are not displayed. You would disable these queues if something is overwriting the message queues, thereby confusing TotalView.

Permitted Values:	**true** or **false**
Default:	**true**

TV::native_platform

A read-only state variable that identifies the native (host) platform on which the TotalView client (GUI or CLI) is running. This variable's value is the same as the value of **TV::platform** except for SiCortex TotalView, where **TV::platform** is "**linux-x86-64-x-linux-mipsel64**" and **TV::native_platform** is "**linux-x86-64**."

Permitted Values:	a string indicating a platform
Default:	platform-specific

TV::nptl_threads

When set to **auto**, TotalView determines which threads package your program is using. Setting this variable to **true** tells TotalView that it is using NPTL threads. **false** means that the program isn't using this package.

Permitted Values:	**true**, **false**, or **auto**
Default:	**auto**

TV::open_cli_window_callback

The CLI executes the string that is this variable's value after you open the CLI by selecting the **Tools > Command Line** command. It is ignored when you open the CLI from the command line. It is most commonly used to set the terminal characteristics of the (pseudo) tty that the CLI is using, since these are inherited from the tty on which TotalView was started. Therefore, if you start TotalView from a shell running inside an Emacs buffer, the CLI uses the raw terminal modes that Emacs is using. You can change your terminal mode by adding the following command to your **.tvdrc** file:

```
dset TV::open_cli_window_callback "stty sane"
```

Permitted Values:	A string representing a Tcl or CLI command
Default:	Null

TV::parallel

When set to **true**, you are enabling TotalView support for parallel program runtime libraries such as MPI, PE, and UPC. You might set this to **false** if you need to debug a parallel program as if it were a single-process program.

Permitted Values:	**true** or **false**
Default:	**true**

TV::parallel_attach

Tells TotalView if it should automatically attach to processes. Your choices are as follows:

- **yes**: Attach to all started processes.
- **no**: Do not attach to any started processes.

- **ask**: Display a dialog box listing the processes to which TotalView can attach, and let the user decide to which ones TotalView should attach.

Permitted Values:	**yes**, **no**, or **ask**
Default:	**yes**

TV::parallel_configs

Defines a new parallel configuration or overwrites an existing one. You can define this variable in a global **.tvdrc** to add new configurations or overwrite those provided by TotalView Technologies. You can also define this variable in a **tvdrc** file contained within your **.totalview** directory.

For more information, please go to **http://www.totalviewtech.com/Documentation/mpi_startup.html**.

TV::parallel_stop

Tells TotalView if it should automatically run processes when your program launches them. Your choices are as follows:

- **yes**: Stop the processes before they begin executing.
- **no**: Do not interfere with the processes; that is, let them run.
- **ask**: Display a question box asking if it should stop before executing.

Permitted Values:	**yes**, **no**, or **ask**
Default:	**ask**

TV::platform

Indicates the platform upon which you are running TotalView. This is a read-only variable.

Permitted Values:	A string indicating a platform, such as **alpha** or **sun5**
Default:	Platform-specific

TV::process_load_callbacks

Names the procedures that TotalView runs immediately after it loads a program and just before it runs it. TotalView executes these procedures after it invokes the procedures in the **TV::image_load_callbacks** list.

The procedures in this list are only called once even though your executable may use many programs and libraries.

Permitted Values:	A list of procedures
Default:	**TV::source_process_startup**. The default procedure looks for a file with the same name as the newly loaded process's executable image that has a **.tvd** suffix appended to it. If it exists, TotalView executes the commands contained within it. This function is passed an argument that is the ID for the newly created process.

TV::pvm

When this is set to **true**, TotalView lets you debug the ORNL (Oak Ridge National Laboratory) implementation of Parallel Virtual Machine (PVM) applications. This variable can only be set in a start up script. However, you can override this value by using the following command-line options:

–pvm sets this variable to **true**

–no_pvm sets this variable to **false**

Permitted Values:	**true** or **false**
Default:	**false**

TV::replay_history_mode

Controls how ReplayEngine handles the history buffer when it is full, as follows:

- 1: Discard the oldest history and continue.
- 2: Stop the process.

Permitted Values:	1 or 2
Default:	1

TV::replay_history_size

Specifies the size of ReplayEngine's buffer for recorded history, in either bytes, kilobytes (K) or megabytes (M). To specify kilobytes or megabytes, append a K or M to the number, as follows: 10000K or 1024M

Permitted Values:	An integer or an integer followed by K or M
Default:	0 (Limited only by available memory)

TV::restart_threshold

When you kill a multi-threaded or multiprocess program, TotalView asks if you really want to kill the program if there is more than one thread or process running. Changing this value tells TotalView how many threads or processes need to be running before it asks.

Permitted Values:	a positive integer
Default:	1

TV::save_global_dialog_defaults

Obsolete.

TV::save_search_path

Obsolete.

TV::save_window_pipe_or_filename

Names the file to which TotalView writes or pipes the contents of the current window or pane when you select the **File > Save Pane** command.

Permitted Values:	A string naming a file or pipe
Default:	None, until something is saved. Afterward, the saved string is the default.

TV::search_case_sensitive

When this is set to **true**, text searches only succeed if a string exists whose case exactly matches what you enter in the **Edit > Find** Dialog Box. For example, searching for **Foo** won't find **foo** if this variable is set to **true**. It will be found if this variable is set to **false**.

Permitted Values:	**true** or **false**
Default:	**false**

TV::server_launch_enabled

When this is set to **true**, TotalView uses its single-process server launch procedure when launching remote **tvdsvr** processes. When the variable is set to **false**, **tvdsvr** is not automatically launched.

Permitted Values:	**true** or **false**
Default:	**true**

TV::server_launch_string

Names the command string that TotalView uses to automatically launch the TotalView Debugger Server (**tvdsvr**) when you start to debug a remote process. This command string is executed by **/bin/sh**. By default, TotalView uses the **rsh** command to start the server, but you can use any other command that can invoke **tvdsvr** on a remote host. If no command is available for invoking a remote process, you can't automatically launch the server; therefore, you should set this variable to **/bin/false**. If you cannot automatically launch a server, you should also set the **TV::server_launch_enabled** variable to **false**. For information on this launch string, see "*Replacement Characters*" on page 257.

Permitted Values:	A string
Default:	**{%C %R –n "%B/tvdsvr –working_directory %D –callback %L –set_pw %P –verbosity %V %F"}**

TV::server_launch_timeout

Specifies the number of seconds that TotalView waits to hear back from the TotalView Debugger Server (**tvdsvr**) that it launches.

Permitted Values:	An integer from 1 to 3600 (1 hour)
Default:	30

TV::server_response_wait_timeout

Specifies how long TotalView will wait for a response from the TotalView Debugger Server (**tvdsvr**). Using a higher value may help avoid server timeouts if you are debugging across multiple nodes that are heavily loaded.

Permitted Values:	An integer from 1 to 3600 (1 hour)
Default:	30

TV::share_action_point

Indicates the scope in which TotalView places newly created action points. In the CLI, this is the **dbarrier**, **dbreak**, and **dwatch** commands. If this Boolean value is **true**, newly created action point are shared across the group. If it is **false**, a newly created action point is only active in the process in which it is set.

As an alternative to setting this variable, you can select the **Plant in share group** check box in the **Action Points** Page in the **File > Preferences** Dialog Box. You can override this value in the GUI by using selecting the **Plant in share group** checkbox in the **Action Point > Properties** Dialog Box.

Permitted Values:	**true** or **false**
Default:	**true**

TV::signal_handling_mode

The list that you assign to this variable modifies the way in which TotalView handles signals. This list consists of a list of *signal_action* descriptions, separated by spaces:

```
signal_action[signal_action] ...
```

A *signal_action* description consists of an action, an equal sign (=), and a list of signals:

```
action=signal_list
```

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**.

A *signal_list* is a list of one or more signal specifiers, separated by commas:

```
signal_specifier[,signal_specifier] ...
```

A *signal_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as **11**), or a star (*****), which specifies all signals. We recommend using the signal name rather than the number because number assignments vary across UNIX versions.

The following rules apply when you are specifying an *action_list*:

- If you specify an action for a signal in an *action_list*, TotalView changes the default action for that signal.
- If you do not specify a signal in the *action_list*, TotalView does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, TotalView ignores it.
- If you specify an action for a signal twice, TotalView uses the last action specified. In other words, TotalView applies the actions from left to right.

If you need to revert the settings for signal handling to built-in defaults, use the **Defaults** button in the **File > Signals** Dialog Box.

For example, to set the default action for the **SIGTERM** signal to *Resend*, you specify the following action list:

```
{Resend=SIGTERM}
```

As another example, to set the action for **SIGSEGV** and **SIGBUS** to *Error*, the action for **SIGHUP** and **SIGTERM** to *Resend*, and all remaining signals to *Stop*, you specify the following action list:

```
{Stop=* Error=SIGSEGV,SIGBUS Resend=SIGHUP,SIGTERM}
```

This action list shows how TotalView applies the actions from left to right.

1. Sets the action for all signals to *Stop*.
2. Changes the action for **SIGSEGV** and **SIGBUS** from *Stop* to *Error*.
3. Changes the action for **SIGHUP** and **SIGTERM** from *Stop* to *Resend*.

Permitted Values:	A list of signals, as was just described
Default:	This differs from platform to platform; type **dset TV::signal_handling_mode** to see what a platform's default values are

TV::source_pane_tab_width

Sets the width of the tab character that is displayed in the Process Window's Source Pane. You may want to set this value to the same value as you use in your text editor.

Permitted Values:	An integer
Default:	8

TV::spell_correction

When you use the **View > Lookup Function** or **View > Lookup Variable** commands in the Process Window or edit a type string in a Variable Window, TotalView checks the spelling of your entries. By default (**verbose**), TotalView displays a dialog box before it corrects spelling. You can set this resource to **brief** to run the spelling corrector silently. (TotalView makes the spelling correction without displaying it in a dialog box first.) You can also set this resource to **none** to disable the spelling corrector.

Permitted Values:	**verbose**, **brief**, or **none**
Default:	**verbose**

TV::stack_trace_qualification_level

Controls the amount of information displayed in stack traces. For more information, see **TV::dwhere_qualification_level**.

Permitted Values:	One or more of the following arguments: **all**, **class_name**, **file_directory**, **hint**, **image_directory**, **loader_directory**, **member**, **module**, **node**, **overload_list**, **parent_function**, **template_args**, **type_name**.
Default:	**class_name+template_args+module+parent_function+member+node**

TV::stop_all

Indicates a default property for newly created action points. This property tells TotalView what else it should stop when it encounters this action point. The values you can set are as follows:

group	Stops the entire control group when the action point is hit.
process	Stops the entire process when the action point is hit.
thread	Only stops the thread that hit the action point. Note that **none** is a synonym for **thread**.

Permitted Values:	**group**, **process**, or **thread**
Default:	**group**

TV::stop_relatives_on_proc_error

When this is set to **true**, TotalView stops the control group when an error signal is raised. This is the variable used by the **Stop control group on error signal** option in the **Options** Page of the **File > Preferences** Dialog Box.

Permitted Values:	**true** or **false**
Default:	**true**

TV::suffixes

Use a space separated list of items to identify the contents of a file. Each item on this list has the form: **suffix:lang[:include]**. You can set more than suffix for an item. If you want to remove an item from the default list, set its value to **unknown**.

Permitted Values:	A list identifying how suffixes are used
Default:	**{:c:include s:asm S:asm c:c h:c:include lex:c:include y:c:include bmap:c:include f:f77 F:f77 f90:f9x F90:f9x hpf:hpf HPF:hpf cxx:c++ cpp:c++ cc:c++ c++:c++ C:c++ C++:c++ hxx:c++:include hpp:c++:include hh:c++:include h++:c++:include HXX:c++:include HPP:c++:include HH:c++:include H:c++:include ih:c++:include th:c++}**

TV::target_platform

A read-only variable that displays a list of the platforms on which you can debug from the native (host) platform, usually in the format *os-cpu*. For example, from a native platform of Linux-x86, the list is "**linux-power linux-cell linux-x86_64 linux-x86 catamount-x86_64 catamount-x86**." The platform names may be listed differently than in **TV::platform and TV::native_platform**. For example, for AIX, **TV::target_platform** is "**aix-power**" but **TV::platform** and **TV::native_platform** are "**rs6000**."

Permitted Values:	A list of platform names
Default:	Platform-dependent

TV:ttf When set to **true**, TotalView uses registered type transformations to change the appearance of data types that have been registered using the **TV::type_transformation** command.

Permitted Values:	**true** or **false**
Default:	**true**

TV::ttf_max_length When transforming STL structures, TotalView must chase through pointers to obtain values. This number indicates how many of these pointers it should follow.

Permitted Values:	an integer number
Default:	10000

TV::use_fast_trap Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional breakpoints, also known as EVAL points. As of TotalView 8.7, when this variable was introduced, only AIX supported the fast trap mechanism for breakpoints, but we anticipate other operating systems adding support. You cannot interactively use this variable. Instead, you must set it within a TotalView startup file; for example, set its value with a **.tvdrc** file.

Your operating system may not be configured correctly to support this option. See the *TotalView Release Notes* on our web site for more information.

Permitted Values:	**true** or **false**
Default:	**true**

TV::use_fast_wp Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional watchpoints, also known as CDWP points. As of TotalView 8.7, when this variable was introduced, only AIX supported the fast trap mechanism for watchpoints, but we anticipate other operating systems adding support. You cannot interactively use this variable. Instead, you must set it within a TotalView startup file; for example, set its value with a **.tvdrc** file.

Your operating system may not be configured correctly to support this option. See the *TotalView Release Notes* on our web site for more information.

Permitted Values:	**true** or **false**
Default:	**false**

TV::use_interface This variable is a synonym for **TV::local_interface**.

TV::user_threads

When this is set to **true**, it enables TotalView support for handling user-level (M:N) thread packages on systems that support two-level (kernel and user) thread scheduling.

Permitted Values:	**true** or **false**
Default:	**true**

TV::version

Indicates the current TotalView version. This is a read-only variable.

Permitted Values:	A string
Default:	Varies from release to release

TV::visualizer_launch_enabled

When this is set to **true**, TotalView automatically launches the Visualizer when you first visualize something. If you set this variable to **false**, TotalView disables visualization. This is most often used to stop evaluation points containing a **$visualize** directive from invoking the Visualizer.

Permitted Values:	**true** or **false**
Default:	**true**

TV::visualizer_launch_string

Specifies the command string that TotalView uses when it launches a visualizer. Because the text is actually used as a shell command, you can use a shell redirection command to write visualization datasets to a file (for example, "**cat** > *your_file*").

Permitted Values:	A string
Default:	**%B/visualize**

TV::visualizer_max_rank

Specifies the default value used in the **Maximum permissible rank** field in the **Launch Strings** Page of the **File > Preferences** Dialog Box. This field sets the maximum rank of the array that TotalView will export to a visualizer. The Visualizer cannot visualize arrays of rank greater than 2. If you are using another visualizer or just dumping binary data, you can set this value to a larger number.

Permitted Values:	An integer
Default:	2

TV::warn_step_throw

If this is set to **true** and your program throws an exception during a single-step operation, TotalView asks if you wish to stop the step operation. The process will be left stopped at the C++ run-time library's "throw" routine. If this is set to **false**, TotalView will not catch C++ exception throws during single-step operations. Setting it to **false** may mean that TotalView will lose control of the process, and you may not be able to control the program.

Permitted Values:	**true** or **false**
Default:	**true**

TV::wrap_on_search

When this is set to **true**, TotalView will continue searching from either the beginning (if **Down** is also selected in the **Edit > Find** Dialog Box) or the end (if **Up** is also selected) if it doesn't find what you're looking for. For example, you search for **foo** and select the **Down** button. If TotalView doesn't find it in the text between the current position and the end of the file, TotalView will continue searching from the beginning of the file if you set this option.

Permitted Values:	**true** or **false**
Default:	**true**

TV::xterm_name

The name of the program that TotalView should use when spawning the CLI. In most cases, you will set this using the **–xterm_name** command-line option.

Permitted Values:	a string
Default:	**xterm**

TV::MEMDEBUG:: Namespace

TV::MEMDEBUG:: default_snippet_ extent

Defines the number of code lines above and below point of allocation that the Memory Debugger saves when it is adding code snippets to saved output.

You can also set this value using a Memory Debugger preference.

Permitted Values:	A positive integer number
Default:	5

TV::MEMDEBUG:: do_not_apply_hia_ defaults

If set to **true**, tells the Memory Debugger that it should use settings it finds in a default **.hiarc** file. Otherwise, the Memory Debuggers sets all options to off.

You can also set this value using a Memory Debugger preference.

Permitted Values:	**true** or **false**
Default:	**false**

TV::MEMDEBUG:: hia_allow_ibm_poe

Tells the Memory Debugger if you can enable memory debugging on poe. As the default value is **false**, set this variable if you want memory debugging to be on by default. This variable is hardly ever used.

Permitted Values:	**true** or **false**
Default:	**false**

TV::MEMDEBUG:: ignore_snippets

When set to **true**, the Memory Debugger ignores code snippets that it saved and instead locates the information from your program's files.

You can also set this value using Memory Debugger preference.

Permitted Values:	**true** or **false**
Default:	**false**

TV::MEMDEBUG:: leak_check_interior_ pointers

When this is set to **true**, the Memory Debugger considers a block as being referenced if a pointer is pointing anywhere within the block instead of just at the block's starting location. In most programs, the code should be keeping track of the block's boundary. However, if your C++ program is using multiple inheritance, you may be pointing into the middle of the block without knowing it.

Permitted Values:	**true** or **false**
Default:	**true**

TV::MEMDEBUG:: leak_max_cache

Sets the size of the Memory Debugger's cache. We urge you not to change this value unless your program is exceptionally large or are asked to make the change by someone on the TotalView support team.

Permitted Values:	A positive integer number
Default:	4194304

TV::MEMDEBUG:: leak_max_chunk

Tells the Memory Debugger how much memory it should obtain when it obtains memory from your operating system. You shouldn't change this value unless asked to by someone on the TotalView support team.

Permitted Values:	A positive integer number
Default:	4194304

TV::MEMDEBUG:: shared_data_filters

Names a filter definition file that is not located in the default directory. (The default directory is the **lib** subdirectory within the TotalView installation directory.) The contents of this variable are read when TotalView begins executing. Consequently, TotalView ignores any changes you make during the debugging session. The following example names the directory in which the filter file resides. This example assumes that filter has the default name, which is **tv_filters.tvd**.

```
dset TV::MEMDEBUG::shared_data_filters {/home/projects/filters/}
```

Use brackets so that Tcl doesn't interpret the "/" as a mathematical operator. If you wish to use a specific file, just use its name in this command. For example:

```
dset TV::MEMDEBUG::shared_data_filters \
    {/home/projects/filters/filter.tvd}
```

The file must have a **.tvd** extension.

Permitted Values:	A string naming the path to the filter directory.
Default:	none

TV::GUI:: Namespace

The variables in this section only have meaning (and in some cases, a value) when you are using the TotalView GUI.

TV::GUI::chase_mouse

When this variable is set to **true**, TotalView displays dialog boxes at the location of the mouse cursor. If this is set to **false**, TotalView displays them centered in the upper third of the screen.

Permitted Values:	**true** or **false**
Default:	**true**

TV::GUI::display_bytes_kb_mb

When this is set to **true**, the Memory Debugger displays memory block sizes in megabytes. If set to **false**, it displays memory blocks sizes in kilobytes.

Permitted Values:	**true** or **false**
Default:	**true**

TV::GUI::display_font_dpi

Indicates the video monitor DPI (dots per inch) at which fonts are displayed.

Permitted Values:	An integer
Default:	75

TV::GUI::enabled

When this is set to **true**, you invoked the CLI from the GUI or a startup script. Otherwise, this read-only value is **false**.

Permitted Values:	**true** or **false**
Default:	**true** if you are running the GUI even though you are seeing this in a CLI window; **false** if you are only running the CLI

TV::GUI::fixed_font

Indicates the specific font TotalView uses when displaying program information such as source code in the Process Window or data in the Variable Window. This variable contains the value set when you select a **Code and Data Font** entry in the **Fonts** Page of the **File** > **Preferences** Dialog Box.

This is a read-only variable.

Permitted Values:	A string naming a fixed font residing on your system
Default:	While this is platform specific, here is a representative value: **-adobe-courier-medium-r-normal--12-120-75-75-m-70-iso8859-1**

TV::GUI::fixed_font_family

Indicates the specific font TotalView uses when displaying program information such as source code in the Process Window or data in the Variable Window. This variable contains the value set when you select a **Code and Data Font** entry of the **Fonts** Page of the **File** > **Preferences** Dialog Box.

Permitted Values:	A string representing an installed font family
Default:	**fixed**

TV::GUI::fixed_font_size

Indicates the point size at which TotalView displays fixed font text. This is only useful if you have set a fixed font family because if you set a fixed font, the value entered contains the point size.

Font sizes are indicated using printer points.

