TotalView

TotalView Reference Guide

Version 2024.1
April, 2024

PERFORCE

www.perforce.com

ACKNOWLEDGMENTS

TotalView by Perforce
http://totalview.io

Contents

About this Guide

This guide is organized in parts:

- Part I, Using the CLI on page 3 contains descriptions of all the CLI commands, the variables that you can set using the CLI, and other CLI-related information.
- Part II, Transformations on page 358 discusses formatting and transformations that display data in a clear and concise format to facilitate easier debugging sessions.
- Part III, Running TotalView on page 385 documents all possible command-line options as well as those that customize the behavior of the **tvdsvr**.
- Part IV, Platforms and Operating Systems on page 407 provides general information on compilers, runtime environments, operating systems, and supported architectures.

Resources

Please see The *Resources* appendix in the User Guide for more information on:

- a complete list of Classic TotalView documentation
- conventions used in the documentation
- contact information

The documentation for Classic TotalView could be useful if you are using features not yet supported in the modern TotalView UI by invoking commands through the Command Line Interface (CLI). The commands themselves are described in this Reference Guide, but the Classic TotalView documentation, in particular the User Guide, can provide useful information on how to use the commands to best advantage in debugging scenarios. This documentation is available in your Classic TotalView distribution, or on the TotalView documentation web site.

PART I Using the CLI

This part of the reference guide describes the TotalView Command Line Interface (CLI).

- CLI Command Summary on page 4

  Summarizes all CLI commands.

- CLI Commands on page 18

  Describes all commands 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.

- CLI Namespace Commands on page 217

  Describes commands found in the **TV::** namespace. These commands are seldom used interactively, as they are most often used in scripts.

- Batch Debugging Using tvscript on page 280

  Discusses how to create batch scripts that run TotalView unattended.

- TotalView Variables on page 297

  Describes all TotalView variables, including those uses 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.

CLI Command Summary

This chapter contains a summary of all TotalView debugger CLI commands. The commands are described in detail in CLI Commands on page 18 and CLI Namespace Commands on page 217.

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**]
[**-enabled_blocks** | **-disabled_blocks**]
[-block_images]
[-block_lines]

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 TotalView control

dattach [**-g** *gid*] [**-r** *hname*]
[**-ask_attach_parallel** | **-no_attach_parallel**]
[**-replay** | **-no_replay**]
[**-go** | **-halt**] [**-rank** *num*]
[**-c** { *core-file* | *recording-file* }]
[**-e**] *executable* [*pid-list*]
[**-parallel_attach_subs**et *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

dcalltree

Displays parallel backtrace data

[-data *pbv_data_array*] **[-show_details] [-sort** *columns*] **[-hide_backtrace]**
[-save_as_csv *filename*] **[-save_as_dot** *filename*]

dcheckpoint

Creates a checkpoint on IBM AIX

dcheckpoint [**-delete | -halt**]

dcont

Continues execution and waits for execution to stop

dcont

dcuda

Manages NVIDIA® CUDA™ GPU threads, providing the ability to inspect them, change the focus, and display their status.

dcuda

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_subs**et *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*]

domp

Displays OpenMP information using the OMPD API

domp [**-parallel_regions**] [**-task_regions**] [**-control_vars**] [**-ompd**] [**-threads {-regions** | **-functions** | **-stack}**] [**-send_symbols]**

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*] [-stats [-data]] *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*]

dsession

Loads a session

dsession [-load *session_name* **]**

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*

dskip

Create a rule to skip over or through a function

dskip [**over** | **through**] [**function** | **-function** | **-fu**] *function-name*

Create a rule to skip over or through a file

dskip [**over** | **through**] [**file** | **-file** | **-fi**] *filename*

Create a rule to skip over or through functions that are also contained in specific source files

dskip [**over** | **through**] { { **-function** | **-fu** } *function-name* | { **-rfunction** | **-rfu** } *function-regexp* } { { **-file** | **-fi** } *filename* | { **-gfile** | **-gfi** } *file-glob* }

Enable or disable skipping of a list of IDs

dskip [**enable** | **disable**] [*id*]

Delete a list of skip IDs

dskip delete [*id*]

Print information about a list of skip IDs

dskip info [*id*]

dstacktransform

Enables or disables the stack transform facility.

dstacktransform [enable | **disable** *id* | *transform_name* **]**

Prints the current state of rules and transforms.

dstacktransform **[list**]

Prints the enabled/disabled state of the stack transform facility.

dstacktransform **[status**]

Removes the rule with the given *id* from the stack transform facility.

dstacktransform **[remove** *id*]

Adds a new transform.

dstacktransform add [**-name** | **-n** *string*] [**-implementation** | **-i** *path*]

Adds a new transform rule.

dstacktransform add [-filter *test_function_list*] [**-transform** | **-t** *name*] [**-operation** | **-o** *operation_name* **[-position** | **-p** *integer*] [**-before** | **-b** *integer*]

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

Gets and sets 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

CLI Commands

This chapter lists all CLI commands.

Commands by Category

NOTE: This chapter describes some functionality that exists in the underlying debugging engine TotalView, but may not be supported in the TotalView user interface. To access these features, use the Command Line view or launch the Classic TotalView UI. See About this Guide on page 1 for more details.

General CLI Commands

These commands provide information on the general CLI operating environment:

- alias: Creates or views pseudonyms for commands and arguments.
- capture: Sends output to a variable for commands that print information.
- 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.
- dsession: Loads a session into TotalView.
- 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 support browsing the program's source files:

- dcalltree: Displays parallel backtrace data.
- ddown: Navigates through the call stack by manipulating the current frame.
- dexamine: Displays 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 (replay): Provides information for ReplayEngine and supports working 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.
- dskip: Creates and manages single-stepper skip rules.
- 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.

Platform-Specific CLI Commands

- dcuda: Manages NVIDIA® CUDA™ GPU threads, providing the ability to inspect them, change the focus, and display their status.
- domp: Displays OpenMP information using the OMPD API

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.
- dstacktransform: Maintains rules that change the displayed stack frames.

All Commands

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***. Often this is just a command name.

Description

The **alias** command associates a specified name with some defined text. This text can contain one or more commands. You can use an alias in the same way as a native TotalView or Tcl command. In addition, you can include an alias as part of the 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 specify only an ***alias-name*** argument, the CLI displays the definition of the alias.

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

The TotalView global startup file, **tvdinit.tvd**, defines a set of default one or two-letter aliases for all common commands. To see a list of these commands, type **alias** with no 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"
}
```

NOTE: Be aware that you can potentially create aliases that are nonsensical or incorrect because the CLI does not parse *defn-body* (the command's definition) until it is used. The CLI detects errors only when it tries to execute your alias.

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

To delete an alias, use the **unalias** command.

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 this alias is used.

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

Creates an alias that performs the same operations as that in the previous example, differing 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.

RELATED TOPICS

unalias **Command**

capture

Returns a command’s output as a string

Format

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

Arguments

-out

Captures only output sent to **stdout**.

-err

Captures only output sent to **stderr**.

-both

Captures output sent to both **stdout** and **stderr**. This is the default.

-f *filename*

Sends the captured output to *filename*. The file must be a writable Tcl file descriptor. Usually the Tcl file de-scrciptor name is obtained with **open** *filename* **w**.

command

The CLI command (or commands) whose output is being captured. If you specify more than one command, you must enclose them within braces (**{ }**).

Description

The **capture**command executes *command*, capturing in a string all output that would normally go to the con-sole. After *command* completes, it returns the string. This command is analogous to the UNIX shell’s back-tick feature (**`***command***`**). The **capture** command obtains 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 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.

RELATED TOPICS

drun **Command**

drerun **Command**

dactions

Displays information, and saves and reloads action points

Format

Displays information about action points.

> **dactions** [*ap-id-list*] [**-at** *source-loc*] [**-full**] [**-enabled** | **-disabled**] [**-enabled_blocks** | **-disabled_blocks**] [**-block_images** | **-block_lines**]

Saves action points to a file.

> **dactions -save** [*filename*]

Loads previously saved action points.

> **dactions -load**[*filename*]

Suppresses or unsuppresses action points.

> **dactions** [**-suppress** | **-unsuppress**]

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.

Without 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 displays just summary information for each.

-at *source-loc*

Displays the action points at *source-loc*. See **dbreak** for the details on the form of *source-loc*.

-full

Displays complete, rather than summary, information about the action points in the current share group. Complete information is the default when **dactions** is used with a single action point argument. Use **-full** to display complete information when invoking **dactions** with no arguments, or with two or more action point arguments.

-enabled

Shows only enabled action points.

-disabled

Shows only disabled action points.

-suppress

Effectively disables all existing action points. If the code is run, threads will not stop at any action points. Although you can create new action points (and delete existing ones), the new action points too will be effectively disabled.

-unsuppress

Restores all action points to the state they were in when suppressed. Any new action points added are set as enabled.

-enabled_blocks

When displaying the full information for an action point, only shows the enabled address blocks. (See example below.)

-disabled_blocks

When displaying the full information for an action point, only shows the disabled address blocks. (See example below.)

-block_images

When displaying the full information for an action point, shows the image name of each address block.

-block_lines

When displaying the full information for an action point, shows the source line of each address block. If the source line is followed by a tilde, the breakpoint block address is approximate.

-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.v4breakpoints**, 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 full breakpoint specification is printed (not returned), including the canonical file name's path.

Using the Action Point Identifier

To get the action point identifier, just enter **dactions** with no arguments. You need this identifier to 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.

Saving and Loading Action Points

The **-save** option writes action point information to a file so that either you or TotalView can restore your action points later. The **-load** option immediately reads the saved file. Using the ***filename*** argument with either option writes to or reads from this file. If you do not use this argument, TotalView names the file ***program_name*****.TVD.v4breakpoints** (where ***program_name*** is the name of your program), and writes it to the directory in which your program resides.

The information saved includes expressions 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.v4breakpoints**.

NOTE: 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 "Action Point Preferences" in the *TotalView User Guide*.

Suppressing and Unsuppressing Action Points

Suppress effectively disables all existing action points. If the code is run, threads will not stop at any action points. Although you can create new action points (and delete existing ones), the new action points too will be effectively disabled. Unsuppress restores all action points to the state they were in when suppressed. Any new action points added are set as enabled.

Command alias

Alias	Definition	Description
ac	**dactions**	Displays all action points

Examples

```
ac -at 81
```

Displays information about the action points on line 75. (This example uses the alias instead of the full command name.) Here is the output from this command:

```
d1.<> ac -at 75
1 shared action point for group 3:
```

```
1 [/home/totalview/tests/src/tx_blocks.cxx#75] Enabled
Address 0: [Enabled] main+0x1d0 (0x0040071c)
Share in group: true
Stop when hit: group
d1.<>
```

dactions 1 2

Displays information about action points 1 and 2, as follows:

```
d1.<> dactions 1 2
2 shared action points for group 3:
1 [/home/totalview/tests/src/tx_blocks.cxx#75] Enabled
2 [/home/totalview/tests/src/tx_blocks.cxx#48] Enabled
d1.<>
```

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
3 shared action points for group 3:
1 [/home/totalview/tests/src/tx_blocks.cxx#75] Enabled
2 [/home/totalview/tests/src/tx_blocks.cxx#69] Enabled
3 [/home/totalview/tests/src/tx_blocks.cxx#57] Enabled

d1.<> set group1 "2 3"
2 3
d1.<> eval ddisable $group1
d1.<> ac
3 shared action points for group 3:
1 [/home/totalview/tests/src/tx_blocks.cxx#75] Enabled
2 [/home/totalview/tests/src/tx_blocks.cxx#69] Disabled
3 [/home/totalview/tests/src/tx_blocks.cxx#57] Disabled
```

dfocus p1 dactions

Displays information about all action points defined in process 1.

```
d1.<> dfocus p1 dactions
3 shared action points for group 3:
1 [/home/totalview/tests/src/tx_blocks.cxx#75] Enabled
2 [/home/totalview/tests/src/tx_blocks.cxx#69] Disabled
3 [/home/totalview/tests/src/tx_blocks.cxx#57] Disabledd1.<>
```

dfocus p1 dactions -enabled

Displays information about all enabled action points in process 1

dactions -full

Displays more complete information about the action points. Here is an example of the output:

```
d1.<> dactions -full
3 shared action points for group 3:
1 [/home/totalview/tests/src/tx_blocks.cxx#75] Enabled
Address 0: [Enabled] main+0x1d0 (0x0040071c)
Share in group: true
Stop when hit: group
2 [/home/totalview/tests/src/tx_blocks.cxx#69] Disabled
Address 0: [Enabled] main+0x189 (0x004006d5)
Share in group: true
Stop when hit: process
3 [/home/totalview/tests/src/tx_blocks.cxx#57] Disabled
Address 0: [Enabled] main+0x9f (0x004005eb)
Address 1: [Enabled] main+0x257 (0x004007a3)
Address 2: [Disabled] main+0x266 (0x004007b2)
Share in group: true
Stop when hit: process
```

Examples of Action Points in Both Host and Dynamically Loaded Code

These examples show the **dactions** output for a program that dynamically loads code at runtime. In this case, an action point may contain a mixture of host and dynamically-loaded code address blocks, some of which may be identified as pending. (See "Pending Breakpoints" in the *TotalView User Guide*.) Note that these examples are for a CUDA program, but are relevant to any code loaded dynamically.

Both examples use **-block_lines** with **-full** to display the source line for each address block.

Pending and Mixed Breakpoint Example

Action points consisting only of invalid/nullified blocks are displayed as `Pending`:

```
dactions -full -block_lines

d1.<> dactions -full -block_lines
2 shared action points for group 3:
1 [/home/nvidia6/totalview/tests/src/tx_cuda.cu#218] Enabled
Pending
Share in group: true
Stop when hit: process
2 [/home/nvidia6/totalview/tests/src/tx_cuda.cu#219] Enabled
Address 0: [Disabled] ScrambleKernel+0x19, src/tx_cuda.cu#228
(0x00403998)
Address 1: [Enabled] ScrambleKernel+0x450, ../../src/tx_cuda.cu#220
(Location not mapped)
Address 2: [Enabled] ScrambleKernel+0x1c50, ../../src/tx_cuda.cu#220
(Location not mapped)
Share in group: true
Stop when hit: process
d1.<>
```

Note that:

Action point 1 has no valid address blocks, so is listed as Pending.

Action point 2 contains a mixture of host and GPU address blocks:

- Block 0 originally slid to line 228, but was disabled when the GPU code was loaded and TotalView found a better match at line 220. (See "Sliding Breakpoints" in the *TotalView User Guide*.)

- Block 1 and 2 show "Location not mapped" because the CLI focus was on the process, not the CUDA thread. Using `dfocus t1.-1 dactions` … would provide the GPU address.

Nullified and Pending Breakpoint Example

In this example, lines 220 and 221 contain "for" loops in the CUDA GPU code (a "for" loop typically has multiple line number symbols):

```
d1.<> l 218 -n 5
218
219 /* Loop over all elements of the matrix, scrambling them */
220 for (int i = start_i; i < A.width; i++)
221 for (int j = 0; j < A.width; j++)
222
```

Set some breakpoints:

```
d1.<> b 220
1
d1.<> b 221
2
```

Use **-full** and **-block_lines** to view the breakpoint's source lines and addresses:

```
d1.<> ac -full -block_lines
2 shared action points for group 3:
1 [/home/nvidia6/totalview/tests/src/tx_cuda.cu#220] Enabled
Pending
Share in group: true
Stop when hit: process
2 [/home/nvidia6/totalview/tests/src/tx_cuda.cu#221] Enabled
Address 0: [Enabled] ScrambleKernel+0x19, src/tx_cuda.cu#228 0x00403998)
Share in group: true
Stop when hit: process
```

Note that:

- Creating the action point 1 at line 220 in the GPU code caused it to slide to line 228 in the host code.
- Creating the action point 2 at line 221 in the GPU code caused it to slide to line 228 in the host code *and* nullified address block 0 in action point 1, which caused it to become pending.

  (See "Sliding Breakpoints" in the *TotalView User Guide*.)

Continue the process so that the GPU code is loaded and a CUDA thread stops at line 220, then view the output again:

```
...
Thread 1.-1 hit breakpoint 1 at line 220 in "ScrambleKernel(Matrix,int)"

d1.<> ac -full -block_lines
2 shared action points for group 3:
1 [/home/nvidia6/totalview/tests/src/tx_cuda.cu#220] Enabled
```

```
Address 0: (Nullified)
Address 1: [Enabled] ScrambleKernel+0x450, ../../src/tx_cuda.cu#220 (0x00dacfb0)
Address 2: [Enabled] ScrambleKernel+0x1c50, ../../src/tx_cuda.cu#220 (0x00dae7b0)
Share in group: true
Stop when hit: process
2 [/home/nvidia6/totalview/tests/src/tx_cuda.cu#221] Enabled
Address 0: [Disabled] ScrambleKernel+0x19, src/tx_cuda.cu#228 (0x00403998)
Address 1: [Enabled] ScrambleKernel+0x568, ../../src/tx_cuda.cu#221 (Location not mapped)
Address 2: [Enabled] ScrambleKernel+0x1c18, ../../src/tx_cuda.cu#221 (Location not mapped)
Share in group: true
Stop when hit: process
d1.<>
```

Loading the GPU code caused the action points to be reevaluated, thus adjusting their address blocks:

- Action point 1 added two address blocks for line 220, and thus is no longer pending. Note that this action point contains a mixture of valid and nullified blocks, therefore **dactions** lists address block 0 as Nullified rather than listing the entire breakpoint as Pending.
- Action point 2 added two address blocks for line 221, and block 0 was disabled because better matching line number symbols were added.

Extended example using -enabled_blocks and -disabled_blocks

dactions*n* [-enabled_blocks|]

This extended example demonstrates the use of these two options.

Set a break point:

```
d1.<> b {bar<std::vector<int, std::allocator<int> > >::bar(int)}
Incorporating 10079 bytes of DWARF '.debug_info' information for tx_test2.cxx
(linenumber)...done
1
```

Entering `dactions` reports on only the top-level action point associated with this action point number:

```
d1.<> dactions
1 shared action point for group 3:
1 [bar<std::vector<int,\ std::allocator<int>\ >\ >::bar(int)] Enabled
```

Entering `dactions` *n* reports on all action point instances (the address block) associated with this action point number:

```
d1.<> dactions 1
1 shared action point for group 3:
1 [bar<std::vector<int,\ std::allocator<int>\ >\ >::bar(int)] Enabled
Address 0: [Enabled] bar<std::vector<int,std::allocator<int> > >::bar+0x12
(0x004013d2)
Address 1: [Enabled] bar<std::vector<int,std::allocator<int> > >::bar+0x84
(0x00401444)
Address 2: [Disabled] bar<std::vector<double,std::allocator<double> > >::bar+0x12
(0x00401496)
Address 3: [Disabled] bar<std::vector<double,std::allocator<double> > >::bar+0x86
(0x0040150a)
Share in group: true
Stop when hit: process
```

Using `-enabled_blocks` reports on only enabled action point instances (the address block) associated with this action point number:

```
d1.<> dactions 1 -enabled_blocks
1 shared action point for group 3:
1 [bar<std::vector<int,\ std::allocator<int>\ >\ >::bar(int) Enabled
Address 0: [Enabled] bar<std::vector<int,std::allocator<int> > >::bar+0x12
(0x004013d2)
Address 1: [Enabled] bar<std::vector<int,std::allocator<int> > >::bar+0x84
(0x00401444)
Share in group: true
Stop when hit: process
```

Using `-disabled_blocks` reports on only disabled action point instances (the address block) associated with this action point number:

```
d1.<> dactions 1 -disabled_blocks
1 shared action point for group 3:
1 [bar<std::vector<int,\ std::allocator<int>\ >\ >::bar(int)] Enabled
Address 2: [Disabled] bar<std::vector<double,std::allocator<double> > >::bar+0x12
(0x00401496)
Address 3: [Disabled] bar<std::vector<double,std::allocator<double> > >::bar+0x86
(0x0040150a)
Share in group: true
Stop when hit: process
d1.<>
```

You could use this information, for example, to enable the currently disabled action point addresses:

```
d1.<> denable -block 2 3
```

RELATED TOPICS

Setting Action Points in the *TotalView User Guide*

Saving Action Points to a File in the *TotalView User Guide*

TV::auto_save_breakpoints **Variable**

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* (TOI). 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 the Data chapter of the TotalView User Guide.)

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, TotalView evaluates the expression. See About Expressions in the *TotalView User Guide*.

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.

RELATED TOPICS

Changing the Value of Variablesin the *TotalView User Guide*

Changing a Variable's Data Type in the *TotalView User Guide*

dattach

Brings currently executing processes under TotalView control

Format

dattach [**-g** *gid*] [**-r** *hname*]
[**-ask_attach_parallel** | **-no_attach_parallel**]
[-replay | -no_replay]
[**-go** | **-halt]**
[**-e**] *executable* [*pid-list*]
[**-c** *core-file* | *recording-file*] **[-rank** *num*]
[**-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 301 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. See the Setting Up Parallel Debugging Sessions chapter of the *TotalView User 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.

-attach_parallel

Attaches to any additional parallel processes in a parallel job.

-ask_attach_parallel

Specifies that TotalView should ask before attaching to parallel processes of a parallel job. The default is to automatically attach to processes. For additional information, see the Parallel Configuration in the **File > Preferences** Dialog Box in the *TotalView User Guide*.

If none of the **attach_parallel** switches is specified, and there is *exactly one* process ID in the process list, the user's preferences are used to determine whether to perform a parallel attach.

If none of the **attach_parallel** switches is specified, and there is *more than one* process ID in the process list, the default is **-no_attach_parallel**.

-no_attach_parallel

Does not attach to any additional parallel processes in a parallel job. For additional information, see the **Parallel Page** in the **File > Preferences** Dialog Box in the in-product helpfor Classic TotalView.

-replay | -no_replay

Enables or disables the ReplayEngine the next time the program is restarted. To enable, the feature must be supported and licensed on the current platform.

-go | -halt

Specifies to explicitly continue or halt target execution after attaching. The default is to leave the target's run state as it was before the attach.

-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 an executable file name. You need to use **-e** if the executable name begins with a dash (-) or consists of only numeric characters. Otherwise, you can just provide the executable file name.

executable

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.

-c *core-file* | *recording-file*

Loads the core file ***core-file*** or the ReplayEngine ***recording-file***, which restores a previous ReplayEngine debugging session. If you use this option, you must also specify an executable name (***executable***).

-parallel_attach_subset *subset_specification*

Defines a list of MPI ranks to attach to when an MPI job is created or attached to. The list is space-separated; each element can have one of three 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 must be either a positive decimal integer or **max** (the last rank in the MPI job).

A ***subset_specification*** that is the empty string (**""**) is equivalent to **0-max**.

For example:

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

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

Description

The **dattach** command attaches to one or more processes, making it possible to continue process execution under TotalView control.

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 ***executable*** 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 TotalView control

Examples

```
dattach mysys 10020
```

Loads debugging information for **mysys** and brings the process known to the runtime system as PID 10020 under TotalView control.

```
dattach -e 123 10020
```

Loads file 123 and brings the process known to the runtime system by PID 10020 under TotalView 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 runtime system by PID 10020 under TotalView 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 TotalView 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; then attaches to **slave** running on the **otherhost** host and inserts them both in the same control group.

RELATED TOPICS

Attaching to Processesin the *TotalView User Guide*

Examining Core Filesin the *TotalView User Guide*

ddetach **Command**

TV::parallel_attach **Variable**

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 absolute 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, usually using a line number or a pathname containing a file name, function name, and line number, each separated by **#** characters (for example, **file#line**). For more information, see "Qualifying Symbol Names" in the *Classic TotalView User Guide*.

For more information on breakpoint expressions, see **dbreak** on page 47, particularly Breakpoint Expressions.

-address *addr*

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

-stop_when_hit*width*

Identifies, using the ***width*** argument, any additional processes or threads to stop when stopping the thread that arrives at a barrier point.

If you do not use this option, the value of BARRIER_STOP_ALL indicates what to stop.

The argument ***width*** may have 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 only 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, releases all processes and threads held at the barrier. (***Released*** means that these threads and processes can run.) Setting this option stops additional threads contained in the same **group** or **process**.

If you do not use this option, the value of **BARRIER_STOP_WHEN_DONE** indicates any other processes or threads to stop.

Use the *width* argument indicates other stopped processes or threads. 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 creates 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 47.

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.

The **dbarrier** command is most often used 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. Note that they are just *released*, 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 unsatisfied 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 the processes and threads that 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.

NOTE: Barriers can create deadlocks. For example, if two threads participate in two different barriers, each could be left waiting at different 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 specifies if other threads should stop when a thread arrives at a barrier.

The **-stop_when_done** option controls the set of additional threads that are stopped 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.

Use the **none** argument for these options to not 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 (POI). 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.

RELATED TOPICS

Barrier Points in the *TotalView User Guide*

Creating a Satisfaction Set in the *TotalView User Guide*

Groups in TotalView in the *TotalView User Guide*

dactions **Command**

dbreak **Command**

denable **Command**

ddisable **Command**

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 absolute 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, usually using a line number or a pathname containing a file name, function name, and line number, each separated by **#** characters (for example, **file#line**). For more information, see "Qualifying Symbol Names" in the *Classic TotalView User 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

Stops 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 167 for more information.

-g

Stops 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 167 for more information.

-t

Stops 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 167 for more information.

-l*lang*

Sets the programming language used when you are entering expression ***expr***. Enter either: **c**, **c++**, **f7**, **f9**, or **asm** (for C, C++, FORTRAN 77, Fortran 9x, and assembler, respectively). If you do not specify a language, TotalView assumes the language in which the routine at the breakpoint was written.

-e *expr*

When the breakpoint is hit, TotalView evaluates expression ***expr*** in the context of the thread that hit the breakpoint. See Breakpoint Expressions.

-pending

If TotalView cannot find a location to set the breakpoint, adding this option creates 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 triggered when execution arrives at the specified location, stopping each thread that arrives at a breakpoint. This command returns the ID of the new breakpoint. If a line does not contain an executable statement, the CLI cannot set a breakpoint.

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.

Specifying a procedure name without a line number sets 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.

The -pending Option

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 allows a breakpoint to be created when the breakpoint expression does not currently match any program locations. For example, a common use case is to create a pending function breakpoint with a breakpoint expression that matches the name of a function that will be loaded at runtime via **dlopen**(), CUDA kernel launch, or anything that dynamically loads executable code.

When displaying information on a pending breakpoint's status, TotalView displays the breakpoint expression followed by "(pending)" indicating that the breakpoint currently contains no valid addresses.

Note that using this option doesn't catch typos or errors in the user's input. For example, if you want to set a breakpoint on a function `foo`, but you typed `voo` instead, a pending breakpoint is immediately created for the function `voo`, which would not be your intention.

To set **dbreak** to always use the **-pending** option, use the **TV::default_breakpoints_pending** state variable.

A stop group Breakpoint

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.

Default Focus Width

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

Breakpoint Expressions

Breakpoint expressions, also called breakpoint specifications, are used in both breakpoints and barrier points, so this discussion is relevant to both.

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**.

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 breakpoint on each of the overloaded functions.

Set a breakpoint 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 breakpoint is placed, and are as follows:

- **[[##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.

- **class** *class_name*

  Specifies that the breakpoint should be planted in all member functions of class ***class_name***.

- **virtual** *class::signature*

  Specifies that the breakpoint should be planted in all virtual member functions that match ***signature*** and are in the class 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 SHARE_ACTION_POINT is **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.

RELATED TOPICS

Barrier Points in the *TotalView User Guide*

Creating Conditional Breakpoints in the *TotalView User Guide*

Groups in TotalView in the *TotalView User Guide*

dactions **Command**

dbreak **Command**

denable **Command**

ddisable **Command**

dcache

Clears the remote library cache

Format

dcache -flush

Arguments

-flush

Deletes all files from the library cache that are not currently being used.

Description

The **dcache -flush** command removes the library files that it places in your cache, located in the **.TotalView/lib_-cache** subdirectory 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 at any time, use this command to remove files not currently being used.

dcalltree

Displays parallel backtrace data

Format

dcalltree [-data*pbv_data_array***] [-show_details] [-sort***columns***] [-hide_backtrace] [-save_as_csv***filename***] [-save_as_dot***filename***]**

Arguments

-data*pbv_data_array*

Captures the data from calling **dcalltree** in an associative Tcl array rather than writing the data to the console.

-show_details

Displays the data with all processes and threads displayed.

-hide_backtrace

Displays the data with only root and leaf nodes displayed.

-sort *column*

Sorts the data display based on the data in a particular column. The possible arguments are *Processes*, *Location*, *PC*, *Host*, *Rank*, *ID*, and *Status*.

-save_as_csv*filename*

Saves the backtrace data as a file of comma-separated values under the name *filename*.

-save_as_dot*filename*

Saves the backtrace data as a dot file under the name *filename*. Dot is a plain text graph description language.

Description

The **dcalltree** command shows the state of processes and threads in a parallel job. Normally the output is written to the console, but the **-data** subcommand makes the data available as a Tcl associative array. The associative array has the following format:

```
{
{
Key <value>
Level <value>
Processes <value>
Location <value>
PC <value>
Host <value>
Rank <value>
ID <value>
Status <value>
}
{
...
}
}
```

If you are using the Classic TotalView UI, the data displayed by this command is similar to the data displayed in the Parallel Backtrace View window.

The **-show_details** and **-hide_backtrace** switches pull in opposite directions. The **-show_details** switch shows the maximum data, including all processes and threads. The **-hide_backtrace** command hides any intermediate nodes, displaying only the root and leaf nodes. If used together, this results in a display of root and leaf nodes and all threads. This reduction can help to de-clutter the data display if the number of processes and threads is large.