Permitted Values:	An integer
Default:	12

TV::GUI::font

Indicates the specific font used when TotalView writes information as the text in dialog boxes and in menu bars. This variable contains the information set when you select a **Select by full name** entry in the **Fonts** Page of the **File** > **Preferences** Dialog Box.

Permitted Values:	A string naming a fixed font residing on your system. While this is platform specific, here is a representative value: **-adobe-helvetica-medium-r-normal--12-120-75-75-p-67-iso8859-1**
Default:	**helvetica**

TV::GUI::force_window_positions

Setting this variable to **true** tells TotalView that it should use the version 4 window layout algorithm. This algorithm tells the window manager where to set the window. It also cascades windows from a base location for each window type. If this is not set, which is the default, newer window managers such as **kwm** or **Enlightenment** can use their smart placement modes.

Dialog boxes still chase the pointer as needed and are unaffected by this setting.

Permitted Values:	**true** or **false**
Default:	**false**

TV::GUI::frame_offset_x

Not implemented.

TV::GUI::frame_offset_y

Not implemented.

TV::GUI::geometry_call_tree

Specifies the position at which TotalView displays the **Tools** > **Call Tree** Window. This position is set using a list containing four values: the window's **x** and **y** coordinates. These are followed by two more values specifying the window's width and height.

If you set any of these values to 0 (zero), TotalView uses its default value. This means, however, you cannot tell TotalView to place a window at **x**, **y** coordinates of 0, 0. Instead, you'll need to place the window at 1, 1.

If you specify negative **x** and **y** coordinates, TotalView aligns the window to the opposite edge of the screen.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	**{0 0 0 0}**

TV::GUI::geometry_cli

Specifies the position at which TotalView displays the **Tools** > **CLI** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	**{0 0 0 0}**

TV::GUI::geometry_expressions

Specifies the position at which TotalView displays the **Tools** > **Expression List** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_globals

Specifies the position at which TotalView displays the **Tools > Program Browser** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_help

Specifies the position at which TotalView displays the **Help** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_memory_stats

Specifies the position at which TotalView displays the **Tools > Memory Statistics** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinate's and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_message_queue

Specifies the position at which TotalView displays the **Tools > Message Queue** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_message_queue_graph

Specifies the position at which TotalView displays the **Tools > Message Queue Graph** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_modules

Specifies the position at which TotalView displays the **Tools > Fortran Modules** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height.
Default:	{0 0 0 0}

TV::GUI::geometry_process

Specifies the position at which TotalView displays the Process Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_ptset

No longer used.

TV::GUI::geometry_pvm

Specifies the position at which TotalView displays the **Tools > PVM** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_root

Specifies the position at which TotalView displays the Root Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_thread_objects

Specifies the position at which TotalView displays the **Tools > Thread Objects** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_variable

Specifies the position at which TotalView displays the Variable Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::geometry_variable_stats

Specifies the position at which TotalView displays the **Tools > Statistics** Window.

See **TV::GUI::geometry_call_tree** for information on setting this list.

Permitted Values:	A list containing four integers indicating the window's **x** and **y** coordinates and the window's width and height
Default:	{0 0 0 0}

TV::GUI::heap_summary_refresh

Not user settable.

TV::GUI::inverse_video

Not implemented.

TV::GUI::keep_expressions	Deprecated.
TV::GUI::keep_search_dialog	When this is set to **true**, TotalView doesn't remove the **Edit > Find** Dialog Box after you select that dialog box's **Find** button. If you select this option, you will need to select the **Close** button to dismiss the **Edit > Find** box. *Permitted Values*: **true** or **false** *Default*: **true**
TV::GUI::pop_at_breakpoint	When this is set to **true**, TotalView sets the **Open (or raise) process window at breakpoint** check box to be selected by default. If this variable is set to **false**, it sets that check box to be deselected by default. *Permitted Values*: **true** or **false** *Default*: **false**
TV::GUI::pop_on_error	When this is set to **true**, TotalView sets the **Open process window on error signal** check box in the **File > Preferences**'s **Option** Page to be selected by default. If you set this to **false**, TotalView sets that check box to be deselected by default. *Permitted Values*: **true** or **false** *Default*: **true**
TV::GUI::show_startup_parameters	Setting this value to true tells TotalView to display that it should display the Process > Startup dialog box when you use a program name as an argument to the totalview command. *Permitted Values*: **true** or **false** *Default*: **true**
TV:GUI:show_sys_thread_id	Setting this value to true tells TotalView to display the current thread's system thread ID within the TotalView GUI. *Permitted Values*: **true** or **false** *Default*: **true**
TV::GUI::single_click_dive_enabled	When set, you can perform dive operations using the middle mouse button. Diving using a left-double-click still works. If you are editing a field, clicking the middle mouse performs a paste operation. *Permitted Values*: **true** or **false** *Default*: **true**
TV::GUI::toolbar_style	The value set for this variable tell TotalView how it should display the toolbar. *Permitted Values*: **icons_above_text**, **icons_besides_text**, **icons**, or **text** *Default*: **icons_above_text**
TV::GUI::tooltips_enabled	When set tp true, TotalView displays more information about variables in the Process Window Source Pane. *Permitted Values*: **true** or **false** *Default*: **true**

TV::GUI::ui_font Indicates the specific font used when TotalView writes information as the text in dialog boxes and in menu bars. This variable contains the information set when you select a **Select by full name** entry in the **Fonts** Page of the **File > Preferences** Dialog Box.

Permitted Values:	While this is platform specific, here is a representative value: **-adobe-helvetica-medium-r-normal--12-120-75-75-p-67-iso8859-1**
Default:	**helvetica**

TV::GUI::ui_font_family Indicates the family of fonts that TotalView uses when displaying such information as the text in dialog boxes and menu bars. This variable contains the information set when you select a **Family** in the **Fonts** Page of the **File > Preferences** Dialog Box.

Permitted Values:	A string
Default:	**helvetica**

TV::GUI::ui_font_size Indicates the point size at which TotalView writes the font used for displaying such information as the text in dialog boxes and menu bars. This variable contains the information set when you select a User Interface **Size** in the **Fonts** Page of the **File > Preferences** Dialog Box.

Permitted Values:	An integer
Default:	**12**

TV::GUI::using_color Not implemented.

TV::GUI::using_text_color Not implemented.

TV::GUI::using_title_color Not implemented.

TV::GUI::version This number indicates which version of the TotalView GUI is being displayed. This is a read-only variable.

Permitted Values:	A number

Creating Type Transformations

chapter

6

The Type Transformation Facility (TTF) lets you define the way TotalView displays aggregate data. *Aggregate data* is simply a collection of data elements from within one class or structure. These elements can also be other aggregated elements. In most cases, you will create transformations that model data that your program stores in an array- or list-like way. You can also transform arrays of structures.

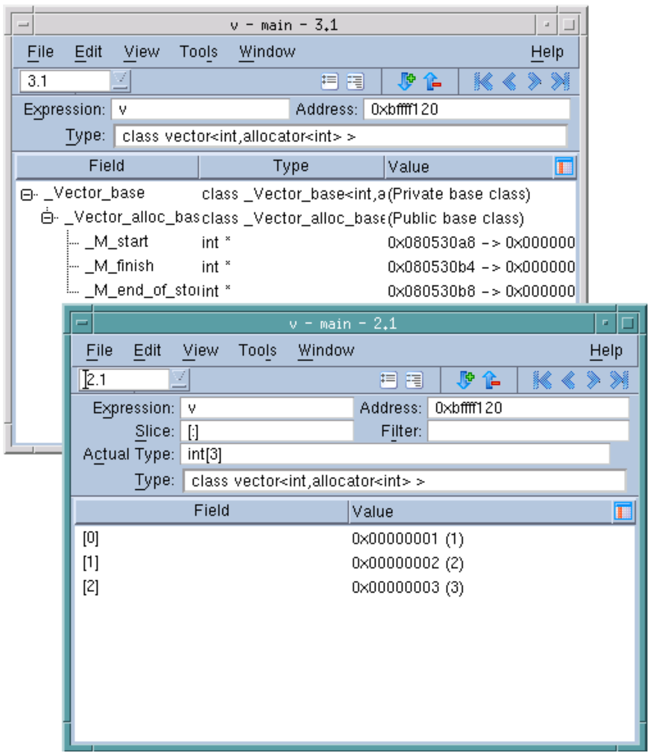
This chapter describes the TTF. It presents information how you create your own. Creating transformations can be quite complicated. This chapter looks at transformations for which TotalView can automatically create an addressing expression. If TotalView can't create this expression, you will need to read the *Creating Type Transformations* Guide, which is located on our web site at **http://www.totalviewtech.com/Documentation/**.

Topics in this chapter are:

Why Type Transformations

Modern programming languages allow you to use abstractions such as structures, class, and STL data types such as lists, maps, and vectors to model the data that your program uses. For example, the STL (Standard Template Library) allows you to create vectors of the data contained within a class. These abstractions simplify the way in which you think of and manipulate program's data. These abstractions can also complicate the way in which you debug your program because it may be nearly impossible or very inconvenient to examine your program's data. For example, the following figure shows a vector transformation.

Figure 4: A Vector Transformation


v - main - 3.1
File Edit View Tools Window Help
3.1

```
Expression: v                              Address: 0xbffff120
      Type: class vector<int,allocator<int> >

Field                 Type                        Value
⊟ _Vector_base        class _Vector_base<int,a    (Private base class)
  ⊟ _Vector_alloc_bas class _Vector_alloc_base    (Public base class)
      _M_start        int *                       0x080530a8 -> 0x000000
      _M_finish       int *                       0x080530b4 -> 0x000000
      _M_end_of_stor  int *                       0x080530b8 -> 0x000000
```

v - main - 2.1
File Edit View Tools Window Help
2.1

```
 Expression: v                Address: 0xbffff120
      Slice: [:]               Filter:
Actual Type: int[3]
       Type: class vector<int,allocator<int> >

Field                         Value
[0]                           0x00000001 (1)
[1]                           0x00000002 (2)
[2]                           0x00000003 (3)
```

The upper left window shows untransformed information. In this example, TotalView displays the complete structure of this GNU C++ STL structure. This means that you are seeing the data exactly as your compiler created it.

The logical model that is the reason for using an STL vector is buried within this information. Neither TotalView nor your compiler has this information. This is where type transformations come in. They give TotalView knowledge of how the data is structured and how it can access data elements. The bottom Variable Window shows how TotalView reorganizes this information.

Transforming STL *strings, vectors, lists, and maps is the TotalView default. If you do not want TotalView to transform your information, select the Options Tab within the* File > Preferences *Dialog Box and then remove the check mark from* View simplified STL containers (and user-defined transformations).

Creating Structure and Class Transformations

The procedure for transforming a structure or a class requires that create a mapping between the elements of the structure or class and the way in which you want this information to appear.

This section contains the following topics:

Transforming Structures

The following small program contains a structure and the statements necessary to initialize it:

```
#include <stdio.h>

int main () {
   struct stuff {
      int month;
      int day;
      int year;
      char * pName;
      char * pStreet;
      char CityState[30];
   };

   struct stuff info;
   char my_name[]   = “John Smith”;
   char my_street[] = “24 Prime Parkway, Suite 106”;
   char my_CityState[] = “Natick, MA 01760”;

   info.month = 6;
   info.day   = 20;
   info.year  = 2004;
   info.pName = my_name;
   info.pStreet  = my_street;
   strcpy(info.CityState, my_CityState);

   printf(“The year is %d\n”, info.year);
}
```

Suppose that you do not want to see the **month** and **day** components. You can do this by creating a transformation that names just the elements you want to include:

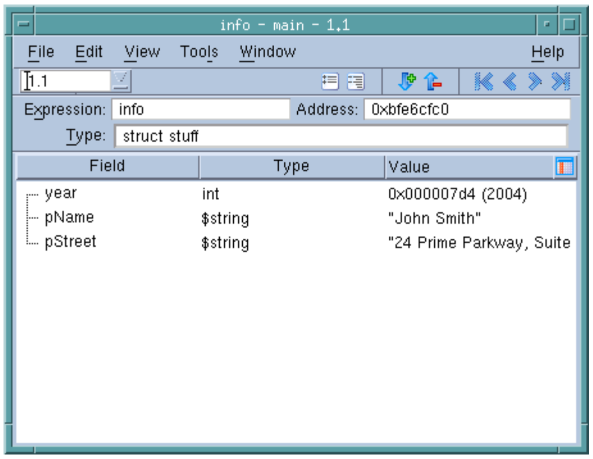
```
::TV::TTF::RTF::build_struct_transform {
   name    {^struct stuff$}
   members {
         { year     { year      } }
         { pName    { * pName   } }
         { pStreet  { * pStreet } }
   }
}
```

You can apply this transformation to your data in the following ways:

- After opening the program, use the **Tools > Command Line** command to open a CLI Window. Next, type this function call.
- If you write the function call into a file, use the Tcl **source** command. If the name of the file is **stuff.tvd**, enter the following command into a CLI Window:

  source stuff.tvd
- You can place the transformation source file into the same directory directory as the executable, giving it the same root name as the executable. If the executable file has the name **stuff**, TotalView will automatically execute all commands within a file named **stuff.tvd** when it loads your executable.

After TotalView processes your transformation, it displays the following Variable Window when you dive on the **info** structure:



Expression: info Address: 0xbfe6cfc0
Type: struct stuff

```
Field      Type      Value
year       int       0x000007d4 (2004)
pName      $string   "John Smith"
pStreet    $string   "24 Prime Parkway, Suite
```

Figure 5: Transforming a Structure

build_struct_transform Function

The **build_struct_transform** routine used in the example in the previous section is a Tcl helper function that builds the callbacks and addressing expressions that TotalView needs when it transforms data. It has two required arguments: **name** and **members**.

name Argument

The **name** argument contains a regular expression that identifies the structure or class. In this example, **struct** is part of the identifier's name. It does not mean that you are creating a structure. In contrast, if **stuff** is class, you would type:

```
name  {^class stuff$}
```

If you use a wildcard such as asterisk ()* or question mark (?), TotalView can match more than one thing. In some cases, this is what you want. If it isn't, you need to be more precise in your wildcard.

members Argument

The **members** argument names the elements that TotalView will include in the information it will display. This argument contains one or more lists. The example in the previous section contained three lists: **year**, **pName**, and **pStreet**. Here again is the **pName** list:

```
{ pName   { * pName   } }
```

The first element in the list is the display name. In most cases, this is the name that exists in the structure or class. However, you can use another name. For example, since the transformation dereferences the pointer, you might want to change its name to **Name**:

```
{ Name    { * pName   } }
```

The sublist within the list defines a type transformation expression. These expressions are discussed in the next section.

Type Transformation Expressions

The list that defines a member has a name component and sublist within the list. This sublist defines a *type transformation expression*. This expression tells TotalView what it needs to know to locate the member. The example in the previous section used two of the six possible expressions. The following list describes these expressions:

{member}	No transformation occurs. The structure or class member that TotalView displays is the same as it displays if you hadn't used a transformation. This is most often used for simple data types such as ints and floats.
{* expr}	Dereferences a pointer. If the data element is a pointer to an element, this expression tells TotalView to dereference the pointer and display the dereferenced information.
{expr . expr}	Names a subelement of a structure. This is used in the same way as the dot operator that exists in C and C++. You must type a space before and after the dot operator.
{expr -> expr }	Names a subelement in a structure accessed using a pointer. This is used in the same way as the **->** operator that exists in C and C++. You must type a space before and after the **->** operator.

{datatype cast expr}	
	Casts a data type. For example:
	{double cast national_debt}
{N upcast expr}	Converts the current class type into one of its base classes. For example:
	{base_class upcast expr }

You can nest expressions within expressions. For example, here is the list for adding an int member that is defined as **int **pfoo**:

{foo { * {* pfoo}}

Example The example in this section changes the structure elements of the example in the previous section so that they are now class members. In addition, this example contains a class that is derived from a second class:

```
#include <stdio.h>
#include <string.h>

class xbase
{
   public:
      char * pName;
      char * pStreet;
      char CityState[30];
};

class x1 : public xbase
{
   public:
      int month;
      int day;
      int year;
      void *v;
      void *q;
};

class x2
{
   public:
      int q1;
      int q2;
};

int main () {
   class x1 info;
   char my_name[]   = “John Smith”;
   char my_street[] = “24 Prime Parkway, Suite 106”;
   char my_CityState[] = “Natick, MA 01760”;

   info.month = 6;
   info.day   = 20;
   info.year  = 2004;
   info.pName = my_name;
   info.pStreet  = my_street;
```

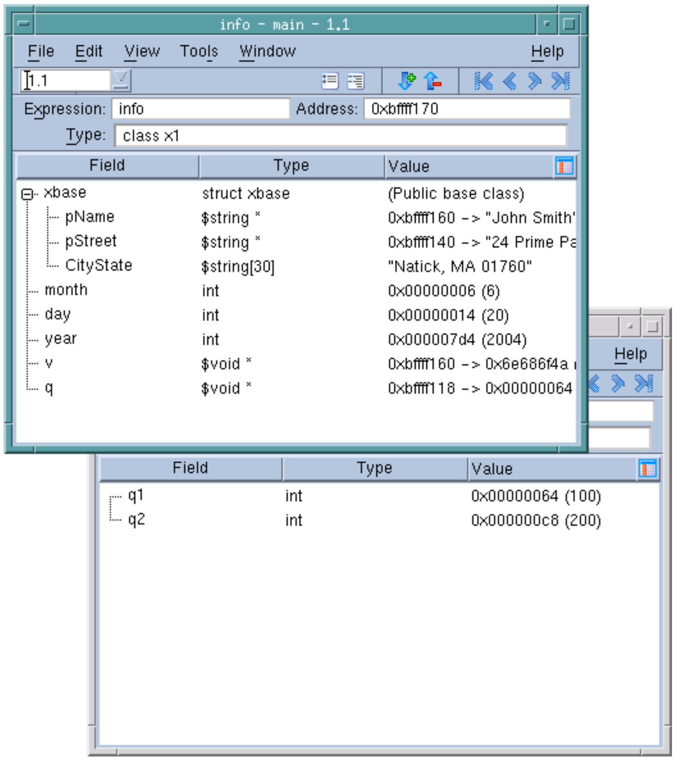
```
    info.v  = (void *) my_name;
    strcpy(info.CityState, my_CityState);

    class x2 x;
    x.q1 = 100;
    x.q2 = 200;
    info.q = (void *) &x;

    printf("The year is %d\n", info.year);
}
```

Figure 6 on page 235 shows the Variables Windows that TotalView displays for the **info** class and the **x** struct.

Figure 6: Untransformed Data



Expression: info Address: 0xbffff170

Type: class x1

Field	Type	Value
xbase	struct xbase	(Public base class)
pName	$string *	0xbffff160 -> "John Smith'
pStreet	$string *	0xbffff140 -> "24 Prime Pa
CityState	$string[30]	"Natick, MA 01760"
month	int	0x00000006 (6)
day	int	0x00000014 (20)
year	int	0x000007d4 (2004)
v	$void *	0xbffff160 -> 0x6e686f4a
q	$void *	0xbffff118 -> 0x00000064

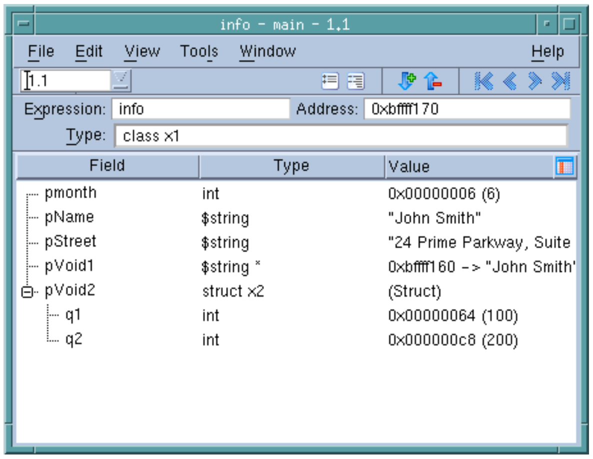
Field	Type	Value
q1	int	0x00000064 (100)
q2	int	0x000000c8 (200)

The following transformation remaps this information:

```
::TV::TTF::RTF::build_struct_transform {
   name    {^(class|struct) x1$}
   members {
      { pmonth    { month } }
      { pName     { xbase upcast { * pName    } } }
      { pStreet   { xbase upcast { * pStreet } } }
      { pVoid1    { "$string *"  cast v      } }
      { pVoid2    { * { "class x2 *"  cast q } } }
   }
}
```

After you remap the information, TotalView displays the **x1** class as shown in Figure 7.

Figure 7: Transformed Class



Expression: info Address: 0xbffff170
Type: class x1

Field	Type	Value
pmonth	int	0x00000006 (6)
pName	$string	"John Smith"
pStreet	$string	"24 Prime Parkway, Suite
pVoid1	$string *	0xbffff160 -> "John Smith'
pVoid2	struct x2	(Struct)
q1	int	0x00000064 (100)
q2	int	0x000000c8 (200)

The members of this transformation are as follows:

- **pmonth**: The **month** member is added to the transformed structure without making any changes to the way TotalView displays its data. This member, however, changes the display name of the data element. That is, the name that TotalView uses to display a member within the remapped structure does not have to be the same as it is in the actual structure.
- **pName**: The **pName** member is added. The transformation contains two operations. The first dereferences the pointer. In addition, as **x1** is derived from **xbase**, you need to upcast the variable when you want to include it.