Command alias

Alias	Definition	Description
ct	**dcalltree**	Prints data to console
ctd	**dcalltree -data**	Puts data in a Tcl associative array
ctsd	**dcalltree -show_details**	Prints more complete data
ctshb	**dcalltree -hide_backtrace**	Prints data only on root and leaf nodes

Examples

```
dfocus group dcalltree
```

This example first changes the focus to the group using dfocus, then calls **dcalltree** with no switches. Note that the ID column is a compressed **ptlist** describing process and thread count, range, and IDs. See Compressed List Syntax (ptlist) for more information.

```
Processes Location PC Host Rank ID Status
--------- -------- -- ---- ---- -- ------
12 / ... <local> -1 4:12[p1-4.1-3] ...
4 _start 0x004011b9 <local> -1 4:4[p1-4.1] ...
4 __libc_start_main 0x2b3425358184 <local> -1 4:4[p1-4.1] ...
4 main 0x004035bf <local> -1 4:4[p1-4.1] ...
4 fork_wrapper 0x00402790 <local> -1 4:4[p1-4.1] ...
4 forker 0x0040274b <local> -1 4:4[p1-4.1] ...
4 snore 0x00401c11 <local> -1 4:4[p1-4.1] ...
1 snore#681 0x00401c05 <local> -1 2.1 - 47502964801120 Stopped
1 snore#705 0x00401c9b <local> -1 4.1 - 47502964801120 Breakpoint
2 wait_a_while 0x00401a09 <local> -1 2:2[p1.1, p3.1] Stopped
2 __select_nocancel 0x2b34253f56e2 <local> -1 2:2[p1.1, p3.1] Stopped
8 start_thread 0x2b3424db1143 <local> -1 4:12[p1-4.1-3] ...
8 snore_or_leave 0x004021cb <local> -1 4:8[p1-4.2-3] ...
8 snore ... <local> -1 4:8[p1-4.2-3] ...
1 snore#681 0x00401c05 <local> -1 1.2 - 1082132800 Breakpoint
1 snore#681 0x00401c05 <local> -1 1.3 - 1090525504 Stopped
1 snore#705 0x00401c9b <local> -1 2.2 - 1082132800 Breakpoint
1 snore#681 0x00401c05 <local> -1 2.3 - 1090525504 Stopped
1 snore#681 0x00401c05 <local> -1 4.2 - 1082132800 Stopped
1 snore#681 0x00401c05 <local> -1 4.3 - 1090525504 Stopped
2 wait_a_while ... <local> -1 1:2[p3.2-3] ...
```

```
dcalltree -show_details
```

By adding the **-show_details**, switch, you get more complete output:

```
Processes Location PC Host Rank ID Status
--------- -------- -- ---- ---- -- ------
12 / ... <local> -1 4:12[p1-4.1-3] ...
4 _start 0x004011b9 <local> -1 4:4[p1-4.1] ...
4 __libc_start_main 0x2b3425358184 <local> -1 4:4[p1-4.1] ...
4 main 0x004035bf <local> -1 4:4[p1-4.1] ...
4 fork_wrapper 0x00402790 <local> -1 4:4[p1-4.1] ...
4 forker 0x0040274b <local> -1 4:4[p1-4.1] ...
4 snore 0x00401c11 <local> -1 4:4[p1-4.1] ...
1 snore#681 0x00401c05 <local> -1 2.1 - 47502964801120 Stopped
1 snore#705 0x00401c9b <local> -1 4.1 - 47502964801120 Breakpoint
2 wait_a_while 0x00401a09 <local> -1 2:2[p1.1, p3.1] Stopped
2 __select_nocancel 0x2b34253f56e2 <local> -1 2:2[p1.1, p3.1] Stopped
1 __select_nocancel 0x2b34253f56e2 <local> -1 1.1 - 47502964801120 Stopped
1 __select_nocancel 0x2b34253f56e2 <local> -1 3.1 - 47502964801120 Stopped
8 start_thread 0x2b3424db1143 <local> -1 4:12[p1-4.1-3] ...
8 snore_or_leave 0x004021cb <local> -1 4:8[p1-4.2-3] ...
8 snore ... <local> -1 4:8[p1-4.2-3] ...
1 snore#681 0x00401c05 <local> -1 1.2 - 1082132800 Breakpoint
1 snore#681 0x00401c05 <local> -1 1.3 - 1090525504 Stopped
1 snore#705 0x00401c9b <local> -1 2.2 - 1082132800 Breakpoint
1 snore#681 0x00401c05 <local> -1 2.3 - 1090525504 Stopped
1 snore#681 0x00401c05 <local> -1 4.2 - 1082132800 Stopped
1 snore#681 0x00401c05 <local> -1 4.3 - 1090525504 Stopped
2 wait_a_while ... <local> -1 1:2[p3.2-3] ...
1 __select_nocancel 0x2b34253f56e2 <local> -1 3.3 - 1090525504 Stopped
1 wait_a_while#580 0x004019e9 <local> -1 3.2 - 1082132800 Breakpoint
```

```
dcalltree -show_details -hide_backtrace
```

Adding the **-hide_backtrace** switch reduces the clutter somewhat:

```
Processes Location PC Host Rank ID Status
--------- -------- -- ---- ---- -- ------
12 / ... <local> -1 4:12[p1-4.1-3] ...
1 __select_nocancel 0x2b34253f56e2 <local> -1 3.3 - 1090525504 Stopped
1 __select_nocancel 0x2b34253f56e2 <local> -1 1.1 - 47502964801120 Stopped
1 __select_nocancel 0x2b34253f56e2 <local> -1 3.1 - 47502964801120 Stopped
1 snore#681 0x00401c05 <local> -1 2.1 - 47502964801120 Stopped
1 snore#705 0x00401c9b <local> -1 4.1 - 47502964801120 Breakpoint
1 snore#681 0x00401c05 <local> -1 1.2 - 1082132800 Breakpoint
1 snore#681 0x00401c05 <local> -1 1.3 - 1090525504 Stopped
1 snore#705 0x00401c9b <local> -1 2.2 - 1082132800 Breakpoint
1 snore#681 0x00401c05 <local> -1 2.3 - 1090525504 Stopped
1 snore#681 0x00401c05 <local> -1 4.2 - 1082132800 Stopped
1 snore#681 0x00401c05 <local> -1 4.3 - 1090525504 Stopped
1 wait_a_while#580 0x004019e9 <local> -1 3.2 - 1082132800 Breakpoint
```

Here is code to get the location of all threads that are at a breakpoint:

```
dcalltree -data pbv_data_array -show_details
```

```
foreach { data_record } [array get pbv_data_array] {
set print_location 0
set break_location
foreach {title value} $data_record {
if {$title == "Location"} {
set break_location $value
}
if {$value == "Breakpoint"} {
set print_location 1
}
if {1 == $print_location} {
puts stdout "Breakpoint found at $break_location"
set print_location 0
}
}
}
```

RELATED TOPICS

Parallel Backtrace View in the *Classic TotalView User Guide*

dcheckpoint

Creates a checkpoint image of processes (IBM RS6000 only)

Format

Creates a checkpoint on IBM RS6000 machines.

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

Arguments

-by *process_set*

This option can take two possible values:

pe

Checkpoint the Parallel Environment job. This value is the default.

pid

Checkpoint the focus process.

-delete

Processes exit after the checkpoint occurs.

-halt

Processes halt after the checkpoint occurs.

Description

The **dcheckpoint** command saves program and process information to a file. This information includes process and group IDs. Later, use the **drestart** command to restart the program.

NOTE: This command does not save TotalView breakpoint information. To save breakpoints, use the dactions command.

By default, TotalView checkpoints the Parallel Environment job. To checkpoint a particular process, make that process the focus and use the **pid** argument to **-by**. If the focus is a group that contains more than one process, the CLI displays an error -message.

By default, the checkpointed processes stop, allowing you to investigate a -program's state at the checkpointed position. You can modify this behavior with the **-delete** and **-halt** options.

When you request a checkpoint:

- TotalView 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 could control it.)

- The CLI detaches from processes before they are checkpointed. After checkpointing, the CLI automatically reattaches to them.

Examples

```
dcheckpoint
```

Checkpoints the Parallel Environment job. All associated processes stop.

```
f3 dcheckpoint -by pid
```

Checkpoints process 3. Process 3 stops.

```
dcheckpoint -by pe -halt
```

Checkpoints the Parallel Environment job. All associated processes halt.

RELATED TOPICS

drestart **Command**

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.

NOTE: You can interrupt this action using *Ctrl+C*to stop process execution.

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 (POI).

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 it interactively.

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 are 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 that 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 that belong to processes 1, 2, and 3.

```
co
```

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

RELATED TOPICS

Starting Processes and Threads in the *Classic TotalView User Guide*

dgo**Command**

dwait**Command**

dcuda

Manages GPU threads

Format

dcuda block [(Bx,By,Bz)]
dcuda thread [(Tx,Ty, Tz)]
dcuda kernel
dcuda device [<n>]
dcuda sm [<n>]
dcuda warp [<n>]
dcuda lane [<n>]
dcuda info-system
dcuda info-device
dcuda info-sm
dcuda info-warp
dcuda info-lane
dcuda focus (Bx,By,Bz),(Tx,Ty, Tz)
dcuda hwfocus <D/S/W/L>

Arguments

Bx, *By*, *Bz*

The x, y and z block indices

Tx, *Ty*, *Tz*

The x, y, and z thread indices

D/S/W/L

The coordinates defining the physical space of the hardware:

D: device number
S: streaming multiprocessor (SM)
W: warp (WP) number on the SM
L: lane (LN) number on the warp

Description

The **dcuda** commands allow you to manage and view GPU threads, in either the logical coordinate space of block and thread indices (<<<(Bx,By,Bz),(Tx,Ty,Tz)>>>) or the physical coordinate space that defines the hardware (the device number, the streaming multiprocessor number on the device, the warp number on the SM, and lane number on the warp).

dcuda block [(Bx,By,Bz)]

- With no arguments, shows the current CUDA block

- With a block argument of the form (*Bx,By,Bz*), changes the CUDA focus to that block. Parameters to the right (*By* and *Bz*, or just *Bz*) may be omitted; these are unchanged.

dcuda thread [(Tx,Ty,Tz)]

- With no arguments, shows the current CUDA thread.
- With a thread argument of the form (*Tx,Ty,Tz*), changes the CUDA focus to that thread. Parameters to the right (*Ty* and *Tz*, or just *Tz*) may be omitted; these are unchanged.

dcuda kernel

Displays the logical and hardware coordinates of the current CUDA context.

dcuda device [<n>]

- With no arguments, shows the current CUDA device.
- With a numeric argument, changes the CUDA device focus to that device.

dcuda sm [<n>]

- With no arguments, shows the current CUDA SM (streaming multiprocessor).
- With a numeric argument, changes the CUDA SM focus to that SM.

dcuda warp [<n>]

- With no arguments, shows the current CUDA warp.
- With a numeric argument, changes the CUDA warp focus to that warp.

dcuda lane [<n>]

- With no arguments, shows the current CUDA lane.
- With a numeric argument, changes the CUDA lane focus to that lane.

dcuda info-system

Displays the CUDA devices in the system.

dcuda info-device

Displays currently running SMs in the current device.

dcuda info-sm

Displays valid warps in the current SM.

dcuda info-warp

Displays valid lanes in the current warp.

dcuda info-lane

Displays the current lane.

dcuda focus (Bx,By, Bz),(Tx,Ty,Tz)

Changes the focus via CUDA logical coordinates of the form `<<<(Bx,By,Bz),(Tx,Ty,Tz)>>>`.

The following abbreviations are also accepted:

```
<<<Tx>>>
<<<(Tx)>>>
<<<(Tx,Ty)>>>
<<<(Tx,Ty,Tz)>>>
<<<(Bx),(Tx)>>>
<<<(Bx),(Tx,Ty)>>>
<<<(Bx),(Tx,Ty,Tz)>>>
<<<(Bx,By),(Tx)>>>
<<<(Bx,By),(Tx,Ty)>>>
<<<(Bx,By),(Tx,Ty,Tz)>>>
<<<(Bx,By,Bz),(Tx)>>>
<<<(Bx,By,Bz),(Tx,Ty)>>>
<<<(Bx,By,Bz),(Tx,Ty,Tz)>>>
```

Angle brackets are optional, but must be balanced.

dcuda hwfocus <D/S/W/L>

Changes the focus via CUDA hardware coordinates of the form D/S/W/L, S/W/L, W/L, or L.

Command alias

Alias	Definition	Description
cuda	**dcuda**	Writes out the focus thread, as in **dcuda kernel**.

Examples

Displaying device information

```
dcuda info-device
```

Output:

```
DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
SM: 0/30 valid warps: 0x0000000000000001
```

```
dcuda info-sm
```

Output:

```
DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
SM: 0/30 valid warps: 0x0000000000000001
```

```
  WP: 0/32 valid/active/divergent lanes: 0x0000000f/0x0000000f/0x00000000 block:
  (0,0,0)

dcuda info-warp
```

Output:

```
  DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
  SM: 0/30 valid warps: 0x0000000000000001
  WP: 0/32 valid/active/divergent lanes: 0x0000000f/0x0000000f/0x00000000 block:
  (0,0,0)
  LN: 0/32 pc=0x000000001ef2efa8 thread: (0,0,0)
  LN: 1/32 pc=0x000000001ef2efa8 thread: (1,0,0)
  LN: 2/32 pc=0x000000001ef2efa8 thread: (0,1,0)
  LN: 3/32 pc=0x000000001ef2efa8 thread: (1,1,0)

dcuda info-lane
```

Output:

```
  DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
  SM: 0/30 valid warps: 0x0000000000000001
  WP: 0/32 valid/active/divergent lanes: 0x0000000f/0x0000000f/0x00000000 block:
  (0,0,0)
```

Displaying the focus

```
dcuda warp sm
```

Output:

```
  sm 0 warp 0

dcuda lane device
```

Output:

```
  device 0 lane 3

dcuda thread
```

Output:

```
  thread (1,1,0)

dcuda kernel
```

Output:

```
  device 0, sm 0, warp 0, lane 3, block (0,0,0), thread (1,1,0)
```

Changing the focus

In these commands, note that TotalView assigns CUDA threads a negative thread ID. In the examples here, the CUDA thread is labeled "1.-1".

```
dcuda thread (1,1,0)
```

Changes the CUDA focus to the thread represented by logical coordinates 1,1,0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 3, block (0,0,0), thread
  (1,1,0)

dcuda lane 2
```

Changes the CUDA focus to lane 2.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 2, block (0,0,0), thread
  (0,1,0)

dcuda lane 1 sm 0
```

Changes the CUDA focus to lane 1 and to SM 0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 1, block (0,0,0), thread
  (1,0,0)

dcuda thread 0,0,0
```

Changes the CUDA focus to thread 0,0,0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 0, block (0,0,0), thread
  (0,0,0)

dcuda thread 1
```

Changes the CUDA focus to thread 1,0,0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 1, block (0,0,0), thread
  (1,0,0)
```

RELATED TOPICS

Using the CUDA Debugger in the *TotalView User Guide*

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

Deletes all action points in the current focus.

Description

The **ddelete** command permanently removes one or more 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 (POI).

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 runtime environment.

You can detach any process controlled by the CLI; the process being detached need not have been 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 (POI).

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.

RELATED TOPICS

Detaching from Processes in the *Classic TotalView User Guide*

dattach**Command**

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 identify the instances to disable. Obtain a list of these numbers using the **dactions**command.

-a

Disables all action points.

Description

The **ddisable** command temporarily deactivates action points. To delete an action point, use **ddelete**.

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 (POI).

Note that you cannot disable a nullified action point, i.e., one that points to an invalid address block.

Command alias

Alias	Definition	Description
di	**ddisable**	Temporarily disables action points

Examples

```
ddisable 3 7
```

Disables the action points with IDs 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.