  Notice that one expression is nested within another.
- **pStreet**: The **pStreet** member is added. The operations that are performed are the same as for **pName**.
- **pVoid1**: The **v** member is added. Because the application's definition of the data is **void** *, casting tells TotalView how it should interpret the information. In this example, the data is being cast into a pointer to a string.
- **pVoid2**: The **q** member is added. The transformation contains two operations. The first casts **q** into a pointer to the **x2** class. The second dereferences the pointer.

Using Type Transformations

When TotalView begins executing, it loads its built in transformations. To locate the directory in which these files are stored, use the following CLI command:

```
dset TOTALVIEW_TCLLIB_PATH
```

Type transformations are always loaded. By default, they are turned on. From the GUI, you can control whether transformations are turned on or off by going to the **Options** Page of the **File > Preferences** Dialog Box and changing the **View simplified STL containers (and user-defined transformations)** item. For example, the following turns on type transformations:

```
dset TV::ttf true
```

Part II: Running TotalView

This section of the *TotalView Reference Guide* contains information about command-line options you use when starting TotalView and the TotalView Debugger Server.

Chapter 7: TotalView Command Syntax
: TotalView contains a great number of command-line options. Many of these options allow you to override default behavior or a behavior that you've set in a preference or a startup file.

Chapter 8: TotalView Debugger Server Command Syntax
: This chapter describes how you modify the behavior of the **tvdsvr**. These options are most often used if a problem occurs in launching the server or if you have some very specialized need. In most cases, you can ignore the information in this chapter.

TotalView Command Syntax

chapter 7

This chapter describes the syntax of the **totalview** command. Topics in this chapter are:

- Command-Line Syntax
- Command-Line Options

Command-Line Syntax

Format: **totalview** [*options*] [*filename* [*corefile*]] [*-a* [*args*]]

or

totalview [*options*] **-args** *filename* [*args*]

Arguments:

options	TotalView options.
filename	Specifies the path name of the executable being debugged. This can be an absolute or relative path name. The executable must be compiled with debugging symbols turned on, normally the **–g** compiler option. Any multiprocess programs that call **fork()**, **vfork()**, or **execve()** should be linked with the **dbfork** library.
corefile	Specifies the name of a core file. Use this argument in addition to *filename* when you want to examine a core file with TotalView.
args	Default target program arguments.

Description: TotalView is a source-level debugger with a motif-based graphic user interface and features for debugging distributed programs, multiprocess programs, and multithreaded programs. TotalView is available on a number of different platforms.

If you specify mutually exclusive options on the same command line (for example, –**dynamic** and –**no_dynamic**), the last option listed is used.

Command-Line Options

–**a** *args*	Pass all subsequent arguments (specified by *args*) to the program specified by *filename*. This option must be the last one on the command line.
–**aix_use_fast_ccw**	Defined only on AIX, a synonym for the platform-independent **-use_fast_wp**, for TotalView script backward compatibility . See **-use_fast_wp** for more information.You must set this option on the command line; you cannot set it interactively using the CLI.
–**aix_use_fast_trap**	Defined only on AIX, a synonym for the platform-independent **-use_fast_trap**, for TotalView script backward compatibility. See **-use_fast_trap** for more information. You must set this option on the command line; you cannot set it interactively using the CLI.
-args *filename* [*args*]	Specifies *filename* as the executable to debug, with *args* as optional arguments to pass to your program. This option must be listed last on the command line. You can also use **--args** instead of **-args**, for compatibility with other debuggers.
–**background** *color*	Sets the general background color to *color.*
default:	**light blue**
–**bg** *color*	Same as –**background**.
–**compiler_vars**	(Alpha, HP, and SGI only.) Shows variables created by the Fortran compiler, as well as those in the user's program. Some Fortran compilers (HP f90/f77, HP f90, SGI 7.2 compilers) output debugging information that describes variables the compiler itself has invented for purposes such as passing the length of character*(*) variables. By default, TotalView suppresses the display of these compiler-generated variables. However, you can specify the –**compiler_vars** option to display these variables. This is useful when you are looking for a corruption of a run-time descriptor or are writing a compiler.

–no_compiler_vars
: (Default) Tells TotalView that it should not show variables created by the Fortran compiler.

–control_c_quick_shutdown
–ccq
: (Default) Tells TotalView to kills attached processes and exits.

–no_control_c_quick_shutdown
–nccq
: Invokes code that sometimes allows TotalView to better manage the way it kills parallel jobs when it works with management systems. This has only been tested with SLURM. It may not work with other systems.

–dbfork
: (Default) Catches the **fork()**, **vfork()**, and **execve()** system calls if your executable is linked with the **dbfork** library.

–no_dbfork
: Tells TotalView that it should not catch **fork()**, **vfork()**, and **execve()** system calls even if your executable is linked with the **dbfork** library.

–debug_file *consoleoutputfile*
: Redirects TotalView console output to a file named *consoleoutputfile*.

All TotalView console output is written to **stderr**.

–default_parallel_attach_subset=*subset_specification*
: Specifies a list of MPI ranks to be attached to when an MPI job is created or attached to. S*ubset_specification* is a space-separated list, the elements of which can be in one of these forms:

rank: that rank only

rank1-ranks2: all ranks between rank1 and rank2 inclusive.

rank1-rank2:stride: every strideth rank between rank1 and rank2

A rank specification can be either a number or "**$max**" (the last rank in the MPI job).

–demangler=*compiler*
: Overrides the demangler and mangler TotalView uses by default. The following indicate override options.

–demangler=compaq	HP cxx on Linux (alpha)
–demangler=dec	HP Tru64 C++ or Fortran
–demangler=gnu	GNU C++ on Linux Alpha
–demangler=gnu_dot	GNU C++ on Linux x86
–demangler=gnu_v3	GNU C++ Linux x86
–demangler=hp	HP aCC compiler
–demangler=irix	SGI IRIX C++
–demangler=kai	KAI C++

–demangler=kai3_n	KAI C++ version 3.n
–demangler=kai_4_0	KAI C++
–demangler=spro	SunPro C++ 4.0 or 4.2
–demangler=spro5	SunPro C++ 5.0 or later
–demangler=sun	Sun CFRONT C++
–demangler=xlc	IBM XLC/VAC++ compilers

–display *displayname*

Set the name of the X Windows display to *displayname*. For example, **–display vinnie:0.0** will display TotalView on the machine named "vinnie."

default: The value of your **DISPLAY** environment variable.

–dll_ignore_prefix *list*

The colon-separated argument to this option tells TotalView that it should ignore files having this prefix when making a decision to ask about stopping the process when it *dlopens* a dynamic library. If the DLL being opened has any of the entries on this list as a prefix, the question is not asked.

–dll_stop_suffix *list*

The colon-separated argument to this option tells TotalView that if the library being opened has any of the entries on this list as a suffix, it should ask if it should open the library.

–dpvm

HP Tru64 UNIX *only*: Enable support for debugging the HP Tru64 UNIX implementation of Parallel Virtual Machine (PVM) applications.

–dump_core

Allows TotalView to dump a core file of itself when an internal error occurs. This is used to help TotalView Technologies debug problems.

–e *commands*

Tells TotalView to immediately execute the CLI commands named within this argument. All information you enter here is sent directly to the CLI's Tcl interpreter. For example, the following writes a string to **stdout**:

```
cli -e 'puts hello'
```

You can have more than one **–e** option on a command line.

–ent

Tells TotalView that it should only use an Enterprise license.

–no_ent

Tells TotalView that it should not use an Enterprise license. You may combine this with **–no_team** or **–noteamplus.**

–env *variable=valiue*

Tells TotalView to add an environment variable to the environment variables passed to your program by the shell. If the variable already exists, it effectively replaces the previous value. You need to use this command for each variable being added; that is, you cannot add more than one variable with an **env** command.

–foreground *color*

Sets the general foreground color (that is, the text color) to *color*.

default: **black**

–fg *color* Same as **–foreground**.

–f9x_demangler=*compiler*

Overrides the Fortran demangler and mangler TotalView uses by default. The following indicate override options.

–demangler=spro_f9x_4	SunPro Fortran, 4.0 or later
–demangler=xlf	IBM Fortran

–global_types (Default) Lets TotalView assume that type names are globally unique within a program and that all type definitions with the same name are identical. The C++ standard asserts that this must be true for standard-conforming code.

If this option is set, TotalView will attempt to replace an opaque type (**struct foo *p;**) declared in one module, with an identically named defined type in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable by using the non-opaque type definition.

–no_global_types

Specifies that TotalView *cannot* assume that type names are globally unique in a program. You should specify this option if your code has multiple different definitions of the same named type, since otherwise TotalView can use the wrong definition for an opaque type.

–gnu_debuglink Tells TotalView that if a program or library has a **.gnu_debug_link** section, it should look for a **gnu_debug_link** file. If found, TotalView reads the debugging information from this file.

–no_gnu_debuglink

Do not load information from a **gnu_debug_link** file even if the file has a **.gnu_debug_link** section.

–gnu_debuglink_checksum

Tells TotalView that it should validate the **gnu_debug_link** file's checksum against the checksum contained in the process's **.gnu_debuglink** section.

–no_gnu_debuglink_checksum

Do not compare checksums. Only do this if you are absolutely certain that the debug file matches.

–ipv6_support Directs TotalView to support IPv6 addresses.

–no_ipv6_support
(Default) Do not support IPv6 addresses.

–kcc_classes (Default) Converts structure definitions output by the KCC compiler into classes that show base classes and virtual base classes in the same way as other C++ compilers. See the description of the **TV::kcc_classes** variable for a description of the conversions that TotalView performs.

–no_kcc_classes
Specifies that TotalView will not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than as data.

–lb (Default) Loads action points automatically from the *filename*.**TVD.v3breakpoints** file, providing the file exists.

–nlb Tells TotalView that it should not automatically load action points from an action points file.

–local_interface *string*
Sets the interface name that the server uses when it makes a callback. For example, on an IBM PS2 machine, you would set this to css0. However, you can use any legal **inet** interface name. (You can obtain a list of the interfaces if you use the **netstat -i** command.)

–memory_debugging
Enables memory debugging. By adding the following suboptions, you enable that particular feature using its the feature's default configuration. In most cases, you will want to use one or more of the following suboptions.

–mem_detect_use_after_free
Tests for memory use after memory is freed.

–mem_guard_blocks
Surrounds allocated memory blocks with guard blocks.

–mem_hoard_freed_memory
Tells the Memory Debugger to hoard memory blocks instead of releasing them when a **free()** routine is called.

–mem_hoard_low_memory_threshold *nnnn*
Sets the low memory threshold amount. When memory falls below this amount an event will be fired.

–mem_notify_events
Turns on memory event notification.

–no_mem_notify_events turns event notification off.

–mem_paint_all
Paint both allocated and deallocated blocks with a bit pattern.

–mem_paint_on_alloc
Paint memory blocks with a bit pattern when they are allocated.

–mem_paint_on_dealloc
Paint memory blocks with a bit pattern when they are freed.

–mem_red_zones_overruns
Turn on testing for Red Zones overruns.

–mem_red_zones_size_ranges min:max,min:max,...
Defines the memory allocations ranges for which Red Zones are in effect. Ranges can be specified as follows:

x:y	allocations from x to y
:y	allocations from 1 to y
x:	allocations of x and higher
x	allocation of x

–mem_red_zones_underruns
Turn on testing for Red Zones underruns.

–message_queue
(Default) Enables the display of MPI message queues when debugging an MPI program.

–mqd Same as –**message_queue**.

–no_message_queue
Disables the display of MPI message queues when you are debugging an MPI program. This might be useful if something is overwriting the message queues and causing TotalView to become confused.

–no_mqd
Same as –**no_message_queue**.

–mpi *starter* Names the MPI that your program requires. The list of starter names that you enter are those that appear in the **Parallel system** pull down list contained within the **New Program**’s Parallel tab. If the starter name has more than one word (for example, **Open MPI**), enclose the name in quotes. For example:

```
-mpi “Open MPI”
```

–nodes Specifies the number of nodes upon which the MPI job will run.

–np Specifies how many tasks that TotalView should launch for the job. This argument usually follows a –**mpi** command-line option.

–nptl_threads Tells TotalView that your application is using NPTL threads. You only need use this option if default cannot determine that you are using this threads package.

–no_nptl_threads
Tells TotalView that you are not using the NPTL threads package. Use this option if TotalView thinks your application is using it and it isn’t.

–parallel (Default) Enables handling of parallel program run-time libraries such as MPI, PE, and UPC.

–no_parallel
Disables handling of parallel program run-time libraries such as MPI, PE, and UPC. This is useful for debugging parallel programs as if they were single-process programs.

–parallel_attach *option*
Sets the action of TotalView when starting a parallel program.Possible options are:

yes (default)	Attach to all processes
no	Attach to no processes
ask	Ask which to attach to

–patch_area_base *address*
Allocates the patch space dynamically at *address*. See "*Allocating Patch Space for Compiled Expressions*" in Chapter 15 of the *TotalView Users Guide*.

–patch_area_length *length*
Sets the length of the dynamically allocated patch space to this *length*. See "*Allocating Patch Space for Compiled Expressions*" in the *TotalView Users Guide*.

–pid *pid*
Tells TotalView to attach to process *pid* after it starts executing.

–procs
Specifies how many tasks that TotalView should launch for the job. This argument usually follows a **–mpi** command-line option.

–pvm
Enables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications.

–remote *hostname*[*:portnumber*]
Debugs an executable that is not running on the same machine as TotalView. For *hostname*, you can specify a TCP/IP host name (such as **vinnie**) or a TCP/IP address (such as **128.89.0.16**). Optionally, you can specify a TCP/IP port number for *portnumber*, such as **:4174**. When you specify a port number, you disable the autolaunch feature. For more information on the autolaunch feature, see "*Setting Single Process Server Launch*" in the *TotalView Users Guide*.

–r *hostname*[*:portnumber*]
Same as **–remote**.

–replay
Tells TotalView that it should enable the ReplayEngine when TotalView begins. This command-line option is ignored if you do not have a license for ReplayEngine.

–s *pathname*
Specifies the path name of a startup file that will be loaded and executed. This path name can be either an absolute or relative name.

You can add more than one **–s** option on a command line.

–serial *device*[:*options*]

Debugs an executable that is not running on the same machine as TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently, the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "*Debugging Over a Serial Line*" in Chapter 4 of the *TotalView Users Guide*.

–search_path *pathlist*

Specify a colon-separated list of directories that TotalView will search when it looks for source files. For example:

```
totalview –search_path proj/bin:proj/util
```

–signal_handling_mode "*action_list***"**

Modifies the way in which TotalView handles signals. You must enclose the *action_list* string in quotation marks to protect it from the shell.

An *action_list* consists of a list of *signal_action* descriptions separated by spaces:

signal_action[*signal_action*] ...

A signal action description consists of an action, an equal sign (=), and a list of signals:

action=signal_list

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**, For more information on the meaning of each action, see Chapter 3 of the *TotalView Users Guide*.

A *signal_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as 11), or a star (*), which specifies all signals. We recommend that you use the signal name rather than the number because number assignments vary across UNIX sessions.

The following rules apply when you are specifying an *action_list*:

(1) If you specify an action for a signal in an *action_list*, TotalView changes the default action for that signal.

(2) If you do not specify a signal in the *action_list*, TotalView does not change its default action for the signal.

(3) If you specify a signal that does not exist for the platform, TotalView ignores it.

(4) If you specify an action for a signal more than once, TotalView uses the last action specified.

If you need to revert the settings for signal handling to built-in defaults, use the **Defaults** button in the **File > Signals** dialog box.

For example, here's how to set the default action for the **SIGTERM** signal to resend:

```
"Resend=SIGTERM"
```

Here's how to set the action for **SIGSEGV** and **SIGBUS** to error, the action for **SIGHUP** to resend, and all remaining signals to stop:

```
"Stop=* Error=SIGSEGV,SIGBUS Resend=SIGHUP"
```

–shm "*action_list*"
Same as **–signal_handling_mode**.

–starter_args "*arguments*"
Tells TotalView to pass *arguments* to the starter program. You can omit the quotation marks if *arguments* is just one string without any embedded spaces.

–stderr *pathname*
Names the file to which your program will write **stderr** information while executing within TotalView. If the file exists, TotalView overwrites it.

–stderr_append
Tells TotalView to append the information if writes to **stderr** to the file named in the **–stderr** command or the file named in GUI or a TotalView variable. If the file does not exist, TotalView creates it.

–stderr_is_stdout *pathname*
Tells TotalView to redirect **stderr** to **stdout**.

–stdin *pathname*
Names the file from which your program will read information while executing within TotalView.

–stdout *pathname*
Names the file to which your program will write **stdout** information while executing within TotalView. If the file exists, TotalView overwrites it.

–stdout_append
Tells TotalView to append the information if writes to **stdout** to the file named in the **–stdout** command or the file named in GUI or a TotalView variable. If the file does not exist, TotalView creates it.

–tasks
Specifies how many tasks that TotalView should launch for the job. This argument usually follows a **–mpi** command-line option.

–team
Tells TotalView that it should only use a Team license.

–no_team
Tells TotalView that it should not use an Enterprise license. You may combine this with **–no_ent** or **–noteamplus.**

–teamplus
Tells TotalView that it should only use a Team Plus license.

–no_teamplus
Tells TotalView that it should not use a Team PLus license. You may combine this with **–no_ent** or **–noteam.**

–**tvhome** *pathname*	The directory from which TotalView reads preferences and other related information and the directory to which it writes this information.
–**use_fast_trap**	Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional breakpoints, also known as EVAL points. As of TotalView 8.7, when this was introduced, only AIX supported the fast trap mechanism for breakpoints, but we anticipate other operating systems adding support. You must set this option on the command line; you cannot set it interactively using the CLI. Your operating system may not be configured correctly to support this option. See the TotalView Release Notes on our web site for more information.
–**use_fast_wp**	Controls TotalView's use of the target operating system's support of the fast trap mechanism for compiled conditional watchpoints, also known as CDWP points. As of TotalView 8.7, when this was introduced, only AIX supported the fast trap mechanism for watchpoints, but we anticipate other operating systems adding support. You must set this option on the command line; you cannot set it interactively using the CLI. Your operating system may not be configured correctly to support this option. See the TotalView Release Notes on our web site for more information.
–**user_threads**	(Default) Enables handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported.
–**no_user_threads**	Disables handling of user-level (M:N) thread packages. This option may be useful in situations where you need to debug kernel-level threads, but in most cases, this option is of little use on systems where two-level thread scheduling is used.
–**xterm_name** *pathname*	Sets the name of the program used when TotalView needs to create a the CLI. If you do not use this command or have not set the **TV::xterm_name** variable, TotalView will attempt to create an **xterm** window.
–**verbosity** *level*	Sets the verbosity level of TotalView messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.
default:	**info**

TotalView Debugger Server Command Syntax

chapter 8

This chapter summarizes the syntax of the TotalView Debugger Server command, **tvdsvr**, which is used for remote debugging. Remote debugging occurs when you explicitly call for it or when you are using disciplines like MPI that startup processes on remote servers. For more information on remote debugging, refer to "*Setting Up Remote Debugging Sessions*" in the *TotalView Users Guide*.

Topics in this chapter are:

- The **tvdsvr** Command and Its Options
- Replacement Characters

The tvdsvr Command and Its Options

tvdsvr {**–server** | **–callback** *hostname*:*port* | **–serial** *device*}
[*other options*]

Description:

tvdsvr allows TotalView to control and debug a program on a remote machine. To accomplish this, the **tvdsvr** program must run on the remote machine, and it must have access to the executables being debugged. These executables must have the same absolute path name as the executable that TotalView is debugging, or the **PATH** environment variable for **tvdsvr** must include the directories containing the executables.

You must specify a **–server**, **–callback**, or **–serial** option with the **tvdsvr** command. By default, TotalView automatically launches **tvdsvr** using the **–callback** option, and the server establishes a connection with TotalView. (Automatically launching the server is called *autolaunching*.)

If you prefer not to automatically launch the server, you can start **tvdsvr** manually and specify the –**server** option. Be sure to note the password that **tvdsvr** prints out with the message:

pw = *hexnumhigh*:*hexnumlow*

TotalView will prompt you for *hexnumhigh*:*hexnumlow* later. By default, **tvdsvr** automatically generates a password that it uses when establishing connections. If desired, you can set your own password by using the –**set_pw** option.

To connect to the **tvdsvr** from TotalView, you use the **File > New Program** Dialog Box and must specify the host name and TCP/IP port number, *hostname*:*portnumber* on which **tvdsvr** is running. Then, TotalView prompts you for the password for **tvdsvr**.

Options

The following options name the port numbers and passwords that TotalView uses to connect with **tvdsvr**.

–**callback** *hostname***:***port*

(Autolaunch feature only) Immediately establishes a connection with a TotalView process running on *hostname* and listening on *port*, where *hostname* is either a host name or TCP/IP address. If **tvdsvr** cannot connect with TotalView, it exits.

If you use the –**port,** –**search_port**, or –**server** options with this option, **tvdsvr** ignores them.

–**callback_host** *hostname*

Names the host upon which the callback is made. The *hostname* argument indicates the machine upon which TotalView is running. This option is most often used with a bulk launch.

–**callback_ports** *port-list*

Names the ports on the host machines that are used for callbacks. The *port-list* argument contains a comma-separated list of the host names and TCP/IP port numbers (*hostname*:*port*,*hostname*:*port*...) on which TotalView is listening for connections from **tvdsvr**. This option is most often used with a bulk launch.

For more information, see Chapter 4, "*Setting Up Remote Debugging Sessions*" in the *TotalView Users Guide*.

–**debug_file** *console_outputfile*

Redirects TotalView Debugger Server console output to a file named *console_outputfile*.

Default: All console output is written to **stderr**.

–dpvm Uses the HP Tru64 UNIX implementation of the Parallel Virtual Machine (DPVM) library process as its input channel and registers itself as the DPVM tasker.

This option is not intended for users launching tvdsvr manually. When you enable DPVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.

–nodes_allowed *num*

Explicitly tells tvdsvr how many nodes the server supports and how many licenses it needs. This is only used for the Cray XT3.

–port *number* Sets the TCP/IP port number on which **tvdsvr** should communicate with TotalView. If this port is busy, **tvdsvr** does not select an alternate port number (that is, it won't communicate with anything) unless you also specify –**search_port**.

Default: 4142

–pvm Uses the ORNL implementation of the Parallel Virtual Machine (PVM) library process as its input channel and registers itself as the ORNL PVM tasker.

This option is not intended for users launching tvdsvr manually. When you enable PVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.

–search_port Searches for an available TCP/IP port number, beginning with the default port (4142) or the port set with the –**port** option and continuing until one is found. When the port number is set, **tvdsvr** displays the chosen port number with the following message:

port = *number*

Be sure that you remember this port number, since you will need it when you are connecting to this server from TotalView.

–serial *device*[:*options*]

Waits for a serial line connection from TotalView. For *device*, specifies the device name of a serial line, such as **/dev/com1**. The only *option* you can specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "*Debugging Over a Serial Line*" in Chapter 4 of the *TotalView Users Guide*.

–server Listens for and accepts network connections on port 4142 (default).

Using –**server** can be a security problem. Consequently, you must explicitly enable this feature by placing an empty file named **tvdsvr.conf** in your **/etc** directory. This file must be owned by user ID 0 (root). When **tvdsvr** encounters this option, it checks if this file exists. This file's contents are ignored.

You can use a different port by using one of the following options: **–search_port** or **–port**. To stop **tvdsvr** from listening and accepting network connections, you must terminate it by pressing Ctrl+C in the terminal window from which it was started or by using the **kill** command.

–set_pw *hexnumhigh*:*hexnumlow*

Sets the password to the 64-bit number specified by the *hexnumhigh* and *hexnumlow* 32-bit numbers. When a connection is established between **tvdsvr** and TotalView, the 64-bit password passed by TotalView must match this password set with this option. **tvdsvr** displays the selected number in the following message:

pw = *hexnumhigh*:*hexnumlow*

We recommend using this option to avoid connections by other users.

If necessary, you can disable password checking by specifying the "–set_pw 0:0" option with the tvdsvr command. Disabling password checking is dangerous; it allows anyone to connect to your server and start programs, including shell commands, using your UID. Therefore, we do not recommend disabling password checking.

–set_pws *password-list*

Sets 64-bit passwords. TotalView must supply these passwords when **tvdsvr** establishes the connection with it. The argument to this command is a comma-separated list of passwords that TotalView automatically generates. This option is most often used with a bulk launch.

For more information, see Chapter 4, "*Setting Up Remote Debugging Sessions*" in the *TotalView Users Guide*.

–verbosity *level*

Sets the verbosity level of TotalView Debugger Server-generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

–working_directory *directory*

Makes *directory* the directory to which TotalView connects.

Note that the command assumes that the host machine and the target machine mount identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on both the host and target machines.

After performing this operation, the TotalView Debugger Server is started.

Replacement Characters

When placing a **tvdsvr** command in a **Server Launch** or **Bulk Launch** string (see the **File > Preferences** command within the online Help for more information), you will need to use special replacement characters. When your program needs to launch a remote process, TotalView replaces these command characters with what they represent. Here are the replacement characters:

%B	Expands to the bin directory where **tvdsvr** is installed.
%C	Is replaced by the name of the server launch command being used. On most platforms, this is **rsh**. On HP, this command is **remsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView will use its value instead of its platform-specific value.
%D	Is replaced by the absolute path name of the directory to which TotalView will be connected.
%F	Contains the "tracer configuration flags" that need to be sent to **tvdsvr** processes. These are system-specific startup options that the **tvdsvr** process needs.
%H	Expands to the host name of the machine upon which TotalView is running. (This replacement character is most often used in bulk server launch commands. However, it can be used in a regular server launch and within a **tvdsvr** command contained within a temporary file.)
%I	Expands to the **pid** of the MPI starter process. For example, it can contain **mpirun**, **aprun**, etc. It can also be the process to which you manually attach. If no **pid** is available, **%I** expands to 0.
%J	Expands to the job ID. For MPICH or poe jobs, is the contents of the **totalview_jobid** variable contained either in the starter or first process. If that variable does not exist, it is set to zero ("0"). If it is not appropriate for the kind of job being launched, its value is -1.
%K	(Red Storm and BlueGene architectures) If TotalView must use an alternative name for **tvdsvr**, specify its name here. For example, on BlueGene, the server name is **tvdsvr_bgl**. On Red Storm, it is **tvdsvr_rs**.
%L	If TotalView is launching one process, this is replaced by the host name and TCP/IP port number (*hostname:port*) on which TotalView is listening for connections from **tvdsvr**. If a bulk launch is being performed, TotalView replaces this with a comma-separated list of the host names and

	TCP/IP port numbers (*hostname*:*port*,*hostname*:*port*...) on which TotalView is listening for connections from **tvdsvr**. For more information, see Chapter 4, "*Setting Up Remote Debugging Sessions*" in the *TotalView Users Guide*.
%M	(Sun) Expands to the command name used for a local server launch.
%N	Is replaced by the number of servers that TotalView will launch. This is only used in a bulk server launch command.
%P	If TotalView is launching one process, this is replaced by the password that it automatically generated. If a bulk launch is being performed, TotalView replaces this with a comma-separated list of 64-bit passwords.
%R	Is replaced by the host name of the remote machine specified in the **File > New Program** command. When performing a bulk launch, this is replaced by a comma-separated list of the names of the hosts upon which TotalView will launch **tvdsvr** processes.
%S	If TotalView is launching one process, it replaces this symbol with the port number on the machine upon which TotalView is running. If a bulk server launch is being performed, TotalView replaces this with a comma-separated list of port numbers.
%t1 and %t2	Is replaced by files that TotalView creates containing information it generates. This is only available in a bulk launch. These temporary files have the following structure: (1) An optional header line containing initialization commands required by your system. (2) One line for each host being connected to, containing host-specific information. (3) An optional trailer line containing information needed by your system to terminate the temporary file. The **File > Preferences Bulk Server** Page allows you to define templates for the contents of temporary files. These files may use these replacement characters. The **%N**, **%t1**, and **%t2** replacement characters can only be used within header and trailer lines of temporary files. All other characters can be used in header or trailer lines or within a host line defining the command that initiates a single-process server launch. In header or trailer lines, they behave as defined for a bulk launch within the host line. Otherwise, they behave as defined for a single-server launch.
%U	(Sun) Expands to the local socket ID.

%V	Is replaced by the current TotalView verbosity setting.
%Z	Expands to the job ID. For MPICH or poe jobs, is the contents of the **totalview_jobid** variable contained either in the starter or first process. If that variable does not exist, it is set to zero ("0"). If it is not appropriate for the kind of job being launched, its value is -1.

Part III: Platforms and Operating Systems

The three chapters in this part of the Reference Guide describe information that is unique to the computers, operating systems, and environments in which TotalView runs.

Chapter 9: Platforms and Compilers
: Here you will find general information on the compilers and runtime environments that TotalView supports. This chapter also contains commands for starting TotalView and information on linking with the **dbfork** library.

Chapter 10: Operating Systems
: While how you use TotalView is the same on all operating systems, there are some things you will need to know that are differ from platform to platform.

Chapter 11: Architectures
: When debugging assembly-level functions, you will need to know how TotalView refers to your machines registers.

Platforms and Compilers

chapter 9

This chapter describes the compilers and parallel runtime environments used on platforms supported by TotalView. You must refer to the *TotalView Platforms and Systems Requirement Guide* for information on the specific compiler and runtime environments that TotalView supports.

For information on supported operating systems, please refer to Chapter 10, "*Operating Systems*," on page 275.

Topics in this chapter are:

- Compiling with Debugging Symbols
- Using gnu_debuglink Files
- Using Exception Data on Tru64 UNIX
- Linking with the dbfork Library

Compiling with Debugging Symbols

You need to compile programs with the **–g** option and possibly other compiler options so that debugging symbols are included. This section shows the specific compiler commands to use for each compiler that TotalView supports.xs

Please refer to the release notes in your TotalView distribution for the latest information about supported versions of the compilers and parallel runtime environments listed here.

Apple Running Mac OS X

The following table lists the procedures to compile programs on HP Alpha running Linux.

Compiler	Compiler Command Line
Absoft Fortran 77	f77 –g *program*.f f77 –g *program*.for
Absoft Fortran 90	f90 –g *program*.f90
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
GCC Fortran	g77 –g *program*.f
IBM xlc C	xlc –g *program*.c
IBM xlC C++	xlC –g *program*.cxx
IBM xlf Fortran 77	xlf –g *program*.f
IBM xlf90 Fortran 90	xlf90 –g *program*.f90

On Mac OS X 10.4, you can create 64-bit applications using GCC 4 by adding the –**m64** command-line option.

HP Tru64 UNIX

The following table lists the procedures to compile programs on HP Tru64 UNIX.

Compiler	Compiler Command Line
HP Tru64 UNIX C	cc –g *program*.c
HP Tru64 UNIX C++	cxx –g *program*.cxx
HP Tru64 UNIX Fortran 77	f77 –g *program*.f
HP Tru64 UNIX Fortran 90	f90 –g *program*.f90
HP Tru64 UPC compiler	upc -g [-fthreads *n*] *program*.upc
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx

HP-UX for IA-64

The following table lists the procedures to compile programs on HP HP-UX for IA-64.

Compiler	Compiler Command Line
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
GCC Fortran	g77 –g *program*.f
HP C	cc –g *program*.cc
HP aCC	aCC –g *program*.cxx
HP Fortran 77	f77 –g *program*.cc
HP Fortran 90	f90 –g *program*.cc

IBM AIX on RS/6000 Systems

The following table lists the procedures to compile programs on IBM RS/6000 systems running AIX.

Compiler	Compiler Command Line
GCC C	**gcc –g** *program*.**c**
GCC C++	**g++ –g** *program*.**cxx**
IBM xlc C	**xlc –g** *program*.**c**
IBM xlC C++	**xlC –g** *program*.**cxx**
IBM xlf Fortran 77	**xlf –g** *program*.**f**
IBM xlf90 Fortran 90	**xlf90 –g** *program*.**f90**

You can set up to seven variables when debugging threaded applications. Here's how you might set six of these variables within a C shell:

```
setenv AIXTHREAD_MNRATIO       "1:1"
setenv AIXTHREAD_SLPRATIO      "1:1"
setenv AIXTHREAD_SCOPE         "S"
setenv AIXTHREAD_COND_DEBUG    "ON"
setenv AIXTHREAD_MUTEX_DEBUG   "ON"
setenv AIXTHREAD_RWLOCK_DEBUG  "ON"
```

The first three variables must be set. Depending upon what you need to examine, you will also need to set one or more of the "DEBUG" variables.

The seventh variable, **AIXTHREAD_DEBUG**, should not be set. If it is, you should unset it before running TotalView

Setting these variables can slow down your application's performance. None of them should be set when you are running non-debugging versions of your program.

When compiling with KCC, you must specify the **–qnofullpath** option; KCC is a preprocessor that passes its output to the IBM xlc C compiler. It will discard **#line** directives necessary for source-level debugging if you do not use the **–qfullpath** option. We also recommend that you use the **+K0** option and not the **–g** option.

When compiling with **guidef77**, the **–WG,–cmpo=i** option may not be required on all versions because **–g** can imply these options.

When compiling Fortran programs with the C preprocessor, pass the **–d** option to the compiler driver. For example: **xlf –d – program.F**

If you will be moving any program compiled with any of the IBM *xl* compilers from its creation directory, or you do not want to set the search directory path during debugging, use the **–qfullpath** compiler option. For example:

```
xlf -qfullpath -g -c program.f
```

IBM Bluegene

The following table lists the procedures to compile programs on the IBM Blue Gene/L computer.

Compiler	Compiler Command Line
IBM Visual Age C	xlc –g *program*.c
IBM Visual Age C++	xlC –g *program*.cxx
IBM Visual Age FORTRAN 77	xlf –g *program*.f
IBM Visual Age Fortran 90	xlf90 –g *program*.cc

IBM Power Linux

The following table lists the procedures to compile programs on the IBM Power Linux computer.

Compiler	Compiler Command Line
Absoft Fortran 77	f77 –g *program*.f f77 –g *program*.for
Absoft Fortran 90	f90 –g *program*.f90
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
IBM Visual Age C	xlc –g *program*.c
IBM Visual Age C++	xlC –g *program*.cxx
IBM Visual Age FORTRAN 77	xlf –g *program*.f
IBM Visual Age Fortran 90	xlf90 –g *program*.cc

Linux Running on an x86 Platform

The following table lists the procedures to compile programs on Linux x86 platforms.

Compiler	Compiler Command Line
Absoft Fortran 77	f77 –g *program*.f f77 –g *program*.for
Absoft Fortran 90	f90 –g *program*.f90
Absoft Fortran 95	f95 –g *program*.f95
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
G77	g77 –g *program*.f
Intel C++ Compiler	icc –g *program*.cxx
Intel Fortran Compiler	ifc –g *program*.f
Lahey/Fujitsu Fortran	lf95 –g *program*.f
PGI Fortran 77	pgf77 –g *program*.f
PGI Fortran 90	pgf90 –g *program*.f

Linux Running on an x86-64 Platform

The following table lists the procedures to compile programs on Linux x86-64 platforms.

Compiler	Compiler Command Line
Absoft Fortran 77	f77 –g *program*.f f77 –g *program*.for
Absoft Fortran 90	f90 –g *program*.f90
Absoft Fortran 95	f95 –g *program*.f95
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
G77	g77 –g *program*.f
Intel C++ Compiler	icc –g *program*.cxx
Intel Fortran Compiler	ifc –g *program*.f
Pathscale EKO C	pathcc –g *program*.f
Pathscale EKO C++	pathCC –g *program*.f
Lahey/Fujitsu Fortran	lf95 –g *program*.f
PGI C++	pcCC –g *program*.f
PGI Fortran 77	pgf77 –g *program*.f
PGI Fortran 90	pgf90 –g *program*.f

Linux Running on an Itanium Platform

The following table lists the procedures to compile programs running on the Intel Itanium platform.

Compiler	Compiler Command Line
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
G77	g77 –g *program*.f
Intel C++ Compiler	icc –g *program*.cxx
Intel Fortran Compiler	ifc –g *program*.f

SGI IRIX-MIPS Systems

The following table lists the procedures to compile programs on SGI MIPS systems running IRIX.

Compiler	Compiler Command Line
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
G77	g77 –g *program*.f
Intrepid (GCC UPC)	upc -g [-fthreads *n*] *program*.upc
SGI MIPSpro C	cc –n32 –g *program*.c cc –64 –g *program*.c
SGI MIPSpro C++	CC –n32 –g *program*.cxx CC –64 –g *program*.cxx

Compiler	Compiler Command Line
SGI MIPSpro77	f77 –n32 –g *program*.f f77 –64 –g *program*.f
SGI MIPSpro 90	f90 –n32 –g *program*.f90 f90 –64 –g *program*.f90

TotalView does not support compiling with **–32**, which is the default for some compilers. You must specify either **–n32** or **–64**.

Sun Solaris

The following table lists the procedures to compile programs on SunOS 5 SPARC.

Compiler	Compiler Command Line
Apogee C	apcc –g *program*.c
Apogee C++	apcc –g *program*.cxx
GCC C	gcc –g *program*.c
GCC C++	g++ –g *program*.cxx
Sun One Studio C	cc –g *program*.c
Sun One Studio C++	CC –g *program*.cxx
Sun One Studio Fortran 77	f77 –g *program*.f
Sun One Studio Fortran 90	f90 –g *program*.f90

Using gnu_debuglink Files

Some versions of Linux allow you to place debugging information in a separate file. These files, which can have any name, are called *gnu_debuglink* files. Because this information is stripped from the program's file, it almost always greatly reduces the size of your program. In most cases, you would create gnu_debuglink files for system libraries or other programs for which it is inappropriate to ship versions have debugging information.

After you create an unstripped executable or shared library, you can prepare the gnu_debuglink file as follows:

1 Create a **.debug** copy of the file. This second file will only contain debugging symbol table information. That is, it differs from the original in that it does not contain code or data.

  Create this file on Linux systems that support the – –**add-gnu-debuglink** and – –**only-keep-debug** command-line options. If **objcopy -– –help** mentions – –**add-gnu-debuglink**, you should be able to create this file. See **man objcopy** for more details.

2 Create a stripped copy of the image file, and add a **.gnu_debuglink** section to the stripped file that contains the name of the **.debug** file and the checksum of the **.debug** file.

3 Distribute the stripped image and **.debug** files separately. The idea is that the stripped image file will normally take up less space on the disk, and if you want the debug information, you can also install the corresponding **.debug** file.

The following example creates the gnu_debuglink file for a program named **hello**. It also strips the debugging information from **hello**:

```
objcopy --only-keep-debug hello
hello.gnu_debuglink.debug
objcopy --strip-all hello hello.gnu_debuglink
objcopy --add-gnu-debuglink=hello.gnu_debuglink.debug \
      hello.gnu_debuglink
```

Total View Command-Line Options and CLI State Variables

The following command line options and CLI variables control how TotalView handles **.gnu_debuglink** files.

- **–gnu_debuglink** and **–no_gnu_debuglink**, **TV::gnu_debuglink**
  Controls Total View processing of the **.gnu_debuglink** section in executables and shared libraries; the default value is true. Setting the variable to false or using the **no_** command-line option prefix saves time when you do not want to process the debug-only files or when you need to avoid other problems associated with the debug-only files.
- **–[no_]gnu_debuglink_checksum** and **TV::gnu_debuglink_checksum_flag**
  Tells TotalView if it should validate the checksum of the debug-only files against the checksum stored in the .**gnu_debuglink** section of the executable or shared library; the default is true. Setting the variable to false or using the **no_** command-line option prefix can save time associated with computing the checksum of large files. Do this only if you are absolutely certain that the debug file matches.
- **–gnu_debuglink_global_directory** and **TV::gnu_debuglink_global_directory**
  Specifies the global debug directory; the default value is **/usr/lib/debug**.

Searching for the gnu_debug_link File

If the **TV::gnu_debuglink_flag** variable is true and if the process contains a **.gnu_debug_link** section, TotalView searches for the **gnu_debug_link** file as follows:

1 In the directory containing the program.
2 In the **.debug** subdirectory of the directory containing the program.
3 In a directory named in the **TV::gnu_debuglink_global_directory** variable.

For example, assume that the program's pathname is **/A/B/hello_world** and the debug filename stored in the **.gnu_debuglink** section of this program is hello_world.debug. If the TV::gnu_debuglink_global_directory variable is set to /usr/lib/debug, TotalView searches for the following files:

1 /A/B/hello_world.debug
2 /A/B/.debug/hello_world.debug
3 /usr/lib/debug/A/B/hello_world.debug

Using Exception Data on Tru64 UNIX

If you receive the following error message when you load an executable into TotalView, you may need to compile your program so that it includes exception data.

```
Cannot find exception information. Stack backtraces may
not be correct.
```

To provide a complete stack backtrace in all situations, TotalView needs for you to include exception data with the compiled executable. To compile with exception data, you need to use the following options:

cc –Wl,–u,_fpdata_size *program.c*

where:

–Wl	Passes the arguments that follow to another compilation phase (–W), which in this case is the linker (l). Each argument is separated by a comma (,).
–u,_fpdata_size	Causes the linker to mark the next argument (_fpdata_size) as undefined. This forces the exception data into the executable.
program.c	Is the name of your program.