```
ddisable 1 -block 0
ddisable: Actionpoint 1 block 0 is nullified and cannot be disabled
```

Disabling an action point that is nullified, i.e., one that points to an invalid address block, returns an error message.

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* ... | **-all**]

Arguments

-now

Includes **RTLD_NOW** in the **dlopen** command's mode argument. (Now immediately resolves all undefined symbols.)

-lazy

Includes **RTLD_LAZY** in the **dlopen** command's mode argument. (Lazy tries to resolve unresolved symbols as code is executed, rather than now.)

-local

Includes **RTLD_GLOBAL** in the **dlopen** command's mode argument. (Local makes library symbols unavailable to libraries that the program subsequently loads.) This argument is the default.

-global

Includes **RTLD_LOCAL** in the **dlopen** command's mode argument. (Global makes library symbols 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 use **ddlopen** without arguments or use the **-list** option without a DLL ID list (**ddlopen -list**), TotalView displays information about all DLL IDs.

dll-ids

A list of one or more DLL IDs. DLL IDs are the return values when you use the **ddlopen** command to load DLLs.

Description

The **ddlopen** command dynamically loads shared object libraries, or lists the shared object libraries loaded using this or the **Tools > Debugger Loaded Libraries** command, available in Classic TotalView.

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

NOTE: **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.

Also, the **ddlopen** command operates by calling **dlopen(3)**. This can alter the string returned by **dlerror(3)**. Thus, issuing a **ddlopen** command can change the values returned to the application by any of its subsequent **dlerror(3)** calls.

The **-now** and **-lazy** options indicate whether **dlopen** immediately resolves unresolved symbol references or defers 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. (Linux supports only the **-global** option. If you don't specify an option, the default is the **-local** option.)

After entering 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 IDfor 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** represents a shareable object that was dynamically loaded by the **ddlopen** command. 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.

To obtain a listing of all objects loaded using **ddlopen**, enter just **ddlopen** without a *filespec* argument, or **ddlopen -list**.

The **ddlopen** command prints its output directly to the console.

Examples

```
ddlopen "mpistat.so"
1
```

Loads the **mpistat.so** library file. The return value (1) indicates the process into which TotalView loaded the library.

```
dfocus g ddlopen "mpistat.so(mpistat.o)"
2
```

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
dll-id susp-eval-id [Switches] DLL name p.t dlopen handle (TV::expr get p.t status)
1 2 -lazy tx_shared_lib.so 1.1 3
```

Prints the list of shared objects dynamically loaded by the **ddlopen** command.

ddlopen prints its output directly to the console. Type "help output" for more information.

RELATED TOPICS

Preloading Shared Libraries in the *Classic TotalView User Guide*

TV::dll**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 and 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.

RELATED TOPICS

dup **Command**

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

Enables all action points.

-block *number-list*

If you set a breakpoint on a line that is ambiguous, this option names which instances to enable. Use the **dactions** command to obtain a list of these numbers.

Description

The **denable** command reactivates action points that you previously disabled with the **ddisable** command. The **-a** option enables all action points in the current focus.

Note that you cannot enable an action point with nullified blocks, i.e. those that point to an invalid address block.

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

If you do not indicate a focus, the default focus is the process of interest (POI).

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 action 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.

```
denable 1 -block 0
denable: Actionpoint 1 block 0 is nullified and cannot be enabled
```

Enabling an action point that is nullified, i.e. that points to an invalid address block, returns an error message.

RELATED TOPICS

Enabling Action Pointsin the *TotalView User Guide*

ddisable**Command**

dbarrier**Command**

dbreak**Command**

dwatch**Command**

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| **-column_count** *cnt*

Specifies the number of columns to display. Without this option, the CLI determines this number of columns based on the data's wordactid size and format.

-c | **-count** *cnt*

Specifies the number of elements to examine. Without 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**

Does not display memory values with a prefixed *address:* field or address annotations. This option is incompatible with **-memory_info**.

-f | **-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.

a | **address**

Interprets memory as addresses; the word size is always the size of a pointer

b | **binary**

Binary; this can also be abbreviated to **t**

c | **char**

Unsigned character

d | **dec**

Signed decimal value of size 1, 2, 4, or 8 bytes

f | **float**

Signed float value, either 4 or 8 byte word size

h | **hex**

Unsigned hexadecimal value of size 1, 2, 4, or 8 bytes

i | **instruction**

Sequence of instructions

o | **oct**

Unsigned octal value of size 1, 2, 4, or 8 bytes

s | string

String

-m | -memory_info

Shows information about the type of memory associated with the address. Without this 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 include the pre-guard region if the user specified guards in the memory debugging configuration.

-sc | -show_chars

Shows a trailing character dump for each line. Without this option, the CLI does not show the trailing characters.

-sl | -string_length *len*

Specifies the maximum size string to display. Without this option, the length is all characters up to the first null character.

-w | -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

Examines 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.

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

Command alias

Alias	Definition	Description
x	**dexamine**	Examines (dumps) memory

Examples

```
d1.<> dexamine -f b {dbl_array[1]}
0x7fffff0d70e8: 0100000000000011001100110011001100110011001100110011001100110011
0x7fffff0d70f0:
```

Examines the memory of element one of `dbl_array` in binary format.

```
d1.<> dexamine -wordsize 8 {dbl_array[1]}
0x7fffff0d70e8: 0x4003333333333333
0x7fffff0d70f0:
```

Examines the memory of element one of `dbl_array` and applies an eight-bit word size to the formatting output.

```
d1.<> dexamine -data_only {dbl_array[1]}
0x4003333333333333
```

Examines the memory of element one of `dbl_array` and displays only the memory values and not the address field or address annotations.

```
d1.<> dexamine -format oct {dbl_array[1]}
0x7fffff0d70e8: 0040003146314631463146 3
0x7fffff0d70f0:
```

Examines the memory of element one of `dbl_array` and formats the output in octal.

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 when using the **dprint -nowait** command. Other possibilities are if an error occurred in a function call in an eval point, in an expression in a **Tools > Evaluate** window (available in Classic TotalView), or if you use a **$stop**function.

Use this command as follows:

- 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 command to 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 to be the target of subsequent CLI commands.

command

A CLI command that operates on its own local focus. This argument may be a single command or a list.

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, or just the command that immediately follows.

If a ***command*** argument is provided, the focus is set temporarily, ***command*** is executed in the new focus, and then the focus is restored to its old value.

For example, to continue the TotalView group containing the focus process, you could type:

```
dfocus g dgo
```

To stop process 3 and display backtraces for each of its threads, type:

```
dfocus p3 { dhalt ; dwhere }
```

Summary

- If ***ptset*** is provided but not ***command***: The default focus for subsequent commands is changed to ***ptset***.
- If neither ***command*** *nor* ***ptset*** are provided: The current default focus is returned as a string value.
- If *no* argument is provided: **dfocus** returns the focus as a string value.
- If *any* argument is provided: **dfocus** returns the result of the command.

For more information on command output, enter "help output".

For more information on P/T sets, see “Group, Process and Thread Control” of the *Classic TotalView User 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 uses 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* (TOI) 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.

“Group, Process, and Thread Control” in the *Classic TotalView User Guide* contains additional **dfocus** examples.

RELATED TOPICS

Groups in TotalView in the *TotalView User Guide*

dga

Displays Global Array variables

Format

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

Arguments

-lang

Specifies the language conventions to use. Without this option, TotalView uses the language used by the *thread of interest* (TOI).

lang_type

Specifies the language type to use when displaying a global array. The type must be either "**c**" or "**f**".

handle_or_name

Displays an array. This can be either a numeric handle or the name of the array. Without this argument, TotalView displays a list of all Global Arrays.

slice

Displays only a slice (that is, part of an array). If you are using C, you must place the array designators within braces **{}** because square brackets (**[]**) have special meaning in Tcl.

Description

The **dga** command displays 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 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
```

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 by its handle.

RELATED TOPICS

Debugging Global Arrays Applications in the *Classic TotalView User Guide*

dgo

Resumes execution of processes

Format

dgo

Arguments

-back | -b

(ReplayEngine only). Runs the nonheld 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 resumes execution of all nonheld processes and threads in the current focus. 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.

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 (POI).

Command alias

Alias	Definition	Description
g	**dgo**	Resumes execution
G	**{dfocus g dgo}**	Resumes 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*(TOI).

```
f pL g
```

Continues all threads in the current process that are at the same PC as the *TOI*.

```
f t g
```

Continues a single thread.

RELATED TOPICS

Starting Processes and Threads in the *Classic TotalView User Guide*

dcont**Command**

dgpu_status

Manages GPU threads

Format

dcuda block [(Bx,By,Bz)]
dcuda thread [(Tx,Ty, Tz)]
dcuda kernel
dcuda device [<n>]
dcuda sm [<n>]
dcuda warp [<n>]
dcuda lane [<n>]
dcuda info-system
dcuda info-device
dcuda info-sm
dcuda info-warp
dcuda info-lane
dcuda focus (Bx,By,Bz),(Tx,Ty, Tz)
dcuda hwfocus <D/S/W/L>

Arguments

Bx, *By*, *Bz*

The x, y and z block indices

Tx, *Ty*, *Tz*

The x, y, and z thread indices

D/S/W/L

The coordinates defining the physical space of the hardware:

D: device number
S: streaming multiprocessor (SM)
W: warp (WP) number on the SM
L: lane (LN) number on the warp

Description

The **dcuda** commands allow you to manage and view GPU threads, in either the logical coordinate space of block and thread indices (<<<(Bx,By,Bz),(Tx,Ty,Tz)>>>) or the physical coordinate space that defines the hardware (the device number, the streaming multiprocessor number on the device, the warp number on the SM, and lane number on the warp).

dcuda block [(Bx,By,Bz)]

- With no arguments, shows the current CUDA block

- With a block argument of the form (*Bx,By,Bz*), changes the CUDA focus to that block. Parameters to the right (*By* and *Bz*, or just *Bz*) may be omitted; these are unchanged.

dcuda thread [(Tx,Ty,Tz)]

- With no arguments, shows the current CUDA thread.
- With a thread argument of the form (*Tx,Ty,Tz*), changes the CUDA focus to that thread. Parameters to the right (*Ty* and *Tz*, or just *Tz*) may be omitted; these are unchanged.

dcuda kernel

Displays the logical and hardware coordinates of the current CUDA context.

dcuda device [<n>]

- With no arguments, shows the current CUDA device.
- With a numeric argument, changes the CUDA device focus to that device.

dcuda sm [<n>]

- With no arguments, shows the current CUDA SM (streaming multiprocessor).
- With a numeric argument, changes the CUDA SM focus to that SM.

dcuda warp [<n>]

- With no arguments, shows the current CUDA warp.
- With a numeric argument, changes the CUDA warp focus to that warp.

dcuda lane [<n>]

- With no arguments, shows the current CUDA lane.
- With a numeric argument, changes the CUDA lane focus to that lane.

dcuda info-system

Displays the CUDA devices in the system.

dcuda info-device

Displays currently running SMs in the current device.

dcuda info-sm

Displays valid warps in the current SM.

dcuda info-warp

Displays valid lanes in the current warp.

dcuda info-lane

Displays the current lane.

dcuda focus (Bx,By, Bz),(Tx,Ty,Tz)

Changes the focus via CUDA logical coordinates of the form `<<<(Bx,By,Bz),(Tx,Ty,Tz)>>>`.

The following abbreviations are also accepted:

```
  <<<Tx>>>
  <<<(Tx)>>>
  <<<(Tx,Ty)>>>
  <<<(Tx,Ty,Tz)>>>
  <<<(Bx),(Tx)>>>
  <<<(Bx),(Tx,Ty)>>>
  <<<(Bx),(Tx,Ty,Tz)>>>
  <<<(Bx,By),(Tx)>>>
  <<<(Bx,By),(Tx,Ty)>>>
  <<<(Bx,By),(Tx,Ty,Tz)>>>
  <<<(Bx,By,Bz),(Tx)>>>
  <<<(Bx,By,Bz),(Tx,Ty)>>>
  <<<(Bx,By,Bz),(Tx,Ty,Tz)>>>
```

Angle brackets are optional, but must be balanced.

dcuda hwfocus <D/S/W/L>

Changes the focus via CUDA hardware coordinates of the form D/S/W/L, S/W/L, W/L, or L.

Command alias

Alias	Definition	Description
cuda	**dcuda**	Writes out the focus thread, as in **dcuda kernel**.

Examples

Displaying device information

```
dcuda info-device
```

Output:

```
  DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
  SM: 0/30 valid warps: 0x0000000000000001

dcuda info-sm
```

Output:

```
  DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
  SM: 0/30 valid warps: 0x0000000000000001
  WP: 0/32 valid/active/divergent lanes: 0x0000000f/0x0000000f/0x00000000 block:
  (0,0,0)

dcuda info-warp
```

Output:

```
  DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
  SM: 0/30 valid warps: 0x0000000000000001
  WP: 0/32 valid/active/divergent lanes: 0x0000000f/0x0000000f/0x00000000 block:
  (0,0,0)
  LN: 0/32 pc=0x000000001ef2efa8 thread: (0,0,0)
  LN: 1/32 pc=0x000000001ef2efa8 thread: (1,0,0)
  LN: 2/32 pc=0x000000001ef2efa8 thread: (0,1,0)
  LN: 3/32 pc=0x000000001ef2efa8 thread: (1,1,0)

dcuda info-lane
```

Output:

```
  DEV: 0/1 Device Type: gt200 SM Type: sm_13 SM/WP/LN: 30/32/32 Regs/LN: 128
  SM: 0/30 valid warps: 0x0000000000000001
  WP: 0/32 valid/active/divergent lanes: 0x0000000f/0x0000000f/0x00000000 block:
  (0,0,0)
```

Displaying the focus

```
dcuda warp sm
```

Output:

```
  sm 0 warp 0

dcuda lane device
```

Output:

```
  device 0 lane 3

dcuda thread
```

Output:

```
  thread (1,1,0)

dcuda kernel
```

Output:

```
  device 0, sm 0, warp 0, lane 3, block (0,0,0), thread (1,1,0)
```

Changing the focus

In these commands, note that TotalView assigns CUDA threads a negative thread ID. In the examples here, the CUDA thread is labeled "1.-1".

```
dcuda thread (1,1,0)
```

Changes the CUDA focus to the thread represented by logical coordinates 1,1,0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 3, block (0,0,0), thread
  (1,1,0)

dcuda lane 2
```

Changes the CUDA focus to lane 2.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 2, block (0,0,0), thread
  (0,1,0)

dcuda lane 1 sm 0
```

Changes the CUDA focus to lane 1 and to SM 0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 1, block (0,0,0), thread
  (1,0,0)

dcuda thread 0,0,0
```

Changes the CUDA focus to thread 0,0,0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 0, block (0,0,0), thread
  (0,0,0)

dcuda thread 1
```

Changes the CUDA focus to thread 1,0,0.

```
  New CUDA focus (1.-1): device 0, sm 0, warp 0, lane 1, block (0,0,0), thread
  (1,0,0)
```

RELATED TOPICS

Using the CUDA Debugger in the *TotalView User Guide*

dgroups

Manipulates and manages groups

Format

Adds members to thread and process groups

dgroups[**-add** | **-a**] [**-g** *gid*] [*id-list*]

Deletes groups

dgroups -delete[**-g** *gid*]

Intersects a group with a list of processes and threads

dgroups [**-intersect** | **-i**] [**-g***gid*] [*id-list*]

Prints process and thread group information

dgroups [**-list** | **-l**] [*pattern-list*]

Creates a new thread or process group

dgroups[**-new** | **-n**] [*thread* | *t* | *process* | *p*] [**-g** *gid*] [*id-list*]

Removes members from thread or process groups

dgroups [**-remove** | **-r**] [**-g** *gid*] [*id-list*]

Arguments

-g *gid*

The group ID on 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.

Description

The **dgroups**command supports the following functions:

- Adding members to process and thread groups.
- Creating a group.
- Intersecting a group with a set of processes and threads.

- Deleting groups.
- Displaying the name and contents of groups.
- Removing members from a group.

dgroups [-add | -a] [-g gid] [id-list]

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. Without an ***id-list*** argument, the CLI adds the threads and processes in the current focus. Similarly, you can name the group using the **-g**option. Without 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.

NOTE: 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 adds it only once.

You cannot use this command to add a process to a control group. Instead, use the **CGROUP(dpid)** variable; for example:

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

dgroups -delete [-g gid]

Deletes the target group. You can delete only groups that you create; you cannot delete groups that TotalView creates.

dgroups [-intersect | -i] [-g gid] [id-list]

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 | -l] [pattern-list]

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.

NOTE: If you do not enter a pattern, the CLI displays only groups that you have created with nonnumeric names.

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

dgroups [-new | -n] [thread | t | process | p] [-g gid] [id-list]

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 with 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.

NOTE: 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 | -r] [-g gid] [id-list]

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

```
dgroups -add
```

Adds the current focus thread to the current focus group.

```
dfocus tW gr -add
```

Adds the focus thread to its workers 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

```
dgroups -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.

```
dfocus tW dgroups -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
```

Displays information about all named groups; for example:

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

gr -l *
```

Displays 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

```
dgroups -new thread -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.

```
dfocus 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.

RELATED TOPICS

Setting and Creating Custom Groups in the *TotalView User Guide*

Groups in TotalView in the *TotalView User Guide*

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 (POI).

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.

RELATED TOPICS

Holding and Releasing Processes and Threads in the *TotalView User Guide*

The Processes and Threads View in the *TotalView User Guide*

dheap

Controls memory heap debugging behavior

Format

Shows Memory Debugger state

> **dheap** [**-status**]

Enables or disables the Memory Debugger

> **dheap** { **-enable** | **-disable** }

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 event notification

> **dheap -event_filter** *subcommands*

Writes memory information

> **dheap -export** *subcommands*

Specifies the filters the Memory Debugger uses

> **dheap -filter** *subcommands*

Displays or increments the generation number given to memory allocations

> **dheap -generation** [*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 *subcommands* [*start_address* [*end_address*]]

Displays or specifies the verbosity level for the Heap Interposition Agent () output

dheap -verbosity [*subcommands*]

Displays the Memory Debugger version number

dheap -version

Arguments

-status

Returns memory debugging status, reporting if a process is capable of having its heap operations traced, and whether TotalView will notify you if a notifiable heap event occurs. If TotalView stops a thread because one of these events occur, it displays information about this event.

Entering **dheap** with no arguments is the same as using this option.

-enable / -disable

The option **-enable** enables the so that heap events are recorded the next time you start the program. The **-disable** option disables the HIA the next time you start your program.

-apply_config { default | *filename* **}**

Applies configuration settings from a file to TotalView. If you type **default**, TotalView looks first in the current directory and then in your **.totalview/hia/** directory for a file named **default.hiarc**. Otherwise, it uses the name of the file entered here. If you do not specify an extension, TotalView assumes that the extension is **.hiarc**. That is, while you can specify a file named **foo.foobar**, you cannot specify a file named simply **foo** as TotalView would then assume that the file is actually named **foo.hiarc**.

-backtrace *subcommands*

Shows the current settings for the backtraces associated with a memory allocation. This information includes the *depth* and the *trim*, a s described below.

Subcommands

-status

Displays backtrace information. If you do not use other backtrace options, you can omit this option.

-set_depth *depth* / **-reset_depth**

Set or reset the *depth*. The depth is the maximum number of PCs that TotalView includes when it creates a backtrace. (The backtrace is created when your program allocates or reallocates a memory block.) The depth value starts after the trim value. That is, the number of excluded frames does not include the trimmed frames.

When you use the **-reset_depth** option, TotalView either restores its default setting or the setting you set using the **TVHEAP_ARGS** environment variable.

-set_trim *trim* / **-reset_trim**

Sets or resets the *trim*. The trim describes the number of PCs from the top of the stack that TotalView ignores when it creates a backtrace. As the backtrace includes procedure calls from within TotalView, setting a trim value removes them from the backtrace.

The default is to exclude TotalView procedures. Similarly, your program might call the heap manager from within library code. If you do not want to see call frames showing a library, you can exclude them.

When you use the **-reset_trim option**, TotalView either restores its default setting or the setting you set using the **TVHEAP_ARGS** environment variable.

-display *backtrace_id*

Displays the stack frames associated with the backtrace identified by ***backtrace_id***.

-compare *required_subcommands* [*optional_subcommands*]
[*process* | *filename* [*process* | *filename*]]

Required_Subcommands (both are required)

-data { **alloc** | **dealloc** | **hoard** | **leaks** }

Names the data to be written into the exported compare file, as follows:

alloc: heap allocations

dealloc: heap deallocations

hoard: deallocations currently held in the hoard

leaks: leaked heap allocations

-output { *directory* | *filename* }

Names the location for writing memory information. If the name ends with a slash (/), TotalView writes information into a directory, with the *filenames* being the TotalView defaults.

Optional_Subcommands

-reverse_diff

Changes the order in which TotalView makes its comparison. That is, TotalView normally compares the first named file to the second. This compares the second to the first.

-format { *html* | *txt* }

Specifies the format to use when it writes information to disk. If you do not use this command, TotalView writes the information as HTML.

process | *filename* [*process* | *filename*]

Specifies if the comparison uses a process or a memory debugging export (**.mdbg**) file. Your choices are:

No arguments: Compare the two processes in the current focus.

One argument: Compare the process in the current focus with the *process/filename* you specify.

Two arguments: Compare the two *processes/filenames* named as arguments.

-event_filter *subcommands*

These subcommands control which HIA events cause TotalView to stop program execution.

Subcommands

-status (optional)

Displays the current event filter settings. If you do not use other event filter options, you can omit this option.

-set { **on** | **off** }

Enables or disables event filtering. If you disable event filtering, TotalView displays all events. If you enable event filtering, then you can control which events TotalView displays.

-show_supported

Lists the set of events that is supported by the HIA in use. Any of these events can be specified in a space-delimited list as the argument to the **-notify**, **-nonotify**, and **-reset_notify** options.

-reset

Resets the event filter to TotalView's default value. You can create your own default in a configuration file or by specifying an environment variable setting.

-[no]notify *event-list*

Enables or disables one or more events. Use the **-show_supported** command to list the events supported by the HIA in use.

-reset_notify *event-list*

Resets the event filter to TotalView's default value for the filters named in the list. Use the **-show_supported** command to list the events supported by the HIA in use.

-export *required_subcommands* [*optional_subcommands*]

Writes information to a file.

Required_Subcommands

-data { **alloc** | **alloc_leaks** | **dealloc** | **hoard** | **leaks** | **raw** }

Specifies the data being written, as follows:

alloc: Shows all heap allocations.

alloc_leaks: Shows all heap allocations and perform leak detection. This differs from the alloc argument in that TotalView annotates leaked allocations.

dealloc: Shows deallocation data.

hoard: Shows deallocations currently held in the hoard.

leaks: Shows leaked heap allocations.

raw: Exports all heap information for the process using .mdbg format. This format's only purpose is to be imported back into TotalView.

-output *filename*

Names the file to which TotalView writes memory information.

Optional_Subcommands

You can optionally use any of the following options with **dheap -export**:

-[no]check_interior

When the **no** prefix is omitted, a memory block is not considered as leaked if a pointer is pointing anywhere within the block. TotalView ignores this option unless you also use the **-data leaks** option.

-format { html| text }

Specifies the format used to write a view's data to a file. The default is **html**.

There are some limitations regarding the **html** format. The only supported browser is Firefox, running versions 1.0 or greater. In addition, you must have Javascript turned on. (While information displays correctly in other browsers such as Internet Explorer 6.0 and Safari, isolated problems can occur.) This file can be quite large and can take a while to load.

When TotalView writes an HTML file, it also creates a subdirectory with a related name containing additional information. For example, if the output file name is **foo.html**, TotalView creates a subdirectory named **foo_files**.

If you need to move the output file, you must also move the related subdirectory to the same directory.

-relative_to_baseline

If used, TotalView limits the information it writes to those created since the last time you created a baseline. If you also use the **-data raw** option, TotalView ignores this option.

-set_show_backtraces { on | off }

When set to **on**, TotalView includes backtrace information within the data being written. As **on** is the default, you only need to use this option with the **off** argument.

-set_show_code { on | off }

When set to **on**, TotalView includes the source code for the place where the memory was allocated with the data being written. As **on** is the default, you only need to use this option with the **off** argument.

-view backtrace

Exports a backtrace view instead of a source view. If you also use the **-data raw** option, TotalView ignores this option.

-filter *subcommands*

Use the **-filter** options to enable, disable, and show information about filtering.

Subcommands

-enable [*filter-name-list* | **all**]

Enables filtering of **dheap** commands. If you do not use an argument with this option, this option is equivalent to selecting **Enable Filtering** in the TotalView UI.

Using a filter name simply defines where to locate filter information; you still need to enable filtering. For example, here is how you would enable filtering and enable the use of a filter named **MyFilter**:

```
dheap -filter -enable MyFilter
dheap -filter -enable
```

If you did not enter the second command, no filtering would occur.

-disable [*filter-name-list* | **all**]

Disables filtering or disables an individual filter. The way that you use this command is similar to **dheap -filter -enable**.

-list [[**-full**] *filter-name-list*]

Displays a filter description and its enabled state. If you do not use a *filter-name-list* argument, the CLI displays all defined filters and their enabled states.

If you include the **full** argument, the information includes all of the filter's criteria.

-generation [**-status** | **-increment**]

The HIA gives each allocation a generation number. The **-status** option displays the current value of the generation number. This is the default option.

The **-increment** option increments the generation number in the HIA.

-guard *subcommands* [*start_address* [*end_address*]]

Use the**-guard** options to enable, disable, set characteristics, and show information about guard blocks.

Subcommands

-status

Displays guard settings. If you do not use other guard options, you can omit the **-status** option when you want to see status information.

-check [*subcommands*]

Checks the guards to see if they have been violated. If it finds a violated guard, TotalView displays information. The information displayed can be modified through the following subcommands:

-[no]show_backtrace: Displays (or not) backtrace information. This list can be very long.

-[no]show_backtrace_id: Displays (or not) backtrace IDs.

-[no]show_generation_id: Displays (or not) generation IDs.

-[no]show_guard_settings: Displays (or not) information about guards settings.

-set { **on** | **off** }

Enables or disables the writing of guards. If you disable this feature after it is enabled, TotalView does not remove existing guard blocks.

-reset

Resets the guards to the TotalView's default values. You can create your own defaults in a configuration file or by specifying an environment variable setting.

-reset_max_size / -reset_post_pattern / -reset_pre_pattern / -reset_post_size / -reset_pre_size

Removes all changes you have made and restores guard settings to what they were when you first invoked TotalView.

-set_max_size *size* **/ -set_post_size** *size* **/ -set_pre_size** *size*

Specify a size in bytes. You can set the sizes of the pre- and post- guards independently. The actual size of a guard can differ from these settings if TotalView needs to adjust the size to meet alignment and allocation unit size constraints. In addition, you can set the maximum guard size. If the guard size will be greater than this maximum, TotalView does not create a guard.

The default sizes are 8 bytes.

A maximum size of zero (0) does not limit guard sizes. Zero is the default value.

-set_post_pattern *pattern* **/ -set_pre_pattern** *pattern*

Defines the pattern TotalView uses when it writes guard blocks.The default pre-allocation pattern is **0x77777777** and the default post-allocation pattern is **0x99999999**.

start_address

If you only specify a ***start_address***, TotalView either tags or removes the tag from the block that contains this address. The action it performs depends on the subcommand you use.

end_address

If you also specify an ***end_address***, TotalView either tags all blocks beginning with the block containing the ***start_address*** and ending with the block containing the ***end_address***, or removes the tag. The action it performs depends on the subcommand you use. If ***end_address*** is 0 (zero) or you do not specify an ***end_address***, TotalView tags or removes the tag from all addresses beginning with ***start_address*** to the end of the heap.

-hoard [*subcommands*]

Do not surrender allocated blocks back to your program's heap manager. If you do not specify a subcommand, TotalView displays information about the hoarded blocks.

Subcommands

-status

Displays hoard settings. Information displayed indicates whether hoarding is enabled, whether deallocated blocks are added to the hoard (or only those that are tagged), the maximum size of the hoard, and the hoard's current size.If you do not use other hoarding options, you can omit the **-status** option when you want to see status information.

-display [*start_address* [*end_address*]]

Displays the contents of the hoard. You can restrict the display by specifying ***start_address*** and ***end_address***. If you omit ***end_address*** or use a value of 0, TotalView displays all contents beginning at ***start_address*** and going to the end of the hoard.

The CLI displays hoarded blocks in the order in which your program deallocated them.

-set { **on** | **off** }

Enables and disables hoarding.

-reset

Resets TotalView settings for hoarding back to their initial values.

-set_all_deallocs { **on** | **off** }

Determines whether to hoard deallocated blocks.

-reset_all_deallocs

Resets TotalView settings for hoarding of deallocated blocks to their initial values.

-set_max_kb *num_kb*

Sets the maximum size of the hoarded information.

-set_max_blocks *num_blocks*

Sets the maximum number of hoarded blocks.

-reset_max_kb / -reset_max_blocks

Resets a hoarding size value back to its default.

-autoshrink [*subcommands*]

The autoshrink feature attemps to avoid the failure of memory allocations because memory is running short by reducing the size of the hoard to free enough memory for the allocation. If hoarding is enabled and this feature is turned on, blocks are removed from the hoard until either there is sufficient memory for the allocation, or the hoard is exhausted. If the hoard is exhausted and the allocation still fails, the normal "allocation operation returned null" event is raised.

There are subcommands to control this feature, as follows:

-status: Displays information about the current status of autoshrinking.

-set { **on** | **off** }: Turns the feature on and off.

-reset: Resets autoshrinking to its default values, obtained from the **TVHEAP_ARGS** environment variable, the HIA configuration file, or the TotalView default values.

-set_threshold_kb *integer* | **-reset_threshold_kb**: Defines a size in kilobytes for the hoard such that if autoshrinking causes the hoard to fall below this value, you are notified. This can be a useful way to know when memory is running short. Use the reset option to return this setting to its default value.

-set_threshold_trigger *integer* | **-reset_threshold_trigger**: If you set a threshold, it can happen that the size of the hoard starts crossing over and under the threshold size again and again, resulting in continuous notifications. This option sets a limit to the number of notifications by decrementing the specified number each time a notification occurs until the number reaches zero, at which time notifications stop. To start them again, use this option to set a new number. The reset option resets the default value, which normally is 1, meaning you receive just a single notification and then no more.

-info [*subcommands*] [**-generation x:y**] [*start_address* [*end_address*]]

Displays information about the heap or regions of the heap within a range of addresses. If you do not use the address arguments, the CLI displays information about all heap allocations.

The information that TotalView displays includes the start address, a block's length, and other information such as flags or attributes.

Subcommands

-[no]show_backtrace

Displays (or not) backtrace information. This list can be very long.

-[no]show_backtrace_id

Displays (or not) backtrace IDs.

-[no]show_generation_id

Displays (or not) information about generation IDs.

-[no]show_guard_settings

Displays (or not) information about guards settings.

-generation x:y

Limits the reporting to leaked heap regions with a HIA generation ID satisfying the range condition **x:y**. The range condition is specified as follows:

```
Specifier Condition
x:y x <= id <= y
x x <= id
x: x <= id
x:0 x <= id
:y 1 <= id <= y
```

where **id** identifies the HIA generation of the heap region, and **x** and **y** are positive integers.

start_address

If you just type a ***start_address***, the CLI reports on all allocations beginning at and following this address. If you also type an ***end_address***, the CLI limits the display to those allocations between the ***start_address*** and the ***end_address***.

end_address

If you also specify an ***end_address***, the CLI reports on all allocations between ***start_address*** and ***end_address***. If you type 0, it's the same as omitting this argument; that is, TotalView displays information from the ***start_address*** to the end of the address space.

-is_dangling *address*

Indicates if an ***address*** that was once allocated and not yet recycled by the heap manager is now deallocated.

-leaks [*subcommands*]

Locates all memory blocks that your program has allocated and that are no longer referenced. That is, using this command tells TotalView to locate all dangling memory.

If neither of the subcommands **-check_interior** and **-no_check_interior** are specified, the default behavior is based on the TotalView variable **TV::GUI::leak_check_interior_pointers**, whose default value is **true**.

A leak report is generated as a result of the command. The report shows the total number of leaks and total bytes leaked for the processes. It also consolidates leaks occurring at the same lines and reports the total number of leaks and total bytes leaked. Some additional statistics such as the smallest, largest and average leak size are also displayed.

Subcommands

-check_interior

TotalView considers a memory block as being referenced if the beginning or an interior portion of the block is referenced.

-no_check_interior

TotalView considers a memory block as being referenced only if a reference points to the beginning of the allocated block.

-generation x:y

Limits the reporting to leaked heap regions with a HIA generation ID satisfying the range condition **x:y**. The range condition is specified as follows:

```
Specifier Condition
x:y x <= id <= y
x x <= id
x: x <= id
x:0 x <= id
:y 1 <= id <= y
```

where **id** identifies the HIA generation of the heap region, and **x** and **y** are positive integers.

-[no]notify

Using the**-notify** option stops your program's execution when TotalView detects a notifiable event, and then prints a message (or display a dialog box if you are also using the UI) that explains what just occurred. TotalView can notify you when heap memory errors occur or when your program deallocates or reallocates tagged blocks.

The **-nonotify** option does not stop execution. Even if you specify the **-nonotify** option, TotalView tracks heap events.

-paint [*subcommands*]

The painting feature fills, or paints, blocks as they are allocated and deallocated. The pattern used to fill the blocks may be specified, and different patterns may be used for allocations and deallocations.

Painting is useful in cases where it is suspected that the application is not initializing memory it acquires from the heap manager before using it. The allocation pattern can be set to something easily recognizable, and to something that may provoke an error if the memory is used before it is initialized. For example, if the memory is being used for floating point numbers, the pattern could be set to something that is not a legal floating point number. Should an element in the block be used in a floating point operation without being initialized, a floating point error should be raised. Similarly, certain “use-after-free” errors can be found by using a deallocation pattern.

Subcommands

-status

Shows the current paint settings. These are either the values you set using other painting options or their default values. This is the default behavior if **-paint** is entered without arguments.

-set_alloc {on | off } / -set_dealloc { on | off } / -set_zalloc { on | off }

Controls block painting. When set to **on**, TotalView paints a block when your program’s heap manager allocates, deallocates, or uses a memory function that sets memory blocks to zero.

You can only paint zero-allocated blocks if you are also painting regular allocations.

The **off** options disable block painting.

-reset_alloc / -reset_dealloc / -reset_zalloc

Resets TotalView settings for block painting to their initial values or to values specified in a startup file.

-set_alloc_pattern *pattern* **/ -set_dealloc_pattern** *pattern*

Sets the pattern that TotalView uses the next time it paints a block of memory. The maximum width of *pattern* can differ between operating systems. However, your pattern can be shorter.

-reset_alloc_pattern / -reset_dealloc_pattern

Resets the patterns used when TotalView paints memory to the default values.

-red_zones [*subcommands*]

The Red Zones feature help catch bounds errors and use-after-free errors. The basic idea is that each allocation is placed in its own page. An allocation is positioned so that if an overrun, that is, an access beyond the end of the allocation, is to be detected, the end of allocation corresponds to the end of the page.

The page following that in which the allocation lies is also allocated, though access to this page is disabled. This page is termed the fence. Should the application attempt to access a location beyond the end of the allocation, that is, in the fence, the operating system sends the target a segment violation signal. This is caught by a signal handler installed by the HIA. The HIA examines the address that caused the violation. If it lies in the fence, then the HIA raises an overrun bounds error using the normal event mechanism.

If, however, the address does not lie in any region the HIA `owns', then the HIA attempts to replicate what would have happened if the HIA's signal handler were not in place. If the application had installed a signal handler, then this handler is called. Otherwise, the HIA attempts to perform the default action for the signal. It should be clear from this that the HIA needs to interpose the signals API to ensure that it always remains the installed handler as far as the operating system is concerned. At the same time, it needs to present the application with what it expects.

Underruns, or errors where the application attempts to read before the start of an allocation are handled in a similar way. Here, though, the allocation is positioned so that its start lies at the start of the page, and fence is positioned to precede the allocation.

One complication that arises concerns overrun detection. The architecture or definition of the allocation routines may require that certain addresses conform to alignment constraints. As a consequence, there may be a conflict between ensuring that the allocation's start address has the correct alignment, and ensuring that the allocation ends at the end of the page.

Use-after-free errors can also be detected. In this case, when the block is deallocated, the pages are not returned to the operating system. Instead, the HIA changes the state of the allocation's table entry to indicate that it's now in the deallocated state, and then disables access to the page in which the allocation lies. This time, should the application attempt to access the block now that it's been deallocated, a signal will be raised. Again, the HIA examines the faulting address to see what it knows about the address, and then either raises an appropriate event for TV, or forwards the signal on.

The key features that distinguishes Red Zones is that it can be engaged and disengaged at will during the course of the target's execution. The settings can be adjusted, so that new allocations have different properties from existing allocations. Red Zones can be turned on or off, so that some of the application's requests are satisfied by the Red Zones allocator, and others by the standard heap manager. The HIA keeps track of which allocator is responsible for, or owns, each block.

Note that **-rz** is an alias for **-red_zones**.

Subcommands

-status [**-all**]

Shows the current HIA red zone settings. By default, **dheap -red_zones** displays only those settings that can vary in the current mode, so that, for example, in overrun mode the settings for fences and end positioning are not shown. The **dheap -red_zones -status -all** command causes all settings to be shown, including those that are overridden for the current mode.

-stats [*start_addr* [*end_addr*]]

Displays statistics relating to the HIA's Red Zones allocator for the optionally specified address range. If no range is specified, the following statistics are shown for the entire address space:

Number of allocated blocks.

Sum of the space requests received by the Red Zones allocator for allocated blocks.

Sum of the space used for fences for allocated blocks.

Overall space used for allocated blocks.

The same set of statistics are also shown for deallocated blocks. In addition, the space used for each category is shown as a percentage of the overall space used for Red Zones.

-info [*start_addr* [*end_addr*]]

Displays the Red Zone entries for allocations (and deallocations) lying in the optionally specified range. If no range is specified, the entries for the entire address space are displayed.

-set { **on** | **off** } / **-reset**

Enables or disables Red Zones. The **-reset** option allows the HIA to determine its setting using the usual rules.

-set_mode { **overrun** | **underrun** | **unfenced** | **manual** }

Sets the HIA in one of several Red Zone modes. When a new allocation is requested, the HIA will override the actual settings for some of the individual controls, and will instead use values that correspond to that mode. The settings that are affected are: pre-fence, post-fence, and end-positioning. The other settings, like use-after-free, exit value, and alignment, take their values from the actual settings of those controls.The modes are:

overrun: The settings used are those that allow overruns to be detected. These are: no for pre-fence, yes for post-fence, and yes for end-positioned.

underrun: The settings used are those that allow underruns to be detected. These are: yes for pre-fence, no for post-fence, and no for end-positioned.

unfenced: The settings used are those that allow use_after_frees to be detected. These are: no for pre-fence, no for post-fence. End-positioned is determined from the control's setting.

manual: All settings are determined from their actual values.

-set_pre_fence { **on** | **off** }

Enables or disables the pre-fence control. However, the setting is ignored unless the mode is manual.

-set_post_fence { **on** | **off** }

Enables or disables the post-fence control. However, the setting is ignored unless the mode is manual.

-set_use_after_free { **on** | **off** }

Enables or disables the use_after_free control. If enabled, any subsequent allocations will be tagged such that the allocation and its fences are retained when the block is deallocated. Access to the block is disabled when it is deallocated to allow attempts to access the block to be detected.

-set_alignment *integer*

Regulates the alignment of the start address of a block issued by the Red Zones allocator. An alignment of 0 indicates that the default alignment for the platform should be used. An alignment of 2 ensures that any address returned by the Red Zones allocator is a multiple of two. In this case, if the length of the block is odd, the end of the block will not line up with the end of the page containing the allocation. An alignment of 1 is necessary for the end of the block to always correspond to the end of the page.

-set_fence_size *integer*

Adjusts the fence size used by Red Zones. A fence size of 0 indicates that the default fence size of one page should be used. If necessary, the fence size is rounded up to the next multiple of the page size. In most cases it should not be necessary to adjust this control. One instance where it may be useful, however, is where it is suspected that a bounds error is a consequence of a badly coded loop, and the stride of the loop is large. In such a case, a larger fence may be helpful.

-set_end_aligned { **on** | **off** }

Controls whether the allocation is positioned at the end or the start of the containing page. The control in the HIA is always updated, though the actual value is ignored in overrun and underrun modes.

-set_exit_value *integer*

Adjusts the exit value used if the HIA terminates the target following detection of a Red Zones error. Generally, the application fails if it is allowed to continue after a Red Zone error has been detected. In order to allow some control over the application's exit code, the HIA will call exit when an error is detected. The value it passes to exit as a termination code can be controlled so that if the application is run from scripts, the cause for the termination can be determined.

-size_ranges [*subcommands*]

Restricts the use of Red Zones to allocations of particular sizes. With size ranges enabled, the Red Zones allocator is used if the size of the request lies in one of the defined ranges. A value is deemed to lie in a range if start <= size <= end.Note that **-sr** is an alias to **-size_ranges**. A range having an end of 0 is interpreted as having no upper limit. Thus if the end is 0, the size matches the range if it is at least as large as the start.This feature allows the HIA to enable Red Zones for specific allocation sizes. The Red Zones allocator will be used if the size of the request lies in any one of these ranges. The HIA *does not* check to see that ranges do not overlap or are otherwise consistent.The determination of whether the Red Zones allocator should be used is made at the time of the original allocation. Thus, once an allocator has taken ownership of a block, that allocator will be used for the remainder of the block's life. In particular, all realloc operations will be handled by the same allocator, irrespective of the size range settings at the time of reallocation.There are two attributes associated with each range. The first is the “in_use” attribute. This is ignored by the HIA, and is provided for the benefit of TotalView. The motivation here is to allow TotalView to keep a state that would otherwise be lost if the target is detached, and then reattached to later.The second attribute is the “active” attribute. This indicates if the size range is active, and therefore whether it is used by the HIA when determining whether the Red Zones allocator should be used.

Subcommands

-set { **on** | **off** }

Enables and disables size ranges. If size ranges are disabled, but Red Zones are enabled, the Red Zones allocator is used for all allocations.

-reset

Unsets the TotalView setting for the enable/disable control.

-status [**-all**] *id_range*

Shows the current settings of the size ranges. The absence of an ***id_range*** is equivalent to an ***id_range*** of **0:0**" By default, only "in_use" size ranges are displayed. To display all known ranges, specify **-all**. ***id_range*** must be in one of the following formats:

x:y = IDs from x to y
:y = IDs from 1 to y
x: = IDs of x and higher
x = ID is x

-set_range *id size_range*

Sets a size range identified by ***id*** to a particular size range. ***size_range*** must be in one of the following formats:

x:y = allocations from x to y
:y = allocations from 1 to y
x: = allocations of x and higher
x = allocation is x

-reset_range *id_range*

Resets an id or range of ids. For ***id_range*** formats, see **-status** above.

-set_in_use { **on** | **off** } *id_range*

Adjusts the "in_use" attribute of all the size ranges whose IDs lie within ***id_range***. For ***id_range*** formats, see **-status** above.

-set_active { **on** | **off** } *id_range*

Adjusts the "active" attribute of all the size ranges whose ids lie within ***id_range***. For ***id_range*** formats, see **-status** above.

-reset_mode / -reset_pre_fence / -reset_post_fence / -reset_use_after_free / -reset_alignment / -reset_fence_size / -reset_exit_value / -reset_end_aligned

Unsets the TotalView settings for the above controls.

-tag_alloc *subcommand* [*start_address* [*end_address*]]

Marks a block so that it can notify you when your program deallocates or reallocates a memory block.

When tagging memory, if you do not specify address arguments, TotalView either tags all allocated blocks or removes the tag from all tagged blocks.

Subcommands

-[no]hoard_on_dealloc

Does not release tagged memory back to your program's heap manager for reuse when it is deallocated. This is used in conjunction with hoarding. To re-enable memory reuse, use the **-no-hoard_on_dealloc** subcommand.

If you use this option, the memory tracker only hoards tagged blocks. In contrast, if you use the **dheap -hoard -set_all_deallocs** on command, TotalView hoards all deallocated blocks.

-[no]notify_dealloc / -[no]notify_realloc

Enable or disable notification when your program deallocates or reallocates a memory block.

start_address

If you only specify a ***start_address***, TotalView either tags or removes the tag from the block that contains this address. The action it performs depends on the subcommand you use.

end_address

If you also specify an ***end_address***, TotalView either tags all blocks beginning with the block containing the ***start_address*** and ending with the block containing the ***end_address***, or removes the tag. The action it performs depends on the subcommand you use. If end_address is 0 (zero) or you do not specify an ***end_address***, TotalView tags or removes the tag from all addresses beginning with ***start_address*** to the end of the heap.

-verbosity [*subcommands*]

The subcommands to this option let you control how much information TotalView displays as it executes.

Subcommands

-status

Displays the current verbosity setting. This is the default if no arguments are specified.

-reset

Restores the verbosity setting to its default.

-set *verbosity*

Controls how much output TotalView writes to its output file. By default, this is file descriptor 1. Higher verbosity values tell TotalView to write more information. Setting ***verbosity*** to zero (0) suppresses output.

-version

Displays the version number of the HIA. If it is available, the distribution number of the version of TotalView with which the HIA was released is also shown.

Description

The **dheap** command controls memory debugging from the command line. For full information on memory debugging in TotalView, see Memory Debugging in the *TotalView User Guide*.

Here are some of the things you can do when debugging memory problems:

- Use the Heap Interposition Agent () to track memory errors.
- Stop execution when a free() error occurs, and display information you need to analyze the error.
- Hoard freed memory so that it is not released to the heap manager.
- Write heap information to a file.
- Control how much information is written to displays.
- Use guard blocks. After TotalView writes guard blocks, you can run a report to see if blocks are violated.
- Detect leaked memory by analyzing whether a memory block is reachable.
- Compare memory states. You can compare the current state against a saved state or compare two saved states.
- Paint memory with a bit pattern when your program allocate and deallocates it.
- Receive notification when your program deallocates or reallocates a memory block.

The first step when debugging memory problems is to type the **dheap -enable** command. This command activates TotalView. You must do this before your program begins executing. If you try to do this after execution starts, TotalView tells you that it will enable TotalView when you restart your program. For example:

```
d1.<> n
64 > int num_reds = 15;
d1.<> dheap -enable
process 1 (30100): This will only take effect on restart
```

You can tell TotalView to stop execution if:

- A free() problem occurs (**dheap -notify**)
- A block is deallocated (**dheap -tag_alloc -notify_dealloc**)
- A block is reallocated (**dheap -tag_alloc -notify_realloc**)

If you enable notification, TotalView stops the process when it detects one of these events. TotalView is always monitoring heap events even if you turned notification off, but TotalView does not stop the program when events occur or tell you that the events occurred.

While you can separately enable and disable notification in any group, process, or thread, you probably want to activate notification only on the control group's master process. Because this is the only process that TotalView creates, it is the only process where TotalView can control TotalView's environment variable. For example, slave processes are normally created by an MPI starter process or as a result of using the **fork()** and **exec()** functions. In these cases, TotalView simply attaches to them. For more information, see "Preparing Programs for Memory Debugging", in the *TotalView User Guide*.

If you do not use a **dheap** subcommand, the CLI displays memory status information. You need to use the **-status** option only when you want the CLI to display status information in addition to doing something else.

The information that the **dheap** command displays can contain a flag containing additional information about the memory location. The following table describes these flags:

Flag Value	Meaning
0x0001	Operation in progress
0x0002	notify_dealloc: you will be notified if the block is deallocated
0x0004	notify_realloc: you will be notified if the block is reallocated
0x0008	paint_on_dealloc: TotalView will paint the block when it is deallocated
0x0010	dont_free_on_dealloc: TotalView will not free the tagged block when it is deallocated
0x0020	hoarded: TotalView is hoarding the block

Examples

The following example shows a scenario of finding and debugging a memory problem with **dheap**.