Compiling with exception data increases the size of your executable slightly. If you choose not to compile with exception data, TotalView can provide correct stack backtraces in most situations, but not in all situations.

Linking with the dbfork Library

If your program uses the **fork()** and **execve()** system calls, and you want to debug the child processes, you need to link programs with the **dbfork** library.

While you must link programs that use **fork()** *and* **execve()** *with the TotalView dbfork library so that TotalView can automatically attach to them when your program creates them, programs that you attach to need not be linked with this library.*

Linking with dbfork and HP Tru64 UNIX

Add one of the following command-line options to the command that you use to link your programs:

- /opt/totalview/alpha/lib/libdbfork.a
- –L/opt/totalview/alpha/lib –ldbfork

For example:

```
cc -o program program.c \
         -L/opt/totalview/alpha/lib -ldbfork
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **–L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/alpha/lib
```

Linking with HP-UX

Add either the **–ldbfork** or **–ldbfork_64** argument to the command that you use to link your programs. If you are compiling 32-bit code, use one of the following arguments:

- /opt/totalview/lib/hpux11-hppa/libdbfork.a
- –L/opt/totalview/hpux11-hppa/lib –ldbfork

For example:

```
cc -n32 -o program program.c \
         -L/opt/totalview/hpux11-hppa/lib -ldbfork
```

If you are compiling 64-bit code, use the following arguments:

- /opt/totalview/lib/hpux11-hppa/libdbfork_64.a
- –L/opt/totalview/hpux11-hppa/lib –ldbfork_64

For example:

```
cc -64 -o program program.c \
         -L/opt/totalview/hpux11-hppa/lib \
         -ldbfork_64
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **–L** command-line option. For example:

```
setenv LD_LIBRARY_PATH \
         /opt/totalview/hpux11-hppa/lib
```

dbfork on IBM AIX on RS/6000 Systems

Add either the
–**ldbfork** or –**ldbfork_64** argument to the command that you use to link your programs. If you are compiling 32-bit code, use the following arguments:

- /usr/totalview/lib/libdbfork.a \
  –bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
- –L/usr/totalview/lib \
  –ldbfork –bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a

For example:

```
cc -o program program.c \
   -L/usr/totalview/rs6000/lib/ -ldbfork \
   -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

If you are compiling 64-bit code, use the following arguments:

- /usr/totalview/lib/libdbfork_64.a \
  –bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
- –L/usr/totalview/lib –ldbfork_64 \
  –bkeepfile:/usr/totalviewrs6000//lib/libdbfork.a

For example:

```
cc -o program program.c \
   -L/usr/totalview/rs6000/lib -ldbfork \
   -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

When you use **gcc** or **g++**, use the –**Wl,–bkeepfile** option instead of using the –**bkeepfile** option, which will pass the same option to the binder. For example:

```
gcc -o program program.c \
   -L/usr/totalview/rs6000/lib -ldbfork -Wl, \
   -bkeepfile:/usr/totalview/rs6000/lib/libdbfork.a
```

Linking C++ Programs with dbfork

You cannot use the –**bkeepfile** binder option with the IBM xlC C++ compiler. The compiler passes all binder options to an additional pass called **munch**, which will not handle the –**bkeepfile** option.

To work around this problem, we have provided the C++ header file **libdbfork.h**. You must include this file somewhere in your C++ program. This forces the components of the **dbfork** library to be kept in your executable. The file **libdbfork.h** is included only with the RS/6000 version of TotalView. This means that if you are creating a program that will run on more than one platform, you should place the **include** within an **#ifdef** statement's range. For example:

```
#ifdef _AIX
#include "/usr/totalview/include/libdbfork.h"
#endif
int main (int argc, char *argv[])
{
}
```

In this case, you would not use the **–bkeepfile** option and would instead link your program using one of the following options:

- **/usr/totalview/include/libdbfork.a**
- **–L/usr/totalview/include –ldbfork**

Linux or Mac OS X

Add one of the following arguments or command-line options to the command that you use to link your programs:

- **/usr/totalview/***platform***/lib/libdbfork.a**
- **-L/usr/totalview/***platform***/lib –ldbfork**
  or
  -L/usr/totalview/*platform***/lib –ldbfork_64 (**

where *platform* is one of the following: **darwin-power**, **linux-x86**, **linux-x86-64**, or **linux-ia64**.

In general, 32-bit programs use **libdbfork.a** and 64-bit programs use **libdbfork_64.a**. Of course, if your architecture doesn't support 32-bit programs, the option won't work.

For example:

```
cc -o program program.c \
         -L/usr/totalview/linux-x86/lib -ldbfork
```

However, **linux-ia64** uses **libdbfork** for 64-bit programs.

SGI IRIX6-MIPS

Add one of the following arguments or command-line options to the command that you use to link your programs.

If you are compiling your code with **–n32**, use the following arguments:

- **/opt/totalview/irix6-mips/lib/libdbfork_n32.a**
- **–L/opt/totalview/irix6-mips/lib –ldbfork_n32**

For example:

```
cc -n32 -o program program.c \
         -L/opt/totalview/irix6-mips/lib -ldbfork_n32
```

If you are compiling your code with **–64**, use the following arguments:

- **/opt/totalview/irix6-mips/lib/libdbfork.a_n64.a**
- **–L/opt/totalview/irix6-mips/lib –ldbfork_n64**

For example:

```
cc -64 -o program program.c \
         -L/opt/totalview/irix6-mips/lib -ldbfork_n64
```

As an alternative, you can set the **LD_LIBRARY_PATH** environment variable and omit the **–L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/irix6-mips/lib
```

SunOS 5 SPARC

Add one of the following command line arguments or options to the command that you use to link your programs:

- /opt/totalview/sun5/lib/libdbfork.a
- –L/opt/totalview/sun5/lib –ldbfork

For example:

```
cc -o program program.c \
         -L/opt/totalview/sun5/lib -ldbfork
```

As an alternative, you can set the LD_LIBRARY_PATH environment variable and omit the –L option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/sun5/lib
```

Operating Systems

chapter 10

This chapter describes the operating system features that can be used with TotalView. This chapter includes the following topics:

- Supported Operating Systems
- Mounting the /proc File System (HP Tru64 UNIX, IRIX, and SunOS 5 only)
- Swap Space
- Shared Libraries
- Debugging Your Program's Dynamically Loaded Libraries
- Remapping Keys (Sun Keyboards only)
- Expression System

Supported Operating Systems

Here is an overview of operating systems and some of the environments supported by TotalView at the time when this book was printed. As this book isn't printed nearly as often as vendors update compilers and operating systems, the compiler and operating system versions mentioned here may be obsolete. For a definitive list, see the *TotalView Platforms and Systems Requirements* document on our web site. You can locate this document by going to **http://www.totalviewtech.com/Documentation/**".

- Apple Macintosh OS X 10.4.
- Cray XT3 (UNICOS/lc environment) running SuSE Linux Enterprise Server 8.2.99 and 9.
- HP Alpha workstations running HP Tru64 UNIX versions V4.0F and V5.1B.
- HP HP-UX for IA-64 systems running version 2.0 (11.23).
- IBM Blue Gene systems running SuSE Linux Enterprise Server 9 for the front end nodes.

- IBM Power Linux running SuSE Linux Enterprise Server 9, 9 SP 2, and 10.
- IBM RS/6000 and SP systems running AIX versions 5.1L, 5.2L, and 5.3L.
- SGI IRIX 6.4.21f and 6.5.27f on any MIPS R4000, R4400, R4600, R5000, R8000, R10000, and R12000 processor-based systems.
- Sun Opteron, Solaris 9
- Sun SPARC Solaris 9 and 10.
- Linux: see the Platforms Guide.

Mounting the /proc File System

To debug programs on HP Tru64 UNIX, SunOS 5, and IRIX with TotalView, you need to mount the **/proc** file system.

If you receive one of the following errors from TotalView, the **/proc** file system might not be mounted:

- `job_t::launch, creating process: process not found`
- `Error launching process while trying to read dynamic symbols`

-
  ```
  Creating Process... Process not found
  Clearing Thrown Flag
  Operation Attempted on an unbound process object
  ```

To determine whether the **/proc** file system is mounted, enter the appropriate command from the following table.

Operating System	Command

HP Tru64 UNIX	`% /sbin/mount -t procfs`
	`/proc on /proc type procfs (rw)`

SunOS 5	`% /sbin/mount | grep /proc`
	`/proc on /proc read/write/setuid on ...`

IRIX	`% /sbin/mount | grep /proc`
	`/proc on /proc type proc (rw)`

If you receive one of these messages from the **mount** command, the **/proc** file system is mounted.

Mounting /proc HP Tru64 UNIX and SunOS 5

To make sure that the **/proc** file system is mounted each time your system boots, add the appropriate line from the following table to the appropriate file.

Operating System	Name of File	Line to add
HP Tru64 UNIX	/etc/fstab	/proc /proc procfs rw 0 0
SunOS 5	/etc/vfstab	/proc - /proc proc - no -

Then, to mount the **/proc** file system, enter the following command:

```
/sbin/mount /proc
```

Mounting proc SGI IRIX

To make sure that the **/proc** file system is mounted each time your system boots, make sure that **/etc/rc2** issues the **/etc/mntproc** command. Then, to mount the **/proc** file system, enter the following command:

```
/etc/mntproc
```

Swap Space

Debugging large programs can exhaust the swap space on your machine. If you run out of swap space, TotalView exits with a fatal error, such as:

- `Fatal Error: Out of space trying to allocate`

  This error indicates that TotalView failed to allocate dynamic memory. It can occur anytime during a debugging session. It can also indicate that the data size limit in the C shell is too small. You can use the C shell's **limit** command to increase the data size limit. For example:

  ```
  limit datasize unlimited
  ```
- `job_t::launch, creating process: Operation failed`

  This error indicates that the **fork()** or **execve()** system call failed while TotalView was creating a process to debug. It can happen when TotalView tries to create a process.

Swap Space on HP Tru64 UNIX

To find out how much swap space has been allocated and is currently being used, use the **swapon** command on HP Tru64 UNIX.

To find out how much swap space is in use while you are running TotalView:

```
/bin/ps -o LFMT
```

To add swap space, use the **/sbin/swapon**(8) command. You must be logged in as **root** to use this command. For more information, refer to the online manual page for this command.

Swap Space on HP HP-UX

The **swapinfo** command on an HP-UX system lets you find out how much swap space is allocated and is being used.

To find out how much swap space is being used while TotalView is running, enter:

```
/usr/bin/ps -lf
```

Here is an example of what you might see. The **SZ** column shows the pages occupied by a program.

To add swap space, use the**/usr/sbin/swapon**(1M) command or the **SAM** (System Administration Manager) utility. If you use **SAM**, invoke the **Swap** command in the **Disks and File Systems** menu.

Maximum Data Size

To see the current data size limit in the C shell, enter:

```
limit datasize
```

The following command displays the current *hard* limit:

```
limit –h datasize
```

If the current limit is lower than the hard limit, you can easily raise the current limit. To change the current limit, enter:

limit datasize *new_data_size*

If the hard limit is too low, you must reconfigure and rebuild the kernel, and then reboot. This is most easily done using **SAM**.

To change **maxdsiz**, use the following path through the **SAM** menus:

```
Kernel Configuration > Configurable Parameters >
    maxdsiz > Actions > Modify Configurable Parameter >
    Specify New Formula/Value > Formula/Value
```

You can now enter the new maximum data segment size.

You may also need to change the value for **maxdsiz_64**.

Here is the command that lets you rebuild the kernel with these changed values:

```
Configurable Parameter > Actions >
Process New Kernel
```

Answer **yes** to process the kernel modifications, **yes** to install the new kernel, and **yes** again to reboot the machine with the new kernel.

When the machine reboots, the value you set for **maxdsiz** should be the new hard limit.

Swap Space on IBM AIX

To find out how much swap space has been allocated and is currently being used, use the **/usr/sbin/pstat -s** command:

To find out how much swap space is in use while you are running TotalView:

1 Start TotalView with a large executable:

   ```
   totalview executable
   ```

   Press Ctrl+Z to suspend TotalView.

2 Use the following command to see how much swap space TotalView is using:

   ```
   ps u
   ```

For example, in this case the value in the SZ column is 5476 KB:

```
USER     PID %CPU %MEM     SZ  RSS     TTY  ...
smith 15080  0.0 6.0 5476 5476  pts/1 ...
```

To add swap space, use the AIX system management tool, **smit**. Use the following path through the **smit** menus:

```
System Storage Management > Logical Volume Manager >
   Paging Space
```

Swap Space on Linux

To find out how much swap space has been allocated and is currently being used, use either the **swapon** or **top** commands on Linux:

You can use the **mkswap**(8) command to create swap space. The **swapon**(8) command tells Linux that it should use this space.

Swap Space on SGI IRIX

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:

   totalview *executable*

   Press Ctrl+Z to suspend TotalView.
2. Use the following command to see how much swap space TotalView is using:

   ```
   /bin/ps -l
   ```

   Use the following command to determine the number of bytes in a page:

   ```
   sysconf PAGESIZE
   ```

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be root to use these commands. For more information, refer to the online manual pages for these commands.

Swap Space on SunOS 5

To find out how much swap space has been allocated and is currently being used, use the **swap -s** command:

To find out how much swap space is in use while you are running TotalView:

1. Start TotalView with a large executable:

   totalview *executable*

   Press Ctrl+Z to suspend TotalView.
2. Use the following command to see how much swap space TotalView is using:

   ```
   /bin/ps -l
   ```

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be **root** to use these commands. For more information, refer to the online manual pages for these commands.

Shared Libraries

TotalView supports dynamically linked executables, that is, executables that are linked with shared libraries.

When you start TotalView with a dynamically linked executable, TotalView loads an additional set of symbols for the shared libraries, as indicated in the shell from which you started TotalView. To accomplish this, TotalView:

1 Runs a sample process and discards it.
2 Reads information from the process.
3 Reads the symbol table for each library.

When you create a process without starting it, and the process does not include shared libraries, the PC points to the entry point of the process, usually the **start** routine. If the process does include shared libraries, TotalView takes the following actions:

- Runs the dynamic loader (SunOS 5: **ld.so,** HP Tru64 UNIX**: /sbin/loader**, Linux: **/lib/ld-linux.so.?**, IRIX: **rld**).
- Sets the PC to point to the location after the invocation of the dynamic loader but before the invocation of C++ static constructors or the **main()** routine.

On HP-UX, *TotalView cannot stop the loading of shared libraries until after static constructors on shared library initialization routines have been run.*

When you attach to a process that uses shared libraries, TotalView takes the following actions:

- If you attached to the process after the dynamic loader ran, then TotalView loads the dynamic symbols for the shared library.
- If you attached to the process before it runs the dynamic loader, TotalView allows the process to run the dynamic loader to completion. Then, TotalView loads the dynamic symbols for the shared library.

If desired, you can suppress the recording and use of dynamic symbols for shared libraries by starting TotalView with the **–no_dynamic** option. Refer to Chapter 7, "*TotalView Command Syntax*," on page 241 for details on this TotalView startup option.

If a shared library has changed since you started a TotalView session, you can use the **Group > Rescan Library** command to reload library symbol tables. Be aware that only some systems such as AIX permit you to reload library information.

Changing Linkage Table Entries and LD_BIND_NOW

If you are executing a dynamically linked program, calls from the executable into a shared library are made using the *Procedure Linkage Table* (PLT). Each function in the dynamic library that is called by the main program has an entry in this table. Normally, the dynamic linker fills the PLT entries with code that calls the dynamic linker. This means that the first time that your code calls a function in a dynamic library, the runtime environment calls the dynamic linker. The linker will then modify the entry so that next time this function is called, it will not be involved.

This is not the behavior you want or expect when debugging a program because TotalView will do one of the following:

- Place you within the dynamic linker (which you don't want to see).
- Step over the function.

And, because the entry is altered, everything appears to work fine the next time you step into this function.

On most operating systems (except HP), you can correct this problem by setting the **LD_BIND_NOW** environment variable. For example:

```
setenv LD_BIND_NOW 1
```

This tells the dynamic linker that it should alter the PLT when the program starts executing rather than doing it when the program calls the function.

HP-UX does not have this (or an equivalent) variable. On HP systems, you can avoid this problem by using the **–B immediate** option the executable being debugged, or by invoking **chatr** with the **–B immediate** option. (See the **chatr** documentation for complete information on how to use this command.)

You will also have to enter **pxdb –s on**.

Using Shared Libraries on HP-UX

The dynamic library loader on HP-UX loads shared libraries into shared memory. Writing breakpoints into code sections loaded in shared memory can cause programs not under TotalView control to fail when they execute an unexpected breakpoint.

If you need to single-step or set breakpoints in shared libraries, you must set your application to load those libraries in private memory. This is done using HP's **pxdb** command.

```
pxdb -s on appname  (load shared libraries into
            private memory)
pxdb -s off appname  (load shared libraries into
            shared memory)
```

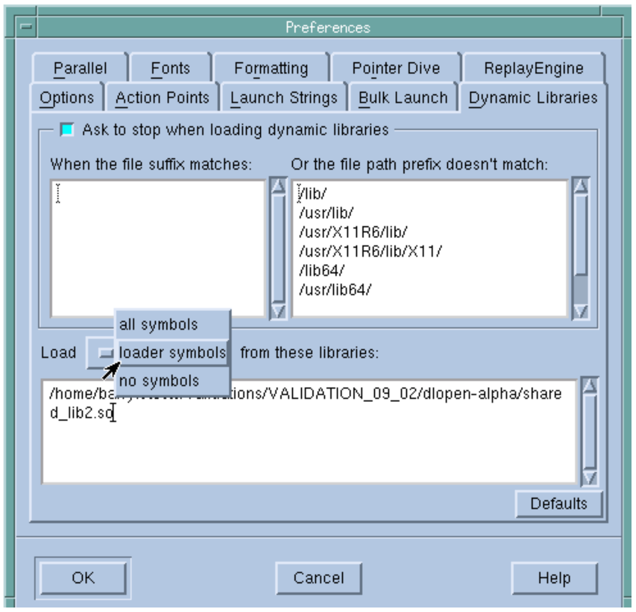
For 64-bit platforms, use **pxdb64** instead of **pxdb**. If the version of **pxdb64** supplied with HP's compilers does not work correctly, you may need to install an HP-supplied patch. You will find additional information in the *TotalView Release Notes*.

Debugging Your Program's Dynamically Loaded Libraries

TotalView automatically reads the symbols of shared libraries that are dynamically loaded into your program at runtime. These libraries are ones that are loaded using **dlopen** (or, on IBM AIX, **load** and **loadbind**).

TotalView automatically detects these calls, and then loads the symbol table from the newly loaded libraries and plants any enabled saved breakpoints for these libraries. TotalView then decides whether to ask you about stopping the process to plant breakpoints. You will set these characteristics by using the **Dynamic Libraries** page in the **File > Preferences** Dialog Box.

Figure 8: File > Preferences Dialog Box: Dynamic Libraries Page



Ask to stop when loading dynamic libraries

When the file suffix matches:

Or the file path prefix doesn't match:

```
/lib/
/usr/lib/
/usr/X11R6/lib/
/usr/X11R6/lib/X11/
/lib64/
/usr/lib64/
```

Load loader symbols from these libraries:

```
/home/ba[illegible]tions/VALIDATION_09_02/dlopen-alpha/share
d_lib2.so
```

TotalView decides according to the following rules:

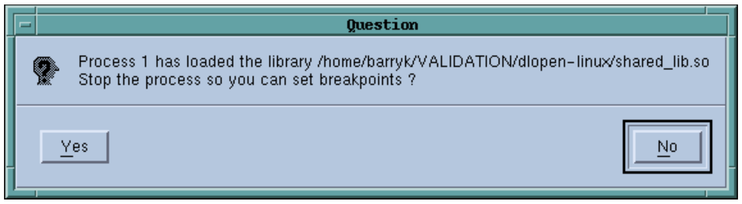
1. If either the **Load symbols from dynamic libraries** or **Ask to stop when loading dynamic libraries** preference is set to **false**, TotalView does not ask you about stopping.
2. If one or more of the strings in the **When the file suffix matches** preference list is a suffix of the full library name (including the path), TotalView asks you about stopping.
3. If one or more of the strings in the **When the file path prefix does not match** list is a prefix of the full library name (including the path), TotalView *does not* ask you about stopping.

4 If the newly loaded libraries have any saved breakpoints, TotalView does not ask you about stopping.
5 If none of the rules above apply, TotalView asks you about stopping.

If TotalView does not ask you about stopping the process, the process is continued.

If TotalView decides to ask you about stopping, it displays a dialog box, asking if it should stop the process so you can set breakpoints. To stop the process, answer **Yes**. (See Figure 9.)

Figure 9: Stop Process Question Dialog Box



Process 1 has loaded the library /home/barryk/VALIDATION/dlopen-linux/shared_lib.so
Stop the process so you can set breakpoints ?

To allow the process to continue executing, answer **No**. Stopping the process allows you to insert breakpoints in the newly loaded shared library.

Do either or both of the following to tell TotalView if it should ask:

- If you can set the **–ask_on_dlopen** command-line option to **true**, or you can set the **–no_ask_on_dlopen** option to false.
- Unset the **Load symbols from dynamic libraries** preference.

The following table lists paths where you are not asked if TotalView should stop the process:

Platform	Value	
HP Tru64 UNIX Alpha	/usr/shlib/ /usr/lib/cmplrs/cc/ /usr/local/lib/	/usr/ccs/lib/ /usr/lib/ /var/shlib/
HP-UX	/usr/lib/ /opt/langtools/lib/	/usr/lib/pa20_64 /opt/langtools/lib/pa20_64/
IBM AIX	/lib/ /usr/lpp/ /usr/dt/lib/	/usr/lib/ /usr/ccs/lib/ /tmp/
SGI IRIX	/lib/ /usr/local/lib/ /usr/lib32/ /lib64/ /usr/local/lib64	/usr/lib/ /lib32/ /usr/local/lib32/ /usr/lib64/
SUN Solaris 2.x	/lib/ /usr/ccs/lib/	/usr/lib/
Linux x86	/lib	/usr/lib
Linux Alpha	/lib	/usr/lib

The values you enter in the TotalView preference should be space-separated lists of the prefixes and suffixes to be used.

After starting TotalView, you can change these lists by using the **When the file suffix matches** and **And the file path prefix does not match** preferences.

Known Limitations

Dynamic library support has the following known limitations:

- TotalView does not deal correctly with parallel programs that call **dlopen** on different libraries in different processes. TotalView requires that the processes have a uniform address space, including all shared libraries.
- TotalView does not yet fully support unloading libraries (using **dlclose**) and then reloading them at a different address using **dlopen**.

Remapping Keys

On the SunOS 5 keyboard, you may need to remap the page-up and page-down keys to the prior and next **keysym** so that you can scroll TotalView windows with the page-up and page-down keys. To do so, add the following lines to your X Window System startup file:

```
# Remap F29/F35 to PgUp/PgDn
xmodmap -e 'keysym F29 = Prior'
xmodmap -e 'keysym F35 = Next'
```

Expression System

Depending on the target platform, TotalView supports:

- An interpreted expression system only
- Both an interpreted and a compiled expression system

Unless stated otherwise below, TotalView supports interpreted expressions only.

Expression System on HP Alpha Tru64 UNIX

On HP Tru64 UNIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembly language in expressions.

Expression System on IBM AIX

On IBM AIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembly language in expressions.

Some program functions called from the TotalView expression system on the Power architecture cannot have floating-point arguments that are passed by value. However, in functions with a variable number of arguments, floating-point arguments *can* be in the varying part of the argument list. For example, you can include floating-point arguments with calls to **printf**:

```
double d = 3.14159;
printf("d = %f\n", d);
```

Expression System on SGI IRIX

On IRIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

TotalView includes the SGI IRIX expression compiler. This feature does not use any MIPS-IV specific instructions. It does use MIPS-III instructions freely. It fully supports –**n32** and –**64** executables.

Due to limitations in dynamically allocating patch space, compiled expressions are disabled by default on SGI IRIX. To enable compiled expressions, use the **TV::compile_expressions** CLI variable to set the option to **true**. This variable tells TotalView to find or allocate patch space in your program for code fragments generated by the expression compiler.

If you enable compiled patches on SGI IRIX with a multiprocess program, you must use static patches. For example, if you link a static patch space into a program and run the program under TotalView control, TotalView should let you debug it. If you attach to a previously started MPI job, however, even static patches will not let the program run properly. If TotalView still fails to work properly with the static patch space, then you probably cannot use compiled patches with your program.

For general instructions on using patch space allocation controls with compiled expressions, see "*Allocating Patch Space for Compiled Expressions*" in Chapter 15 of the *TotalView Users Guide*.

Architectures

chapter 11

This chapter describes the architectures TotalView supports, including:

- "*AMD and Intel x86-64*" on page 287
- "*HP Alpha*" on page 291
- "HP PA-RISC" on page 293
- "*IBM Cell Broadband Engine*" on page 296
- "*Power Architectures*" on page 297
- "*Intel* IA-64" on page 301
- "*Intel x86*" on page 305 (Intel 80386, 80486 and Pentium processors)
- "SGI MIPS" on page 308
- "*Sun* SPARC" on page 313

AMD and Intel x86-64

This section describes AMD's 64-bit processors and the Intel EM64T processoers. It contains the following information:

- "*x86-64 General Registers*" on page 288
- "*x86-64 Floating-Point Registers*" on page 288
- "*x86-64* FPCR *Register*" on page 289
- "*x86-64* FPSR *Register*" on page 290
- "*x86-64* MXCSR *Register*" on page 291

The x86-64 can be programmed in either 32- or 64-bit mode. TotalView supports both. In 32-bit mode, the processor is identical to an x86, and the stack frame is identical to the x86. The information within this section describes 64-bit mode.

The AMD x86-64 *processor supports the* IEEE *floating-point format*.

x86-64 General Registers

TotalView displays the x86-64 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
RAX	General registers	$long	yes	yes	$rax
RDX		$long	yes	yes	$rdx
RCX		$long	yes	yes	$rcx
RBX		$long	yes	yes	$rbx
RSI		$long	yes	yes	$rsi
RDI		$long	yes	yes	$rdi
RBP		$long	yes	yes	$rbp
RSP		$long	yes	yes	$rsp
R8-R15		$long	yes	yes	$r8-$r15
RA	Selector registers	$int	no	no	$ra
SS		$int	no	no	$ss
DS		$int	no	no	$ds
ES		$int	no	no	$es
FS		$int	no	no	$fs
GS		$int	no	no	$gs
EFLAGS		$int	no	no	$eflags
RIP	Instruction pointer	$code[]	no	yes	$eip
FS_BASE		$long	yes	yes	$fs_base
GS_BASE		$long	yes	yes	$gs_base
TEMP		$long	no	no	$temp

x86-64 Floating-Point Registers

TotalView displays the x86-64 floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
ST0	ST(0)	$extended	yes	yes	$st0
ST1	ST(1)	$extended	yes	yes	$st1
ST2	ST(2)	$extended	yes	yes	$st2
ST3	ST(3)	$extended	yes	yes	$st3
ST4	ST(4)	$extended	yes	yes	$st4
ST5	ST(5)	$extended	yes	yes	$st5
ST6	ST(6)	$extended	yes	yes	$st6
ST7	ST(7)	$extended	yes	yes	$st7

Register	Description	Data Type	Edit	Dive	Specify in Expression
FPCR	Floating-point control register	$int	yes	no	$fpcr
FPSR	Floating-point status register	$int	no	no	$fpsr
FPTAG	Tag word	$int	no	no	$fptag
FPOP	Floating-point operation	$int	no	no	$fpop
FPI	Instruction address	$int	no	no	$fpi
FPD	Data address	$int	no	no	$fpd
MXCSR	SSE status and control	$int	yes	no	$mxcsr
MXCSR_ MASK	MXCSR mask	$int	no	no	$mxcsr_ mask
XMM0_L ... XMM7_L	Streaming SIMD Extension: left half	$long	yes	yes	$xmm0_l ... $xmm7_l
XMM0_H ... XMM7_H	Streaming SIMD Extension: right half	$long	yes	yes	$xmm0_h ... $xmm7_h
XMM8_L ... XMM15_L	Streaming SIMD Extension: left half	$long	yes	yes	$xmm8_l ... $xmm15_l
XMM8_H ... XMM15_H	Streaming SIMD Extension: right half	$long	yes	yes	$xmm8_h ... $xmm15_h

The x86-64 has 16 128-bit registers that are used by SSE and SSE2 instructions. TotalView displays these as 32 64-bit registers. These registers can be used in the following ways: 16 bytes, 8 words, 2 longs, 4 floating point, 2 double, or a single 128-bit value. TotalView shows each of these hardware registers as two $long registers. To change the type, dive and then edit the type in the data window to be an array of the type you wish. For example, cast it to "$char[16]", "$float[4], and so on.

x86-64 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in the next table.

Value	Bit Setting	Meaning
RC=RN	0x0000	To nearest rounding mode
RC=R-	0x2000	Toward negative infinity rounding mode
RC=R+	0x4000	Toward positive infinity rounding mode
RC=RZ	0x6000	Toward zero rounding mode
PC=SGL	0x0000	Single-precision rounding
PC=DBL	0x0080	Double-precision rounding
PC=EXT	0x00c0	Extended-precision rounding
EM=PM	0x0020	Precision exception enable

Value	Bit Setting	Meaning
EM=UM	0x0010	Underflow exception enable
EM=OM	0x0008	Overflow exception enable
EM=ZM	0x0004	Zero-divide exception enable
EM=DM	0x0002	Denormalized operand exception enable
EM=IM	0x0001	Invalid operation exception enable

Using the x86-64 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPCR to include **0x0004** so that TotalView traps the "divide-by-zero" bit. The string displayed next to the FPCR register should now include **EM=(ZM).** Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See "*Handling Signals*" in Chapter 3 of the *TotalView Users Guide* for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the **EF=(ZE)** bit in the FPSR.

x86-64 FPSR Register

The bit settings of the x86-64 FPSR register are outlined in the following table.

Value	Bit Setting	Meaning
TOP=<*i*>	0x3800	Register <*i*> is top of FPU stack
B	0x8000	FPU busy
C0	0x0100	Condition bit 0
C1	0x0200	Condition bit 1
C2	0x0400	Condition bit 2
C3	0x4000	Condition bit 3
ES	0x0080	Exception summary status
SF	0x0040	Stack fault
EF=PE	0x0020	Precision exception
EF=UE	0x0010	Underflow exception
EF=OE	0x0008	Overflow exception
EF=ZE	0x0004	Zero divide exception
EF=DE	0x0002	Denormalized operand exception
EF=IE	0x0001	Invalid operation exception

x86-64 MXCSR Register

This register contains control and status information for the SSE registers. Some of the bits in this register are editable. You cannot dive in these values.

The bit settings of the x86-64 MXCSR register are outlined in the following table.

Value	Bit Setting	Meaning
FZ	0x8000	Flush to zero
RC=RN	0x0000	To nearest rounding mode
RC=R-	0x2000	Toward negative infinity rounding mode
RC=R+	0x4000	Toward positive infinity rounding mode
RC=RZ	0x6000	Toward zero rounding mode
EM=PM	0x1000	Precision mask
EM=UM	0x0800	Underflow mask
EM=OM	0x0400	Overflow mask
EM=ZM	0x0200	Divide-by-zero mask
EM=DM	0x0100	Denormal mask
EM=IM	0x0080	Invalid operation mask
DAZ	0x0040	Denormals are zeros
EF=PE	0x0020	Precision flag
EF=UE	0x0010	Underflow flag
EF=OE	0x0008	Overflow flag
EF=ZE	0x0004	Divide-by-zero flag
EF=DE	0x0002	Denormal flag
EF=IE	0x0001	Invalid operation flag

HP Alpha

This section contains the following information:

- Alpha General Registers
- Alpha Floating-Point Registers
- Alpha FPCR Register

The Alpha processor supports the IEEE *floating-point format.*

Alpha General Registers

TotalView displays the Alpha general registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each general register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
V0	Function value register	$long	yes	yes	$v0
T0 – T7	Conventional scratch registers	$long	yes	yes	$t0 – $t7
S0 – S5	Conventional saved registers	$long	yes	yes	$s0 – $s5
S6	Stack frame base register	$long	yes	yes	$s6
A0 – A5	Argument registers	$long	yes	yes	$a0 – $a5
T8 – T11	Conventional scratch registers	$long	yes	yes	$t8 – $t11
RA	Return Address register	$long	yes	yes	$ra
T12	Procedure value register	$long	yes	yes	$t12
AT	Volatile scratch register	$long	yes	yes	$at
GP	Global pointer register	$long	yes	yes	$gp
SP	Stack pointer	$long	yes	yes	$sp
ZERO	ReadAsZero/Sink register	$long	no	yes	$zero
PC	Program counter	$code[]	no	yes	$pc
FP	Frame pointer. The Frame Pointer is a software register that TotalView maintains; it is not an actual hardware register. TotalView computes the value of FP as part of the stack backtrace.	$long	no	yes	$fp

Alpha Floating-Point Registers

TotalView displays the Alpha floating-point registers in the Stack Frame Pane of the Process Window. Here is a table that describes how TotalView treats each floating-point register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F1	Floating-point registers (*f* registers), used singly	$double	yes	yes	$f0 – $f1
F2 – F9	Conventional saved registers	$double	yes	yes	$f2 – $f9
F10 – F15	Conventional scratch registers	$double	yes	yes	$f10 – $f15
F16 – F21	Argument registers	$double	yes	yes	$f16 – $f21
F22 – F30	Conventional scratch registers	$double	yes	yes	$f22 – $f30
F31	ReadAsZero/Sink register	$double	yes	yes	$f31
FPCR	Floating-point control register	$long	yes	no	$fpcr

Alpha FPCR Register

For your convenience, TotalView interprets the bit settings of the Alpha FPCR register. You can edit the value of the FPCR and set it to any of the bit settings outlined in the following table.

Value	Bit Setting	Meaning
SUM	0x8000000000000000	Summary bit
DYN=CHOP	0x0000000000000000	Rounding mode — Chopped rounding mode
DYN=MINF	0x0400000000000000	Rounding mode — Negative infinity
DYN=NORM	0x0800000000000000	Rounding mode — Normal rounding
DYN=PINF	0x0c00000000000000	Rounding mode — Positive infinity
IOV	0x0200000000000000	Integer overflow
INE	0x0100000000000000	Inexact result
UNF	0x0080000000000000	Underflow
OVF	0x0040000000000000	Overflow
DZE	0x0020000000000000	Division by zero
INV	0x0010000000000000	Invalid operation

HP PA-RISC

This section contains the following information:

- PA-RISC General Registers
- PA-RISC Process Status Word
- PA-RISC Floating-Point Registers
- PA-RISC Floating-Point Format

PA-RISC General Registers

TotalView displays the PA-RISC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register and the actions you take with them.

Register	Description	Data Type	Edit	Dive	Specify in Expression
r0	Always contains zero	$long	no	no	$r0
r1-r31	General registers	$long	yes	yes	$r1-$r31
pc	Current instruction pointer	$long	yes	yes	$pc
nxtpc	Next instruction pointer	$long	yes	yes	$nxtpc
pcs	Current instruction space	$long	no	no	$pcs
nxtpcs	Next instruction space	$long	no	no	$nxtpcs

Register	Description	Data Type	Edit	Dive	Specify in Expression
psw	Processor status word	$long	yes	no	$psw
sar	Shift amount register	$long	yes	no	$sar
sr0-sr7	Space registers	$long	no	no	$sr0-$sr7
recov	Recovery counter	$long	no	no	$recov
pid1-pid8	Protection IDs	$long	no	no	$pid1-$pid8
ccr	Coprocessor configuration	$long	no	no	$ccr
scr	SFU configuration register	$long	no	no	$scr
eiem	External interrupt enable mask	$long	no	no	$eiem
iir	Interrupt instruction	$long	no	no	$iir
isr	Interrupt space	$long	no	no	$isr
ior	Interrupt offset	$long	no	no	$ior
cr24-cr26	Temporary registers	$long	no	no	$cr24-$cr26
tp	Thread pointer	$long	yes	yes	$tp

PA-RISC Process Status Word

For your convenience, TotalView interprets the bit settings of the PA-RISC Processor Status Word. You can edit the value of this word and set some of the bits listed in the following table.

Value	Bit Setting	Meaning
W	0x0000000008000000	64-bit addressing enable
E	0x0000000004000000	Little-endian enable
S	0x0000000002000000	Secure interval timer
T	0x0000000001000000	Taken branch flag
H	0x0000000000800000	Higher-privilege transfer trap enable
L	0x0000000000400000	Lower-privilege transfer trap enable
N	0x0000000000200000	Nullify current instruction
X	0x0000000000100000	Data memory break disable
B	0x0000000000080000	Taken branch flag
C	0x0000000000040000	Code address translation enable
V	0x0000000000020000	Divide step correction
M	0x0000000000010000	High-priority machine check mask
O	0x0000000000000080	Ordered references
F	0x0000000000000020	Performance monitor interrupt unmask
R	0x0000000000000010	Recovery counter enable
Q	0x0000000000000008	Interrupt state collection enable
P	0x0000000000000004	Protection identifier validation enable
D	0x0000000000000002	Data address translation enable
I	0x0000000000000001	External interrupt unmask
C/B	0x000000FF0000FF00	Carry/borrow bits

PA-RISC Floating-Point Registers

The PA-RISC has 32 floating-point registers. The first four are used for status and exception registers. The rest can be addressed as 64-bit doubles, as two 32-bit floats in the right and left sides of the register, or even-odd pairs of registers as 128-bit extended floats.

Register	Description	Data Type	Edit	Dive	Specify in Expression
status	Status register	$int	no	no	$status
er1-er7	Exception registers	$int	no	no	$er1-$er7
fr4-fr31	Double floating-point registers	$double	yes	yes	$fr4-$fr31
fr4l-fr31l	Left half floating-point registers	$float	yes	yes	$fr4l-$fr31l
fr4r-fr31r	Right half floating-point registers	$float	yes	yes	$fr4r-$fr31r
fr4/fr5-fr30/fr31	Extended floating-point register pairs	$extended	yes	yes	$fr4_fr5-$fr30_fr31

The floating-point status word controls the arithmetic rounding mode, enables user-level traps, enables floating-point exceptions, and indicates the results of comparisons.

Type	Value	Meaning
Rounding Mode	0	Round to nearest
	1	Round toward zero
	2	Round toward +infinity
	3	Round toward –infinity
Exception Enable and Exception Flag Bits	V	Invalid operation
	Z	Division by zero
	O	Overflow
	U	Underflow
	I	Inexact result
Comparison Fields	C	Compare bit; contains the result of the most recent queued compare instruction.
	CQ	Compare queue; contains the result of the second-most recent queued compare through the twelfth-most recent queued compare. Each queued compare instruction shifts the CQ field right one bit and copies the C bit into the left-most position. This field occupies the same bits as the CA field and is undefined after a targeted compare.
	CA	Compare array; an array of seven compare bits, each of which contains the result of the most recent compare instruction targeting that bit. This field occupies the same bits as the CQ field and is undefined after a queued compare.
Other Flags	T	Delayed trap
	D	Denormalized as zero

PA-RISC Floating-Point Format

The PA-RISC processor supports the IEEE floating-point format.

IBM Cell Broadband Engine

TotalView displays Cell general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each register, and the actions you can take with it.

Register	Description	Data Type	Edit	Dive	Specify in Expression
R0–R127	General Registers	union $spu_vec128	yes	yes	$r0–$r127
FPSCR	Floating point status and control register	$int128	yes	yes	$fpscr
OBJECT_ID	Context ID created by application; used later by profiling tools to uniquely identify context	$long_long	no	yes	$object_id
ID	Instruction dispatch register	$int	no	yes	$id
PC	Program counter	$code[272]	yes	yes	$pc
SP	Stack pointer	$int	yes	yes	$sp
FP	Floating point register	$int	no	yes	$fp
LR	Link register	$code[657]	yes	yes	$lr
SRR0	Interrupt return address register	$int	yes	no	$srr0
LSLR	Local storage limit register	$int	no	yes	$lslr
DECR	SPU decrementer	$int	yes	yes	$decr
DECR_STATUS	Decrementer status	$int	yes	yes	$decr_status
CNTL	Read access to the status register	$int	yes	yes	$cntl
EVENT_MASK	Event mask for SPU interrupts	$int	no	yes	$event_mask
EVENT_STATUS	Number of pending SPU events	$int	no	yes	$event_status
IBOX	Second SPU to CPU communication mailbox	$int	no	yes	$ibox
MBOX	First SPU to CPU communication mailbox	$int	no	yes	$mbox
WBOX	CPU to SPU communication mailbox	$int128	no	yes	$wbox
IBOX_STAT	Length of the IBOX queue	$int	no	yes	$ibox_stat
MBOX_STAT	Length of the MBOX queue	$int	no	yes	$mbox_stat
WBOX_STAT	Length of the WBOX queue	$int	no	yes	$wbox_stat
SIGNAL1	Signal notification channel	$int	no	yes	$signal1
SIGNAL1_TYPE	Behavior of signal1 notification channel	$int	no	yes	$signal1_type
SIGNAL2	Signal notification channel	$int	no	yes	$signal2
SIGNAL2_TYPE	Behavior of signal1 notification channel	$int	no	yes	$signal2_type
MFC	Access to the MFC DMA area	$long_long	no	yes	$mfg.

Power Architectures

This section contains the following information:

- Power General Registers
- Power MSR Register
- Power Floating-Point Registers
- Power FPSCR Register
- Using the Power FPSCR Register

The Power architecture supports the IEEE *floating-point format*.

Power General Registers

TotalView displays Power general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
R0	General register 0	$int	yes	yes	$r0
SP	Stack pointer	$int	yes	yes	$sp
RTOC	TOC pointer	$int	yes	yes	$rtoc
R3 – R31	General registers 3 – 31	$int	yes	yes	$r3 – $r31
INUM		$int	yes	no	$inum
PC	Program counter	$code[]	no	yes	$pc
SRR1	Machine status save/ restore register	$int	yes	no	$srr1
LR	Link register	$code	yes	no	$lr
CTR	Counter register	$int	yes	no	$ctr
CR	Condition register (see below)	$int	yes	no	$cr
XER	Integer exception register (see below)	$int	yes	no	$xer
DAR	Data address register	$int	yes	no	$dar
MQ	MQ register	$int	yes	no	$mq
MSR	Machine state register	$int	yes	no	$msr
SEG0 – SEG9	Segment registers 0 – 9	$int	yes	no	$seg0 – $seg9
SG10 – SG15	Segment registers 10 –15	$int	yes	no	$sg10 – $sg15
SCNT	SS_COUNT	$int	yes	no	$scnt
SAD1	SS_ADDR 1	$int	yes	no	$sad1
SAD2	SS_ADDR 2	$int	yes	no	$sad2
SCD1	SS_CODE 1	$int	yes	no	$scd1

Register	Description	Data Type	Edit	Dive	Specify in Expression
SCD2	SS_CODE 2	$int	yes	no	$scd2
TID		$int	yes	no	

CR Register

TotalView writes information for each of the eight condition sets, appending a a >, <, or = symbol. For example, if the summary overflow (0x1) bit is set, TotalView might display the following:

```
0x22424444 (574768196) (0=,1=,2>,3=,4>,5>,6>,7>)
```

XER Register

Depending upon what was set, TotalView can display up to five kinds of information, as follows:

STD:0x%02x	The string terminator character (bits 25-31)
SL:%d	The string length field (bits 16-23)
S0	Displayed if the summary overflow bit is set (bit 0)
OV	Displayed if the overflow bit is set (bit 1)
CA	Displayed if the carry bit is set (bit 2)

For example:

```
0x20000002 (536870914) (STD:0x00,SL:2,CA)
```

Power MSR Register

For your convenience, TotalView interprets the bit settings of the Power MSR register. You can edit the value of the MSR and set it to any of the bit settings outlined in the following table.

Value	Bit Setting	Meaning
0x8000000000000000	SF	Sixty-four bit mode
0x0000000000040000	POW	Power management enable
0x0000000000020000	TGPR	Temporary GPR mapping
0x0000000000010000	ILE	Exception little-endian mode
0x0000000000008000	EE	External interrupt enable
0x0000000000004000	PR	Privilege level
0x0000000000002000	FP	Floating-point available
0x0000000000001000	ME	Machine check enable
0x0000000000000800	FE0	Floating-point exception mode 0
0x0000000000000400	SE	Single-step trace enable
0x0000000000000200	BE	Branch trace enable
0x0000000000000100	FE1	Floating-point exception mode 1
0x0000000000000040	IP	Exception prefix
0x0000000000000020	IR	Instruction address translation
0x0000000000000010	DR	Data address translation
0x0000000000000002	RI	Recoverable exception
0x0000000000000001	LE	Little-endian mode enable

Power Floating-Point Registers

TotalView displays the Power floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F31	Floating-point registers 0 – 31	$double	yes	yes	$f0 – $f31
FPSCR	Floating-point status register	$int	yes	no	$fpscr
FPSCR2	Floating-point status register 2	$int	yes	no	$fpscr2

Power FPSCR Register

For your convenience, TotalView interprets the bit settings of the Power FPSCR register. You can edit the value of the FPSCR and set it to any of the bit settings outlined in the following table.

Value	Bit Setting	Meaning
0x80000000	FX	Floating-point exception summary
0x40000000	FEX	Floating-point enabled exception summary
0x20000000	VX	Floating-point invalid operation exception summary
0x10000000	OX	Floating-point overflow exception
0x08000000	UX	Floating-point underflow exception
0x04000000	ZX	Floating-point zero divide exception
0x02000000	XX	Floating-point inexact exception
0x01000000	VXSNAN	Floating-point invalid operation exception for SNaN
0x00800000	VXISI	Floating-point invalid operation exception: $\infty - \infty$
0x00400000	VXIDI	Floating-point invalid operation exception: ∞ / ∞
0x00200000	VXZDZ	Floating-point invalid operation exception: 0 / 0
0x00100000	VXIMZ	Floating-point invalid operation exception: $\infty * \infty$
0x00080000	VXVC	Floating-point invalid operation exception: invalid compare
0x00040000	FR	Floating-point fraction rounded
0x00020000	FI	Floating-point fraction inexact
0x00010000	FPRF=(C)	Floating-point result class descriptor
0x00008000	FPRF=(L)	Floating-point less than or negative
0x00004000	FPRF=(G)	Floating-point greater than or positive
0x00002000	FPRF=(E)	Floating-point equal or zero
0x00001000	FPRF=(U)	Floating-point unordered or NaN

Value	Bit Setting	Meaning
0x00011000	FPRF=(QNAN)	Quiet NaN; alias for FPRF=(C+U)
0x00009000	FPRF=(-INF)	-Infinity; alias for FPRF=(L+U)
0x00008000	FPRF=(-NORM)	-Normalized number; alias for FPRF=(L)
0x00018000	FPRF=(-DENORM)	-Denormalized number; alias for FPRF=(C+L)
0x00012000	FPRF=(-ZERO)	-Zero; alias for FPRF=(C+E)
0x00002000	FPRF=(+ZERO)	+Zero; alias for FPRF=(E)
0x00014000	FPRF=(+DENORM)	+Denormalized number; alias for FPRF=(C+G)
0x00004000	FPRF=(+NORM)	+Normalized number; alias for FPRF=(G)
0x00005000	FPRF=(+INF)	+Infinity; alias for FPRF=(G+U)
0x00000400	VXSOFT	Floating-point invalid operation exception: software request
0x00000200	VXSQRT	Floating-point invalid operation exception: square root
0x00000100	VXCVI	Floating-point invalid operation exception: invalid integer convert
0x00000080	VE	Floating-point invalid operation exception enable
0x00000040	OE	Floating-point overflow exception enable
0x00000020	UE	Floating-point underflow exception enable
0x00000010	ZE	Floating-point zero divide exception enable
0x00000008	XE	Floating-point inexact exception enable
0x00000004	NI	Floating-point non-IEEE mode enable
0x00000000	RN=NEAR	Round to nearest
0x00000001	RN=ZERO	Round toward zero
0x00000002	RN=PINF	Round toward +infinity
0x00000003	RN=NINF	Round toward –infinity

Using the Power FPSCR Register

On AIX, if you compile your program to catch floating-point exceptions (IBM compiler –**qflttrap** option), you can change the value of the FPSCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSCR to include **0x10** so that TotalView traps the "divide by zero" exception. The string displayed next to the FPSR register should now include **ZE**. Now, when your program divides by zero, it receives a **SIGTRAP** signal, which will be caught by TotalView. See "*Handling Signals*" in Chapter 3 of the *TotalView Users Guide* for more information. If you did not set the bit for trapping divide by zero or you did not compile to catch floating-point exceptions, your program would not stop and the processor would set the **ZX** bit.

Intel IA-64

This section contains the following information:

- Intel IA-64 General Registers
- "*IA-64 Processor Status Register Fields* (PSR)" on page 302
- "*Current Frame Marker Register Fields* (CFM)" on page 303
- "*Register Stack Configuration Register Fields* (RSC)" on page 303
- "*Previous Function State Register Fields* (PFS)" on page 303
- "*Floating Point Registers*" on page 304
- "*Floating Point Status Register Fields*" on page 304

The Cray XT3 front end runs on this chip.

Intel IA-64 General Registers

TotalView displays the IA-64 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each.

The descriptions in this section are taken (almost verbatim) from the "Intel Itanium Architecture Software Developer's Manual. Volume 1: Application Architecture". This was revision 2.0, printed in December 2001.

Register	Description	Data Type	Edit	Dive	in expression
r0	register 0	$long	N	Y	$r0
r1	global pointer	$long	N	Y	$r1
r2-r31	static general registers	$long	Y	Y	$r2-$r31
r31-r127	stacked general registers (all may not be valid)	$long	Y	Y	$r32-$r127
b0-b7	branch registers	$code[]	Y	Y	$b0-$b7
ip	instruction pointer	$code[]	N	Y	$ip
cfm	current frame marker	$long	Y	Y	$cfm
psr	processor status register	$long	Y	Y	$psr
rsc	register stack configuration register (AR 16)	$long	Y (N on HP-UX)	Y	$rsc
bsp	rse backing store pointer (AR 17)	$long	Y	Y	$bsp
bspstore	rse backing store pointer for memory stores (AR 18)	$long	N	Y	$bspstore
rnat	rse NAT collection register (AR 19)	$long	Y	Y	$rnat
ccv	compare and exchange value register (AR 32)	$long	Y	Y	$ccv

Register	Description	Data Type	Edit	Dive	in expression
unat	user NAT collection register (AR 36)	$long	Y	Y	$unat
fpsr	floating point status register (AR 40)	$long	Y	Y	$fpsr
pfs	previous function state (AR 64)	$long	Y	Y	$pfs
lc	loop count register (AR 65)	$long	Y	Y	$lc
ec	epilog count register (AR 66)	$long	Y	Y	$ec
pr	predication registers (packed)	$long	Y	Y	$pr
nat	nat registers (packed)	$long	Y	Y	$nat

All general registers r32-r127 may not be valid in a given stack frame.

IA-64 Processor Status Register Fields (PSR)

These fields control memory access alignment, byte-ordering, and user-configured performance monitors. It also records the modification state of floating-point registers.

Bit	Field	Meaning
1	be	big-endian enable
2	up	user performance monitor enable
3	ac	alignment check
4	mfl	lower (f2-f31) floating point registers written
5	mfh	upper (f32-f127) floating point registers written
13	ic	interruption collection
14	i	interrupt bit
15	pk	protection key enable
17	dt	data address translation
18	dfl	disabled lower floating point register set
19	dfh	disabled upper floating point register set
20	sp	secure performance monitors
21	pp	privileged performance monitor enable
22	di	disable instruction set transition
23	si	secure interval timer
24	db	debug breakpoint fault
25	lp	lower privilege transfer trap
26	tb	taken branch trap
27	rt	register stack translation
33:32	cpl	current privilege level
34	is	instruction set
35	mc	machine check abort mask
36	it	instruction address translation
37	id	instruction debug fault disable
38	da	disable data access and dirty-bit faults

Bit	Field	Meaning
39	dd	data debug fault disable
40	ss	single step enable
42:41	ri	restart instruction
43	ed	exception deferral
44	bn	register bank
45	ia	disable instruction access-bit faults

Current Frame Marker Register Fields (CFM)

Each general register stack frame is associated with a frame marker. The frame maker describes the state of the general register stack. The Current Frame Marker (CFM) holds the state of the current stack frame.

Bit Range	Field	Meaning
6:0	sof	Size of frame
13:7	sol	Size of locals portion of stack frame
17:14	sor	Size of rotating portion of stack frame (number of rotating registers is sor*8
24:18	rrb.gr	Register rename base for general registers
31:25	rrb.fr	Register rename base for float registers
37:32	rrb.pr	Register rename base for predicate registers

Register Stack Configuration Register Fields (RSC)

The Register Stack Configuration (RSC) Register is a 64-bit register used to control the operation of the Register Stack Engine (RSE).

Bit Range	Field	Meaning
1:0 00	mode	enforced lazy
1:0 01		load intensive
1:0 10		store intensive
1:0 11		eager
3:2	pl	RSE privilege level
4	be	RSE endian mode (0=little endian, 1=big endian)
29:16	loadrs	RSE load distance to tear point

Previous Function State Register Fields (PFS)

The Previous Function State register (PFS) contains multiple fields: Previous Frame Marker (pfm), Previous Epilog Count (pec), and Previous Privilege Level (ppl). These values are copied automatically on a call from the CFM register, Epilog Count Register (EC), and PSR.cpl (Current Privilege Level in the Processor Status Register) to accelerate procedure calling.

Value	Bit Setting	Meaning
37:0	pfm	previous frame marker
57:52	pec	previous epilog count
63:62	ppl	previous privilege level

Floating Point Registers

The IA-64 contains 128 floating-point registers. The first two are read only.

Register	Description	Data Type	Edit	Dive	Specify in Expression
f0	float register 0	$long	N	Y	$f0
f1	float register 1	$long	N	Y	$f1
f2-f31	lower float registers	$long	Y	Y	$f2-$f31
f32-f127	upper float registers	$long	Y	Y	$f32-$f127

Floating Point Status Register Fields

The Floating-Point Status Register (FPSR) contains the dynamic control and status information for floating-point operations. There is one main set of control and status information and three alternate sets.

Field	Bits	Meaning
traps.vd	0	Invalid Operation Floating-Point Exception fault disabled
traps.dd	1	Denormal/Unnormal Operating Floating-Point Exception fault disabled
traps.zd	2	Zero Divide Floating-Point Exception trap disabled
traps.od	3	Overflow Floating-Point Exception trap disabled
traps.ud	4	Underflow Floating-Point Exception trap disabled
traps.id	5	Inexact Floating-Point Exception trap disabled
sfo	18:6	main status field
sf1	31:19	alternate status field 1
sf2	44:32	alternate status field 2
sf3	57:45	alternate status field 3

Here is a description of the FPSR status field descriptions.

Bits	Field	meaning
0	ftz	flush-to-zero mode
1	wre	widest range exponent
3:2	pc	precision control
5:4	rc	rounding control
6	td	traps disabled
7	v	invalid operation
8	d	denormal/unnormal operand
9	z	zero divide
10	o	overflow
11	u	underflow
12	i	inexact

Intel x86

This section contains the following information:

- "*Intel x86 General Registers*" on page 305
- "*Intel x86 Floating-Point Registers*" on page 306
- "*Intel x86 FPCR Register*" on page 306
- "*Intel x86 FPSR Register*" on page 307
- "*Intel x86 MXCSR Register*" on page 308

The Intel x86 processor supports the IEEE floating-point format.

Intel x86 General Registers

TotalView displays the Intel x86 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
EAX	General registers	$long	yes	yes	$eax
ECX		$long	yes	yes	$ecx
EDX		$long	yes	yes	$edx
EBX		$long	yes	yes	$ebx
EBP		$long	yes	yes	$ebp
ESP		$long	yes	yes	$esp
ESI		$long	yes	yes	$esi
EDI		$long	yes	yes	$edi
CS	Selector registers	$int	no	no	$cs
SS		$int	no	no	$ss
DS		$int	no	no	$ds
ES		$int	no	no	$es
FS		$int	no	no	$fs
GS		$int	no	no	$gs
EFLAGS		$int	no	no	$eflags
EIP	Instruction pointer	$code[]	no	yes	$eip
FAULT		$long	no	no	$fault
TEMP		$long	no	no	$temp
INUM		$long	no	no	$inum
ECODE		$long	no	no	$ecode

Intel x86 Floating-Point Registers

TotalView displays the x86 floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
ST0	ST(0)	$extended	yes	yes	$st0
ST1	ST(1)	$extended	yes	yes	$st1
ST2	ST(2)	$extended	yes	yes	$st2
ST3	ST(3)	$extended	yes	yes	$st3
ST4	ST(4)	$extended	yes	yes	$st4
ST5	ST(5)	$extended	yes	yes	$st5
ST6	ST(6)	$extended	yes	yes	$st6
ST7	ST(7)	$extended	yes	yes	$st7
FPCR	Floating-point control register	$int	yes	no	$fpcr
FPSR	Floating-point status register	$int	no	no	$fpsr
FPTAG	Tag word	$int	no	no	$fptag
FPIOFF	Instruction offset	$int	no	no	$fpioff
FPISEL	Instruction selector	$int	no	no	$fpisel
FPDOFF	Data offset	$int	no	no	$fpdoff
FPDSEL	Data selector	$int	no	no	$fpdsel
MXCSR	SSE status and control	$int	yes	no	$mxcsv
MXCSR_MASK	MXCSR mask	$int	no	no	$mxcsr_mask
XMM0_L ... XMM7_L	Streaming SIMD Extension: left half	$long long	yes	yes	$xmm0_l ... $xmm7_l
XMM0_H ... XMM7_H	Streaming SIMD Extension: right half	$long long	yes	yes	$xmm0_h ... $xmm7_h

The Pentium III and 4 have 8 128-bit registers that are used by SSE and SSE2 instructions. TotalView displays these as 16 64-bit registers. These registers can be used in the following ways: 16 bytes, 8 words, 2 long longs, 4 floating point, 2 double, or a single 128-bit value. TotalView shows each of these hardware registers as two $long long registers. To change the type, dive and then edit the type in the data window to be an array of the type you wish. For example, cast it to "$char[16]", "$float[4], and so on.

Intel x86 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in the next table.

Value	Bit Setting	Meaning
RC=RN	0x0000	To nearest rounding mode
RC=R-	0x2000	Toward negative infinity rounding mode
RC=R+	0x4000	Toward positive infinity rounding mode
RC=RZ	0x6000	Toward zero rounding mode
PC=SGL	0x0000	Single-precision rounding
PC=DBL	0x0080	Double-precision rounding
PC=EXT	0x00c0	Extended-precision rounding
EM=PM	0x0020	Precision exception enable
EM=UM	0x0010	Underflow exception enable
EM=OM	0x0008	Overflow exception enable
EM=ZM	0x0004	Zero-divide exception enable
EM=DM	0x0002	Denormalized operand exception enable
EM=IM	0x0001	Invalid operation exception enable

Using the Intel x86 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPCR to include **0x0004** so that TotalView traps the "divide-by-zero" bit. The string displayed next to the FPCR register should now include **EM=(ZM).** Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See "*Handling Signals*" in Chapter 3 of the *TotalView Users Guide* for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the **EF=(ZE)** bit in the FPSR.

Intel x86 FPSR Register

The bit settings of the Intel x86 FPSR register are outlined in the following table.

Value	Bit Setting	Meaning
TOP=<*i*>	0x3800	Register <*i*> is top of FPU stack
B	0x8000	FPU busy
C0	0x0100	Condition bit 0
C1	0x0200	Condition bit 1
C2	0x0400	Condition bit 2
C3	0x4000	Condition bit 3
ES	0x0080	Exception summary status
SF	0x0040	Stack fault
EF=PE	0x0020	Precision exception
EF=UE	0x0010	Underflow exception
EF=OE	0x0008	Overflow exception
EF=ZE	0x0004	Zero divide exception

Value	Bit Setting	Meaning
EF=DE	0x0002	Denormalized operand exception
EF=IE	0x0001	Invalid operation exception

Intel x86 MXCSR Register

This register contains control and status information for the SSE registers. Some of the bits in this register are editable. You cannot dive in these values.

The bit settings of the Intel x86 MXCSR register are outlined in the following table.

Value	Bit Setting	Meaning
FZ	0x8000	Flush to zero
RC=RN	0x0000	To nearest rounding mode
RC=R-	0x2000	Toward negative infinity rounding mode
RC=R+	0x4000	Toward positive infinity rounding mode
RC=RZ	0x6000	Toward zero rounding mode
EM=PM	0x1000	Precision mask
EM=UM	0x0800	Underflow mask
EM=OM	0x0400	Overflow mask
EM=ZM	0x0200	Divide-by-zero mask
EM=DM	0x0100	Denormal mask
EM=IM	0x0080	Invalid operation mask
DAZ	0x0040	Denormals are zeros
EF=PE	0x0020	Precision flag
EF=UE	0x0010	Underflow flag
EF=OE	0x0008	Overflow flag
EF=ZE	0x0004	Divide-by-zero flag
EF=DE	0x0002	Denormal flag
EF=IE	0x0001	Invalid operation flag

SGI MIPS

This section contains the following information:

- MIPS General Registers
- MIPS SR Register
- MIPS Floating-Point Registers
- MIPS FCSR Register
- Using the MIPS FCSR Register
- MIPS Delay Slot Instructions

The MIPS *processor supports the* IEEE *floating-point format.*

MIPS General Registers

TotalView displays the MIPS general-purpose registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Programs compiled with **–64** or **–n32** have 64-bit registers. TotalView uses **$long** for **–64** compiled programs and **$long long** for **–n32** compiled programs.

Register	Description	Data Type	Edit	Dive	Specify in Expression
ZERO	Always has the value 0	$long	no	no	$zero
AT	Reserved for the assembler	$long	yes	yes	$at
V0 – V1	Function value registers	$long	yes	yes	$v0 – $v1
A0 – A7	Argument registers	$long	yes	yes	$a0 – $a7
T0 – T3	Temporary registers	$long	yes	yes	$t0 – $t3
S0 – S7	Saved registers	$long	yes	yes	$s0 – $s7
T8 – T9	Temporary registers	$long	yes	yes	$t8 – $t9
K0 – K1	Reserved for the operating system	$long	yes	yes	$k1 – $k2
GP	Global pointer	$long	yes	yes	$gp
SP	Stack pointer	$long	yes	yes	$sp
S8	Hardware frame pointer	$long	yes	yes	$s8
RA	Return address register	$code[]	no	yes	$ra
MDLO	Multiply/Divide special register, holds least-significant bits of multiply, quotient of divide	$long	yes	yes	$mdlo
MDHI	Multiply/Divide special register, holds most-significant bits of multiply, remainder of divide	$long	yes	yes	$mdhi
CAUSE	Cause register	$long	yes	yes	$cause
EPC	Program counter	$code[]	no	yes	$epc
SR	Status register	$long	no	no	$sr
VFP	Virtual frame pointer The virtual frame pointer is a software register that TotalView maintains. It is not an actual hardware register. TotalView computes the VFP as part of stack backtrace.	$long	no	no	$vfp

MIPS SR Register

For your convenience, TotalView interprets the bit settings of the SR register as outlined in the next table.

Value	Bit Setting	Meaning
0x00000001	IE	Interrupt enable
0x00000002	EXL	Exception level
0x00000004	ERL	Error level
0x00000008	S	Supervisor mode
0x00000010	U	User mode
0x00000018	U	Undefined (implemented as User mode)
0x00000000	K	Kernel mode
0x00000020	UX	User mode 64-bit addressing
0x00000040	SX	Supervisor mode 64-bit addressing
0x00000080	KX	Kernel mode 64-bit addressing
0x0000FF00	IM=i	Interrupt Mask value is *i*
0x00010000	DE	Disable cache parity/ECC
0x00020000	CE	Reserved
0x00040000	CH	Cache hit
0x00080000	NMI	Non-maskable interrupt has occurred
0x00100000	SR	Soft reset or NMI exception
0x00200000	TS	TLB shutdown has occurred
0x00400000	BEV	Bootstrap vectors
0x02000000	RE	Reverse-Endian bit
0x04000000	FR	Additional floating-point registers enabled
0x08000000	RP	Reduced power mode
0x10000000	CU0	Coprocessor 0 usable
0x20000000	CU1	Coprocessor 1 usable
0x40000000	CU2	Coprocessor 2 usable
0x80000000	XX	MIPS IV instructions usable

MIPS Floating-Point Registers

TotalView displays the MIPS floating-point registers in the Stack Frame Pane of the Process Window. Here is a table that describes how TotalView treats each floating-point register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0, F2	Hold results of floating-point type function; $f0 has the real part, $f2 has the imaginary part	$double	yes	yes	$f0, $f2
F1 – F3, F4 – F11	Temporary registers	$double	yes	yes	$f1 – $f3, $f4 – $f11
F12 – F19	Pass single- or double-precision actual arguments	$double	yes	yes	$f12 – $f19

Register	Description	Data Type	Edit	Dive	Specify in Expression
F20 – F23	Temporary registers	$double	yes	yes	$f20 – $f23
F24 – F31	Saved registers	$double	yes	yes	$f24 – $f31
FCSR	FPU control and status register	$int	yes	no	$fcsr

MIPS FCSR Register

For your convenience, TotalView interprets the bit settings of the MIPS FCSR register. You can edit the value of the FCSR and set it to any of the bit settings outlined in the following table.

Value	Bit Setting	Meaning
RM=RN	0x00000000	Round to nearest
RM=RZ	0x00000001	Round toward zero
RM=RP	0x00000002	Round toward positive infinity
RM=RM	0x00000003	Round toward negative infinity
flags=(I)	0x00000004	Flag=inexact result
flags=(U)	0x00000008	Flag=underflow
flags=(O)	0x00000010	Flag=overflow
flags=(Z)	0x00000020	Flag=divide by zero
flags=(V)	0x00000040	Flag=invalid operation
enables=(I)	0x00000080	Enables=inexact result
enables=(U)	0x00000100	Enables=underflow
enables=(O)	0x00000200	Enables=overflow
enables=(Z)	0x00000400	Enables=divide by zero
enables=(V)	0x00000800	Enables=invalid operation
cause=(I)	0x00001000	Cause=inexact result
cause=(U)	0x00002000	Cause=underflow
cause=(O)	0x00004000	Cause=overflow
cause=(Z)	0x00008000	Cause=divide by zero
cause=(V)	0x00010000	Cause=invalid operation
cause=(E)	0x00020000	Cause=unimplemented
FCC=(0/c)	0x00800000	FCC=Floating-Point Condition Code 0; c=Condition bit
FS	0x01000000	Flush to zero
FCC=(1)	0x02000000	FCC=Floating-Point Condition Code 1
FCC=(2)	0x04000000	FCC=Floating-Point Condition Code 2
FCC=(3)	0x08000000	FCC=Floating-Point Condition Code 3
FCC=(4)	0x10000000	FCC=Floating-Point Condition Code 4
FCC=(5)	0x20000000	FCC=Floating-Point Condition Code 5
FCC=(6)	0x40000000	FCC=Floating-Point Condition Code 6
FCC=(7)	0x80000000	FCC=Floating-Point Condition Code 7

Using the MIPS FCSR Register

You can change the value of the MIPS FCSR register within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FCSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FCSR to include **0x400**. The string displayed next to the FCSR register should now include **enables=(Z)**. Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See "*Setting Up a Debugging Session*" in the *TotalView Users Guide* for more information.

MIPS Delay Slot Instructions

On the MIPS architecture, jump and branch instructions have a "delay slot". This means that the instruction after the jump or branch instruction is executed before the jump or branch is executed.

In addition, there is a group of "branch likely" conditional branch instructions in which the instruction in the delay slot is executed only if the branch is taken.

The MIPS processors execute the jump or branch instruction and the delay slot instruction as an indivisible unit. If an exception occurs as a result of executing the delay slot instruction, the branch or jump instruction is not executed, and the exception appears to have been caused by the jump or branch instruction.

This behavior of the MIPS processors affects both the TotalView instruction step command and TotalView breakpoints.

The TotalView instruction step command will step both the jump or branch instruction and the delay slot instruction as if they were a single instruction.

If a breakpoint is placed on a delay slot instruction, execution will stop at the jump or branch preceding the delay slot instruction, and TotalView will not know that it is at a breakpoint. At this point, attempting to continue the thread that hit the breakpoint without first removing the breakpoint will cause the thread to hit the breakpoint again without executing any instructions. Before continuing the thread, you must remove the breakpoint. If you need to reestablish the breakpoint, you might then use the instruction step command to execute just the delay slot instruction and the branch.

A breakpoint placed on a delay slot instruction of a *branch likely* instruction will be hit only if the branch is going to be taken.

Sun SPARC

This section has the following information:

- SPARC General Registers
- SPARC PSR Register
- SPARC Floating-Point Registers
- SPARC FPSR Register
- Using the SPARC FPSR Register

The SPARC *processor supports the* IEEE *floating-point format*.

SPARC General Registers

TotalView displays the SPARC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
G0	Global zero register	$int	no	no	$g0
G1 – G7	Global registers	$int	yes	yes	$g1 – $g7
O0 – O5	Outgoing parameter registers	$int	yes	yes	$o0 – $o5
SP	Stack pointer	$int	yes	yes	$sp
O7	Temporary register	$int	yes	yes	$o7
L0 – L7	Local registers	$int	yes	yes	$l0 – $l7
I0 – I5	Incoming parameter registers	$int	yes	yes	$i0 – $i5
FP	Frame pointer	$int	yes	yes	$fp
I7	Return address	$int	yes	yes	$i7
PSR	Processor status register	$int	yes	no	$psr
Y	Y register	$int	yes	yes	$y
WIM	WIM register	$int	no	no	
TBR	TBR register	$int	no	no	
PC	Program counter	$code[]	no	yes	$pc
nPC	Next program counter	$code[]	no	yes	$npc

SPARC PSR Register

For your convenience, TotalView interprets the bit settings of the SPARC PSR register. You can edit the value of the PSR and set some of the bits outlined in the following table.

Value	Bit Setting	Meaning
ET	0x00000020	Traps enabled
PS	0x00000040	Previous supervisor
S	0x00000080	Supervisor mode
EF	0x00001000	Floating-point unit enabled
EC	0x00002000	Coprocessor enabled
C	0x00100000	Carry condition code
V	0x00200000	Overflow condition code
Z	0x00400000	Zero condition code
N	0x00800000	Negative condition code

SPARC Floating-Point Registers

TotalView displays the SPARC floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0, F1, F0_F1	Floating-point registers (*f* registers), used singly	$float	no	yes	$f0, $f1, $f0_f1
F2 – F31	Floating-point registers (*f* registers), used singly	$float	yes	yes	$f2– $f31
F0, F1, F0_F1	Floating-point registers (*f* registers), used as pairs	$double	no	yes	$f0, $f1, $f0_f1
F0/F1 – F30/F31	Floating-point registers (*f* registers), used as pairs	$double	yes	yes	$2 – $f30_f31
FPCR	Floating-point control register	$int	no	no	$fpcr
FPSR	Floating-point status register	$int	yes	no	$fpsr

TotalView allows you to use these registers singly or in pairs, depending on how they are used by your program. For example, if you use F1 by itself, its type is **$float**, but if you use the F0/F1 pair, its type is **$double**.

SPARC FPSR Register

For your convenience, TotalView interprets the bit settings of the SPARC FPSR register. You can edit the value of the FPSR and set it to any of the bit settings outlined in the following table.

Value	Bit Setting	Meaning
CEXC=NX	0x00000001	Current inexact exception
CEXC=DZ	0x00000002	Current divide by zero exception
CEXC=UF	0x00000004	Current underflow exception
CEXC=OF	0x00000008	Current overflow exception
CEXC=NV	0x00000010	Current invalid exception
AEXC=NX	0x00000020	Accrued inexact exception
AEXC=DZ	0x00000040	Accrued divide by zero exception
AEXC=UF	0x00000080	Accrued underflow exception
AEXC=OF	0x00000100	Accrued overflow exception
AEXC=NV	0x00000200	Accrued invalid exception
EQ	0x00000000	Floating-point condition =
LT	0x00000400	Floating-point condition <
GT	0x00000800	Floating-point condition >
UN	0x00000c00	Floating-point condition unordered
QNE	0x00002000	Queue not empty
NONE	0x00000000	Floating-point trap type None
IEEE	0x00004000	Floating-point trap type IEEE Exception
UFIN	0x00008000	Floating-point trap type Unfinished FPop
UIMP	0x0000c000	Floating-point trap type Unimplemented FPop
SEQE	0x00010000	Floating-point trap type Sequence Error
NS	0x00400000	Nonstandard floating-point FAST mode
TEM=NX	0x00800000	Trap enable mask – Inexact Trap Mask
TEM=DZ	0x01000000	Trap enable mask – Divide by Zero Trap Mask
TEM=UF	0x02000000	Trap enable mask – Underflow Trap Mask
TEM=OF	0x04000000	Trap enable mask – Overflow Trap Mask
TEM=NV	0x08000000	Trap enable mask – Invalid Operation Trap Mask
EXT	0x00000000	Extended rounding precision – Extended precision
SGL	0x10000000	Extended rounding precision – Single precision
DBL	0x20000000	Extended rounding precision – Double precision
NEAR	0x00000000	Rounding direction – Round to nearest (tie-even)
ZERO	0x40000000	Rounding direction – Round to 0
PINF	0x80000000	Rounding direction – Round to +Infinity
NINF	0xc0000000	Rounding direction – Round to –Infinity

Using the SPARC FPSR Register

The SPARC processor does not catch floating-point errors by default. You can change the value of the FPSR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSR to include **0x01000000** so that TotalView traps the "divide by zero" bit. The string displayed next to the FPSR register should now include **TEM=(DZ)**. Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See "H*andling Signals*" in Chapter 3 of the *TotalView Users Guide* for more information. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the **AEXC=(DZ)** bit.

Part IV: Appendix

appendix A

MPI Startup

Overview

TotalView Technologies products know about different MPI implementations. Because so many implementations are standard, our products usually do the right thing. Unfortunately, subtle differences in your environment or an implementation can cause difficulties that prevent our products from automatically starting your program. In these cases, you must declare what needs to be done.

The following explanation is for TotalView and MemoryScape.

The only way MemoryScape users can alter the way an MPI program stars up is by altering the **parallel_support.tvd** file, which is contained within the **totalview/lib** installation directory area. TotalView users can also alter this file and they can create a local definition.

If you are using a locally-installed MPI implementation, you should add it to your PATH variable. By default, our products use the information in PATH to find the parallel launcher (for example, **mpirun**, **mpiexec**, **poe**, **srun**, **prun**, **dmpirun**, and so on). Generally, if you can run your parallel job from a command line, TotalView and MemoryScape can also run it.

If you have multiple installed MPI systems—for example, multiple versions of MPICH installed on a common file server—only one can be in your path. In this case, you would need to specify an absolute path to launch it, which means you will need to customize the **TV::parallel_configs** list variable or the **parallel_support.tvd** file contained within your installation directory so that it does not rely on your PATH variable.

The easiest way to create your own startup configuration is to copy a similar configuration from the **TV::private::parallel_configs_base** variable to the **TV::parallel_configs** variable, then make changes.

When you add configurations, they are simply added to a list. This means that if TotalView Technologies supplies a definition named **foo** and you create a definition named **foo**, both exist and your product chooses the first one in the list. Because both are displayed, you must be careful to give each new definition a unique names.

Customizing Your Parallel Configuration

The **File > New** dialog box (TotalView) or the **Add parallel program** screen (MemoryScape) lets you select a parallel configuration. If the default configurations that TotalView Technologies provides do not meet your needs, you can either overwrite these configurations or create new ones by setting the **TV::parallel_configs** variable. Here are three examples:

```
dset TV::parallel_configs {
        #Argonne MPICH
        name:              MPICH;
        description:       Argonne MPICH;
        starter:           mpirun -tv -ksq %s %p %a;
        style:             setup_script;
        tasks_option:      -np;
        nodes_option:      -nodes;
        env_style:         force;
        pretest:           mpichversion;

        #Argonne MPICH2
        name:              MPICH2;
        description:       Argonne MPICH2;
        starter:           $mpiexec -tvsu %s %p %a;
        style:             manager_process;
        tasks_option:      -n;
        env_option:        -env;
        env_style:         assign_space_repeat;
        comm_world:        0x44000000;
        pretest:           mpich2version

        # AIX POE
        name:               poe - AIX;
        description:        IBM PE - AIX;
        tasks_option:       -procs;
        tasks_env:          MP_PROCS;
        nodes_option:       -nodes;
        starter:            /bin/poe %p %a %s;
        style:              bootstrap;
       env:            NLSPATH=/usr/lib/nls/msg/%L/%N/: \
                            /usr/lib/nls/msg/%L/%N.cat;
        service_tids:      2 3 4;
```

```
    comm_world:     0;
    pretest:        test -x /bin/poe
    msq_lib:        /usr/lpp/ppe.poe/lib/%m
   }
```

All lines (except for comments) end with a semi-colon (;). Add spaces freely to make what you enter readable as TotalView and MemoryScape ignore them.

Notice that the MPICH2 definition contains the **$mpiexec** variable. This variable is defined elsewhere in the **parallel_support.tvd** file as follows:

```
set mpiexec mpiexec;
```

There is no limit to how many definitions you can place within the **parallel_support.tvd** file or within a variable. The definitions you create will appear in the **Parallel** system pulldown list in the **File > New** dialog box (TotalView) or the **Add parallel program** screen (MemoryScape) and can be used an argument to the **–mpi** option of the CLI's **dload** command.

When running TotalView, you can set this variable in two places:

- Your system's **.tvdrc** file. If you set this variable here, everyone using this TotalView version will see the definition.
- Your **.totalview/tvdrc** file. You will be the only person to see this definition when you start TotalView.

The fields that you can set are as follows:

comm_world	Only use this option when **style** is set to **bootstrap**. This variable is the definition of MPI_COMM_WORLD in C and C++. MPI_COMM_WORLD is usually a **#define** or **enum** to a special number or a pointer value. If you do not include this field, TotalView and MemoryScape cannot acquire the rank for each MPI process.
description	(optional) A string describing what the configuration is used for. There is no length limit.
env	(optional) Defines environment variables that are placed in the starter program's environment. (Depending on how the starter works, these variables may not make their way into the actual ranked processes.) If you are defining more than one environment variable, define each in its own **env** clause. The format to use is: `variable_name=value`
env_option	(optional) Names the command-line option that exports environment variables to the tasks started by the launcher program. Use this option along with the **env_style** field.
env_style	(optional) Contains a list of environment variables that are passed to tasks.

assign: The argument to be inserted to the command-line option named in **env_option** is a comma-separated list of environment variable **name=value** pairs; that is,

```
    NAME1=VALUE1,NAME2=VALUE2,NAME3=VALUE3
```

This option is ignored if you do not use an **env_option** clause.

assign_space_repeat: The argument after **env_option** is a space-separated name/value pair that is assigned to an environment variable. The command within **env_option** is repeated for each environment variable; that is, suppose you enter:

```
-env NAME1 VALUE1 -env NAME2 VALUE2
-env NAME3 VALUE3
```

This mode is primarily used for the **mpiexec**.py MPICH2 starter program.

excenv One of the following three strings:

export: The argument to be inserted after the command named in **env_option**. This is a comma-separated list of environment variable names; that is,

```
    NAME1,NAME2,NAME3
```

This option is ignored if you do not use the **env_option** clause.

force: Environment variables are forced into the ranked processes using a shell script. TotalView or MemoryScape will generate a script that launches the target program. The script also tells the starter to run that script. This clause requires that your home directory be visible on all remote nodes. In most cases, you will use this option when you need to dynamically link memory debugging into the target. While this option does not work with all MPI implementations, it is the most reliable method for MPICH1.

none: No argument is inserted after **env_option**.

msq_lib (optional) Names the dynamically loaded library that TotalView and MemoryScape use when it needs to locate message queue information. You can name this file using either a relative or full pathname.

name A short name describing the configuration. This name shows up in such places as the **File > New** dialog box and in the **Process > Startup Parameter**'s Parallel tab in TotalView and the **Add parallel program** screen in MemoryScape. TotalView and MemoryScape remember which configuration you use when starting a program so that they can automatically reapply the configuration when you restart the program.

Because the configuration is associated with a program's name, renaming or moving the program destroys this association.

nodes_option	Names the command-line option (usually **–nodes**) that sets the number of node upon which your program runs. This statement does not define the value that is the argument to this command-line option. Only omit this statement if your system doesn't allow you to control the number of nodes from the command line. If you set this value to zero ("0"), this statement is omitted.
pretest	(optional) Names a shell command that is run before the parallel job is launched. This command must run quickly, produce a timely response, and have no side-effects. This is a test, not a setup hook. TotalView or MemoryScape may kill the test if it takes too long. It may call it more than once to be sure if everything is OK. If the shell command exit is not as expected, TotalView or Memoryscape complains and asks for permission before continuing,
pretext_exit	The expected error code of the pretest command. The default is zero.
service_tids	(optional) The list of thread IDs that TotalView and MemoryScape marks as service threads. When using TotalView, you can use the **View > Display Managers** command to tell TotalView to display them. A service thread differs from a system manager thread in that it is created by the parallel runtime and are not created by your program. POE for example, often creates three service threads.
starter	Defines a template that TotalView and MemoryScape use to create the command line that starts your program. In most cases, this template describes the relative position of the arguments. However, you can also use it to add extra parameters, commands, or environment variables. Here are the three substation parameters: **%a**: Replaced with the command-line arguments passed to rank processes. **%p**: Replaced with the absolute pathname of the target program. **%s**: Replaced with additional startup arguments. These are parameters to the starter process, not the rank processes. For example: `starter: mpirun -tv -all-local %s %p %a;` When the user selects a value for the option indicated by the **nodes_option** and **tasks_options**, the argument and the value are placed within the **%s** parameter. If you enter a value of 0 for either of these, MemoryScape and TotalView omit the parameter. In MemoryScape, 0 is the default.

style	MPI programs are launched in two ways: either by a manager process or by a script. Use this option to name the method, as follows: **manager_process**: The parallel system uses a binary manager process to oversee process creation and process lifetime. Our products attach to this process and communicate with it using its debug interface. For example, IBM's poe uses this style. `style: manager_process;` **setup_script**: The parallel system uses a script—which is often **mpirun**—to set up the arguments, environment, and temporary files. However, the script does not run as part of the parallel job. This script must understand the –**tv** command-line option and the TOTALVIEW environment variable. **bootstrap**: The parallel system attempts to launch an uninstruemented MPI by interposing TotalView or MemoryScape inside the parallel launch sequence in place of the target program. This does not work for MPICH and SGI MPT.
tasks_env	The name of an environment variable whose value is the expected number of parallel tasks. This is consulted when the user does not explicitly specify a task count.
tasks_option	(sometimes required) Lets you define the option (usually -**np** or -**procs**) that controls the total number of tasks or processes. Only omit this statement if your system doesn't allow you to control the number of tasks from the command line. If you set this to 0, this statement is omitted.

Index