```
d1.<> dheap
process: Enable Notify Available
1 (18993): yes yes yes
1.1 realloc: Address does not match any allocated block.: 0xbfffd87c

d1.<> dheap -info -backtrace
process 1 (18993):
0x8049e88 -- 0x8049e98 0x10 [ 16]
flags: 0x0 (none)
: realloc PC=0x400217e5 [/.../malloc_wrappers_dlopen.c]
: argz_append PC=0x401ae025 [/lib/i686/libc.so.6]
: __newlocale PC=0x4014b3c7 [/lib/i686/libc.so.6]
:
...
.../malloc_wrappers_dlopen.c]
: main PC=0x080487c4 [../realloc_prob.c]
: __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
: _start PC=0x08048621 [/.../realloc_prob]


0x8049f18 -- 0x8049f3a 0x22 [ 34]
flags: 0x0 (none)
: realloc PC=0x400217e5 [/.../malloc_wrappers_dlopen.c]
: main PC=0x0804883e [../realloc_prob.c]
: __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
: _start PC=0x08048621 [/.../realloc_prob]
```

Here is an example of a reported free error:

```
d1.<> dheap
process: Enable Notify Available
1 (30420): yes yes yes
1.1 free: Address is not the start of any allocated block.:
free: existing allocated block:
free: start=0x08049b00 length=(17 [0x11])
free: flags: 0x0 (none)
free: malloc PC=0x40021739 [/.../malloc_wrappers_dlopen.c]
free: main PC=0x0804871b [../free_prob.c]
free: __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
free: _start PC=0x080485e1 [/.../free_prob]

free: address passed to heap manager: 0x08049b08
```

dhistory

Performs actions upon ReplayEngine

Format

Enable or disable recording mode

dhistory { **-enable** | **-disable** }

Get information about the current state of Replay

dhistory -info

Create a bookmark so you can return to a point in the execution history. The command returns an ID for referencing the bookmark.

dhistory { **-create_bookmark** [*comment*]|**-cb** [*comment*] }

Go to a bookmark

dhistory { **-goto_bookmark** *ID* | **-gb** *ID* }

Return to the live execution point, that is, the end of the current recording, and continue recording

dhistory -go_live

List the bookmarks currently set, with IDs and comments

dhistory { **-show_bookmarks** | **-sb** }

Remove a bookmark, or all bookmarks

dhistory { { **-delete_bookmark** *ID* | **-db** *ID* }| **-clear_bookmarks** }

Save a recording file

dhistory -save [*recording-file*]

Deprecated arguments for setting and going to a bookmark (use the new 'bookmark' arguments)

dhistory { **-get_time** | **-go_time** *time* }

Arguments

-enable

Enables Replay immediately. Once replay is enabled and recording has started, it cannot be disabled until restart.

-disable

Disables Replay for next restart. Once enabled, replay cannot be disabled for a live process.

-info

Displays ReplayEngine information including the current time, the live time, and whether the process is in Replay or Record mode. If you enter **dhistory** without arguments, **-info** is the default.

-create_bookmark *comment*

Creates a Replay bookmark at the current execution location so you can return to it later. You can specify an optional comment to this command and it will be stored with the bookmark for display when you use the `show_bookmarks` command. A bookmark is created with a unique numeric ID, which is the return value.

-goto_bookmark *ID*

Goes to the bookmark with the specified ID. This returns the focus process to the execution location where the bookmark was first created.

-go_live

Returns the process to the PC and back into Record mode. You can resume your "regular" debugging session.

-show_bookmarks

Displays all Replay bookmarks. This command shows the bookmark ID along with information about what line number, PC and function the bookmark is on. If you added a comment to help you remember the significance of the bookmark, it displays this as well.

-delete_bookmark *ID*

Deletes the bookmark with the given ID.

-clear_bookmarks

Deletes all Replay bookmarks.

-save *recording-file*

Saves the current replay history to a file. There is an optional argument to specify the name of the file to save to. The file specification can be a path or a simple file name, in which case it is saved in the current working directory. If no file is specified, the recording is saved in the current working directory with the file name `replay_`pid_hostname`.recording`.

To reload the recording file, use one of the following commands based on the functionality for loading core files. TotalView recognizes the recording file for what it is and acts appropriately.

To reload a recording at startup:

totalview *executable recording-file*

To reload a recording file when TotalView is running:

dattach *filename* **-c** *recording-file*

The *recording-file* argument can be either a path or a simple file name, in which case the current working directory is assumed.

-get_time — deprecated: use **create_bookmark**

Returns an integer value representing the program execution location at the current time. The 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.

-go_time *time* — deprecated: use **goto_bookmark**

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 code or for searching execution history. It may need to be combined with stepping and **duntil**commands to return to an exact position.

Description

The **dhistory** command displays information about the current process either by default or when using the **-info**argument. In addition, options to this command can obtain a debugging time, which can be stored in a variable to go back to a particular time.

In addition, you can enable and display 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**.

Command alias

Alias	Definition	Description
replay	**dhistory**	Performs actions upon ReplayEngine.

Examples

```
dhistory [-info]
```

Typing `dhistory` with no arguments or with the `-info` argument 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

replay -create_bookmark "This is where the crash occurs"
3
```

Creates a bookmark at the current execution location and returns an ID. The comment appears in the list of bookmarks displayed with `-show_bookmarks` (see below). Note the use of the `replay` alias for this command, which might be easier to remember than `dhistory`.

```

replay -show_bookmarks
```

Displays a list of the currently defined bookmarks:

```
bookmark: 1: pc: 0x004005df, function: main, line: 59, comment:
bookmark: 2: pc: 0x004006b6, function: main, line: 69, comment:
bookmark: 3: pc: 0x004006fb, function: main, line: 75, comment: This is where the
crash occurs

replay -delete_bookmark 2
deleted bookmark: 2
```

Deletes the bookmark with the given ID, and returns a confirmation of the deleted bookmark.

dhold

Holds threads or processes

Format

Holds processes

dhold -process

Holds threads

dhold -thread

Arguments

-process

Holds processes in the current focus. Can be abbreviated to **-p**.

-thread

Holds threads in the current focus. Can be abbreviated to **-t**.

Description

The **dhold** command holds the threads or processes in the current focus. With "**-thread**", the threads in the thread of interest (TOI) are held. With "**-process**", the processes in the focus set are held.

NOTE: 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**.

RELATED TOPICS

Holding and Releasing Processes and Threads in the *Classic TotalView User Guide*

dunhold**Command**

dkill

Terminates execution of processes

Format

dkill [-remove]

Arguments

-remove

Removes 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 restart only the initial process.

If you do not indicate a focus, the default focus is the process of interest (POI). 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.

RELATED TOPICS

Starting Your Program in the *TotalView User Guide*

Restarting and Deleting Programs in the *Classic TotalView User Guide*

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. This 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 CLI 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.

RELATED TOPICS

dset **Command**

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 following 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, 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 the *Classic TotalView User Guide*.) The CLI creates defaults if you omit parts of this specification.

If you enter a different file, it is used for future display. 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 48. If you name more than one address, TotalView picks one.

-e

Sets the display location to include the current execution point of the *thread of interest* (TOI). 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 source code location, 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.

Tabs in the source file are expanded as blanks in the output. The TAB_WIDTH variable controls the tab stop width, which defaults to 8. If **TAB_WIDTH** is set to **-**1, no tab processing is performed, 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 to be the 20th 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*]
[-list_reverse_connect]
[-reject_reverse_connect [*ID* | **all**]]
[-accept_reverse_connect [*ID*]]

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.)

{ **-np** | **-procs** | **-tasks** } *num*

Indicates the number of processes or tasks that the starter program creates.

-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. To enable, the feature must be supported and licensed on the current platform.

-starter_args *argument*

Indicates additional arguments to be passed to the starter program.

-env *variable=value*

Sets a variable that is added to the program's environment.

Adds, changes, or removes an environment variable in the target process. A target process inherits its environment from its parent process, but this option allows you to modify the environment passed to target processes created by the debugger. If the **variable** does not exist in the inherited environment, it is inserted with the given **value**. If the **variable** already exists in the inherited environment, it is replaced with the given **value**. In either case, **value** can be an empty string. If the string contains no equal sign, then **variable** is removed from the inherited environment.

Multiple **-env** options may be specified.

-e

Indicates that the next argument is an executable file name. You need to use **-e** if the executable name begins with a dash (-) or consists of only numeric characters. Otherwise, just provide the executable file name.

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 attach to when an MPI job is created or attached to. The list is space-separated; each element can have one of three 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 must be either a positive decimal integer or **max** (the last rank in the MPI job).

A *subset_specification* that is the empty string (**""**) is equivalent to **0-max**.

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,...

-list_reverse_connect

Lists all the available reverse connect requests that are found. Each request is preceded by the reverse connect ID.

-reject_reverse_connect [*ID* | **all**]

Rejects a reverse connection.

If an ID is specified, then that specific request is rejected. The waiting back end initiating the request will be terminated. If "all" is specified, then all requests found will be rejected. If no ID is specified, the first request seen will be rejected.

-accept_reverse_connect [*ID*]

Accepts a reverse connection.

If an ID is specified, then that specific request is accepted. If no ID is specified, the first request seen will be accepted.

Description

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

NOTE: Your license may limit the number of processes that you can run at the same time. For specifics regarding your license, see the *TotalView Installation and Licensing Guide*.

Command alias

Alias	Definition	Description
lo	**dload**	Loads debugging information
lo -list_rc	**dload -list_reverse_connect**	Lists any reverse connection requests found
lo -reject_rc	**dload -reject_reverse_connect**	Reject either all or one reverse connection requests
dload -accept_rc	**dload -accept_reverse_connect**	Accept either all or one reverse connection requests

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
```

Sets up two environment variables **$DISPLAY** and **$STARTER** for the program `myprog` and loads `myprog`'s debugging information.

RELATED TOPICS

Loading Executables in the *TotalView User Guide*

dattach **Command**

drun **Command**

dmstat

Displays memory use information

Format

dmstat

Arguments

This command has no arguments

Description

The **dmstat** command displays information on your program's memory use, returning information in three parts:

- **Memory usage summary**: 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**: The amount of memory that each process is currently using.
- **Image information**: 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 describes the displayed columns:

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 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 the routines in its backtrace. If this is a multithreaded process, TotalView shows only 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 contains only 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. Note this four-process program:

```
  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
```

RELATED TOPICS

For information on memory debugging, see *Memory Debugging* in the *TotalView User Guide*.

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 defines 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 185.

Command alias

Alias	Definition	Description
n	**dnext**	Runs the *thread of interest* (TOI) 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 *TOI*, 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 *TOI* one statement and runs all threads in the process that are at the same PC as the *TOI* 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.
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 *TOI* one statement. Other threads in the control group run freely.
nw	{dfocus W dnext}	Steps worker threads in the process. This steps the *TOI* 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 *TOI* 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.

RELATED TOPICS

Debugging Using Group Width in the *TotalView User Guide*

Debugging Using Process Width in the *TotalView User Guide*

Debugging Using Thread Width in the *TotalView User Guide*

dnexti **Command**

dstep **Command**

dfocus **Command**

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* (TOI) while allowing other threads in the process to run.

The optional ***num-steps*** argument defines 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 185.

Command alias

Alias	Definition	Description
ni	**dnexti**	Runs the *TOI* 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 *TOI*, 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 *TOI* one instruction, and runs all threads in the process that are at the same PC as the *TOI* 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.
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 *TOI* one instruction. Other threads in the control group run freely.
niw	**{dfocus W dnexti}**	Steps worker threads in the process. This steps the *TOI* 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 *TOI* 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.

RELATED TOPICS

Debugging Using Group Width in the *TotalView User Guide*

Debugging Using Process Width in the *TotalView User Guide*

Debugging Using Thread Width in the *TotalView User Guide*

dnexti **Command**

dstep **Command**

dfocus **Command**

domp

Displays OpenMP information using the OMPD API

Format

domp [-parallel_regions] [-task_regions] [-control_vars] [-ompd] [-threads {-regions | -functions | -stack}][-send_symbols]

Arguments

-parallel_regions

Displays information about the parallel regions in the focus, including how the regions are nested. This command uses the standard **ptset** notation to identify which threads are in which regions. TotalView attempts to find the source location of each region shown in the aggregated display.

Default focus: thread

-task_regions

Displays information about the task regions in the focus, including how the regions are nested. This command uses the standard **ptset** notation to identify which threads are in which regions. TotalView attempts to find the source location of each region shown in the aggregated display.

Default focus: thread

-control_vars

Displays the settings of the OpenMP display control variables.

Default focus: process

-ompd

Displays information about the OMPD plugin support library being used by TotalView for each process in the focus:

Table 1: Information displayed by domp -ompd

Entry	Description
API Version	The version of the OMPD API supported by the DLL, which is returned by `ompd_get_api_version()`. For OMPD v5.0, that value is 201811.
DLL Version	The string returned by `ompd_get_version_string()`
DLL Name	The location of the OMPD DLL loaded by TotalView to handle that process. A specific DLL is loaded once into TotalView and then used for as many processes that specify it via the **ompd_dll_locations** variable in the process.

Default focus: process

-threads

Displays thread-centric information about the threads in the focus, including:

Table 2: Properties displayed by domp -threads

The TotalView ID for this thread
The operating system id for this thread
The OpenMP thread number within its team, i.e., the value returned to the thread by `omp_get_thread_num()`
The OpenMP state for this thread
OpenMP flags: **i**: The task is an implicit task. No corresponding routine in the OpenMP runtime reports this information. **p**: The thread is in an active parallel region. This corresponds to the `omp_in_parallel()` predicate in the OpenMP runtime. **f** : The task is a final task. This corresponds to the `omp_in_final()` in the OpenMP runtime. When **true**, the flag is displayed by just its letter. If **false**, a hyphen ("**-**") displays. If the flag could not be fetched, a "**?**" is displayed. For example, a value of `ip-` identifies a thread as implicit, in an active parallel region, and reports that it's not the final task, while `-pf` identifies a thread as explicit, in a parallel region, and the final task.

The **domp -threads** command also has these sub-options:

-regions

Lists the thread's nest of parallel and task regions.

-functions

Reports the source location of the code corresponding to the regions. The name of the function, its line number, and the file are displayed.

-stack

Reports the stack addresses corresponding to where: 1) control exited the OpenMP runtime to execute the task user code; and 2) control reentered the runtime from the user task code. The exit address is 0 for the root region, and for a leaf, the reentry address is 0.

Default focus: thread

-send_symbols

Sends the OMPD symbols to the TotalView tracers. These symbols are referenced by the OMPD library and their addresses are resolved by the tracers.

Classic UI only: For the Classic UI and the CLI only, explicitly sending the OMPD symbols to the tracers for each address space (process or CUDA context) by invoking **domp -send_symbols** is required. For the new UI, the TotalView client broadcasts the symbols required by the OMPD library to the tracers, allowing symbol lookup without requiring the access or storage of full symbol information in the servers.
Default focus: process

Description

The **domp** command displays OpenMP information in an OpenMP program. The information displayed here is used to populate the OpenMP view in the new UI.

Command alias

Alias	Definition	Description
-pr	**{domp -parallel_regions}**	Displays detail on the parallel regions in focus
-tr	**{domp -task_regions}**	Displays detail on the task regions in focus
-cv	**{domp -control_vars}**	Displays the OpenMP display control variables settings
-ss	**{domp -send_symbols}**	Sends symbols to the TotalView tracers

Examples

```
d1.<> domp -threads
```

Displays detail about the thread in focus. The "**i**" flag indicates that this thread is performing an implicit task, while "**p**" reports that this thread is in an active parallel region, while "**-**" means it is not the final task. The state "**work_parallel**" is an OMPD runtime flag indicating that the thread is executing code within the scope of a parallel region.

Output:

```
1 (216921): 1 OpenMP thread
tv_id          os_thread_id   thread_num          state     flags
=================================================================
   1         0x7f9679894740            0   work_parallel      ip-
```

```
d1.<> dfocus p1 domp -tr
```

First, changes the focus to the entire process, then calls `domp -tr` to display the nest of task regions for all the threads in process **p1**.

Here, the program has a single process with eight threads and an overall task region identified by the compiler-created outlined function `omp_outlined..13`, also identified by its line in the code (#91). This region in turn contains a single nested region `omp_outlined..6` which itself has two nested regions, one of which contains a task, `omp_task_entry`.

Output:

```
 +- / 1:8[p1.1-8]
    +- omp_outlined..13 [/tests/openmp#91] 1:8[p1.1-8]
       +- omp_outlined..6 [/tests/openmp#55] 1:8[p1.1-8]
          +- omp_outlined. [/tests/openmp#29] 1:2[p1.1, p1.5]
          +- omp_outlined..3 [/tests/openmp#38] 1:6[p1.2-4, p1.6-8]
             +- omp_task_entry. [/tests/openmp#42] 1:6[p1.2-4, p1.6-8]
```

```
d1.<> domp -threads -regions
```

Displays the nest of parallel and task regions for the thread in focus.

Output:

```
1 (216921): 1 OpenMP thread
tv_id          os_thread_id    thread_num          state    flags
===================================================================
    1       0x7f9679894740            0   work_parallel      ip-
    1                                                         ip-
    2                                                         i--
    3                                                         i--
```

To show all the threads for the process, enter: **dfocus p domp -threads -regions**.

RELATED TOPICS

Debugging OpenMP Applications in the *TotalView User Guide*

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. Without 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* (TOI) 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 *TOI*, 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 *TOI* out of the current function, and also runs all threads in the process that are at the same PC as the *TOI* 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 *TOI* out of the current function. Other threads in the control group run freely.
ouw	**{dfocus W dout}**	Runs worker threads in the process. This runs the *TOI* 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 *TOI* 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 185command information.

Examples

```
f t ou
```

Runs the current *TOI* 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.

RELATED TOPICS

Executing Out of a Function in the “Stepping through and Executing your Program” chapter of the *Classic TotalView User Guide*

dprint

Evaluates and displays information

Format

Prints the value of a variable or expression.

dprint [**-wait** | **-nowait**] [**-group_by**] [**-slice** "*slice_expr*"] [**-timeout** *seconds*] [**-stats** [**-data**]]
{ *variable* | *expression* }

Arguments

-wait

The default. TotalView waits until the expression is evaluated across the current focus, prints the values, and only then prompts for more interactive commands.

-nowait

Evaluates expressions (i.e., those that call functions in the target process) in the background. Use **TV::expr** to obtain the results, as they are not displayed.

-group_by

Aggregates data across a group rather than showing each individual process or thread's data value. The variable's value will be displayed with the **ptlist** (a compressed syntax for the process and thread list) alongside.

See Compressed List Syntax (ptlist)for a description of a **ptlist**.

-slice "*slice_expr*"

Defines an array slice—that is, a portion of the array—to print. If the programming language is C or C++, use a backslash (\) when you enter the array subscripts. For example, **"\[100:110\]"**.

-timeout *seconds*

Times out after the given number of seconds and returns an error if printing is not finished. If timeout is 0, **dprint** runs until it has completed, is interrupted, or encounters an error.

-stats

Displays statistical data about an array. When using this switch, the expression provided to **dprint** must resolve to an array. The **-slice** switch may be used with **-stats** to select a subset of values from the array to calculate statistics on.

-data

Returns the results of **dprint-stats** as data in the form of a Tcl nested associative array rather than as output to the console. See the description section for the structure of the array.

Note: This switch can be used *only* in conjunction with the **-stats** switch.

variable

A variable whose value is displayed. The variable can be local to the current stack frame or it can be global. If the displayed variable is an array, you can qualify the variable's name with a slice that displays 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).

For the option **-wait**, which is the default, if an event such as a **$stop**, SEGV, or 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
```

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

For 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**.

For the **-slice** option, TotalView uses the argument after **-slice** to select the values from the array to be printed, similar to the Slice field in the Data window. If the last argument does not result in an array value, the **-slice** switch is ignored.

For the **-timeout** option, a ***seconds*** argument of **0** turns off a previously set timeout value:

```
dprint -timeout 0
```

As the CLI displays data, it passes the data through a simple *more* processor that prompts you after it displays each screen of text. Press the Enter key to tell the CLI to continue displaying information. Entering **q** stops printing.

Since the **dprint** command can generate a considerable amount of output, you might want to use the **capture** command 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 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 following example illustrates the output from **dprint -stats** command:

```
d1.<> dprint -stats twod_array

Count: 2500
Zero Count: 1
Sum: 122500
Minimum: 0
Maximum: 98
Median: 49
Mean: 49
Standard Deviation: 20.4124145231932
First Quartile: 34
Third Quartile: 64
Lower Adjacent: 0
Upper Adjacent: 98

NaN Count: N/A
Infinity Count: N/A
Denormalized Count: N/A

Checksum: 41071
```

By adding the **-data** switch,

```
  d1.<> dprint -stats -data twod_array
```

the statistics are returned in a Tcl nested associative array, which has the following structure:

```
{
<dpid.dtid>
{
Count <value>
ZeroCount <value>
Sum <value>
Minimum <value>
Maximum <value>
Median <value>
Mean <value>
StandardDeviation <value>
FirstQuartile <value>
ThirdQuartile <value>
LowerAdjacent <value>
UpperAdjacent <value>
NaNCount <value>
InfinityCount <value>
DenormalizedCount <value>
Checksum <value>
}
<dpid.dtid>
{
...
}
}
```

The following example illustrates the output from **dprint -group_by** command, showing the value of a variable *random_int* across a group of 10 processes:

```
d1.<> dfocus g dprint -group_by random_int
Focus: 10:10[0-9.1]
0x00000000 (0) : 1:1[2.1]
0x00000001 (1) : 2:2[4.1, 6.1]
0x00000003 (3) : 2:2[0-1.1]
0x00000005 (5) : 2:2[5.1, 9.1]
0x00000006 (6) : 2:2[3.1, 8.1]
0x00000007 (7) : 1:1[7.1]
```

To access data for a single process/thread, use the following Tcl commands:

```
array set stats_data [dprint -stats -data <array-expression>]
array set stats $stats_data([lindex [array names stats_data] 0])
puts "Array Sum: $stats(Sum)"
```

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
```

NOTE: 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
```

RELATED TOPICS

Examining and Editing Data in the *TotalView User Guide*

Entering Expressions in the *TotalView User Guide*

TV::errorCodes**Command**

TV::expr**Command**

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 “Group, Process, and Thread Control” of the *Classic TotalView User 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 “Group, Process and Thread Control” in the *Classic TotalView User 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 163for 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.

RELATED TOPICS

Using Command Arguments in the *TotalView User Guide*

drun**Command**

dgo**Command**

captureCommand

drestart

Restarts a checkpoint (IBM RS6000 machines only)

Format

Restarts a checkpoint on IBM AIX

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

Arguments

-halt

TotalView stops checkpointed processes after it restarts them.

-g *gid*

Names the control group into which TotalView places all created processes.

-r *host*

Names the remote host upon which the restart occurs.

-no_same_hosts

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.

Description

The **drestart** command restores and restarts all of the checkpointed processes. The CLI attaches to the base process, and if there are parallel processes related to this base process, TotalView then attaches to them.

Restarting using LoadLeveler

If you checkpointed a LoadLeveler POE job, you cannot restart it with this command. 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. POE tells TotalView when it is time to attach to the parallel task so that it can complete the restart operation.

NOTE: When attaching to **POE**, parallel tasks will not have been created yet, so you should avoid trying to attach to them. Therefore, use the**-no_attach_parallel**option when using the **dattach** command to attach to POE.

Examples

```
drestart
```

Restarts the checkpointed processes. The CLI automatically attaches to parallel processes.

```
drestart -halt -no_same_hosts
```

Restarts the checkpointed processes using available hosts. Stops checkpointed processes after restoring them.

RELATED TOPICS

dcalltree **Command**

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(*dpid*)

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
```

Begins 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.

RELATED TOPICS

Using Command Arguments in the *TotalView User Guide*

Starting Processes and Threads in the “Manipulating Processes and Threads” chapter of the *Classic TotalView User Guide*

drerun **Command**

dgo **Command**

dsession

Loads a session

Format

Loads a session.

dsession [-load *session_name*]

Arguments

-load *session_name*

Loads the session with the given ***session_name***.

Description

Loads a previously created session. The session attributes are applied to the TotalView process object created for the executable named in the session. Returns the TotalView ID for the new object as a string value. A ***session_name*** that contains a space must be surrounded by quotes.

Sessions that attach to an existing process cannot be loaded this way; use the **dattach**command instead.

RELATED TOPICS

Loading a Session Using the Sessions Managerin the *TotalView User Guide*

Managing Sessions in the *TotalView User Guide*

dattach **Command**

dset

Changes or views CLI variables

Format

Sets a CLI variable

dset *debugger-var value*

Displays the current value of a CLI variable

dset *debugger-var*

Sets the default for a CLI variable

dset -set_as_default *debugger-var value*

Displays the current values of all the CLI variables in a debugger namespace. Using **dset** with no argument displays all the CLI variables in the global name space.

dset *debugger-namespace*

Arguments

debugger-var

Name of a CLI variable.

value

Value to be assigned to *debugger-var*.

debugger-namespace

Name of a CLI namespace. E.g., **TV::GUI::**. Note that you need to type the double colons at the end of the namespace name to indicate to Tcl that this is a namespace name rather than a variable name.

-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 a CLI variable to a string. With no new value, the current value is returned. If no variable is specified, all variables in the global namespace are displayed with their values.

For a list of ProductNameGeneric variables and their meanings, type **help variables** in the Command Line window. ProductNameGeneric variables are described in more depth in TotalView Variables on page 297.

The TotalView state variables are divided into several TCL namespaces. The most commonly used variables are in the global namespace. So to view the **LINES_PER_SCREEN** variable, for example, which is in the global namespace, you simply enter:

```
d1.<> dset LINES_PER_SCREEN
AUTO
```

For the other namespaces, the namespace for a variable must be included in the name. So to view the **platform** variable, which is in the **TV::** namespace, you would enter:

```
d1.<> dset TV::platform
linux-x86-64
```

To view all variables in a namespace, enter the namespace name including the trailing **::**. For example:

```
d1.<> dset TV::
TV::ask_on_dlopen true
TV::auto_array_cast_bounds {[10]}
TV::auto_array_cast_enabled false
...
```

Wild cards can be used to view a subset of the variables. The ***** wildcard matches any string, including "**:**", so to view all variables in all namespaces, you could enter:

```
d1.<> dset *
```

To view all variables with the string "font" in them you could enter:

```
d1.<> dset *font*
TV::GUI::display_font_dpi {}
TV::GUI::fixed_font fixed
TV::GUI::fixed_font_family {}
TV::GUI::fixed_font_size {}
TV::GUI::font 7x13bold
...
```

Using -set_as_default

When you press a default button in one of the tabs of the **File > Preferences** dialog box, TotalView reinitializes the settings to their default values. This happens even if you have set oneor more values in your **tvdrc** file. Settings in tvdrc do not change what TotalView thinks the default is, so it still changes the settings back to their defaults for the current session. However, the next time you invoke TotalView, TotalView will again use the values in your **tvdrc**.

You can tell TotalView that the values set in your **tvdrc** file are the defaults if you use the **-set_as_default** option. Now when you press a default button, TotalView will use the tvdrc value instead of the product-defined defaults.

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 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 its original 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.

RELATED TOPICS

dunset **Command**

dlappend **Command**

dskip

Creates and manages single-stepper skip rules

Format

Create a rule to skip over or through a function

dskip [**over** | **through**] [**function** | **-function** | **-fu**] *function-name*

Create a rule to skip over or through a file

dskip [**over** | **through**] [**file** | **-file** | **-fi**] *filename*

Create a rule to skip over or through functions that are also contained in specific source files

dskip [**over** | **through**] { { **-function** | **-fu** } *function-name* | { **-rfunction** | **-rfu** } *function-regexp* } { { **-file** | **-fi** } *filename* | { **-gfile** | **-gfi** } *file-glob* }

Enable or disable skipping of a list of IDs

dskip [**enable** | **disable**] [*id*]

Delete a list of skip IDs

dskip delete [*id*]

Print information about a list of skip IDs

dskip info [*id*]

Arguments

function-name

The name of the function for the skip rule. If ***function-name*** matches the base name or the fully qualified name of the subroutine symbol, then the rule matches the subroutine. However, if ***function-name*** does not match the base name, TotalView attempts to do a partial name match by demangling ***function-name*** using the demangler in the subroutine's containing file, or splitting an already demangled name into its components (base name, class name, overload string, etc.). If the base name and the specified name components in ***function-name*** match the subroutine's name components, then the rule matches the subroutine. Otherwise, the rule does not match the subroutine.

For example, all of the following ***function-name*** strings match the subroutine name `EnumS<Color>::EnumS(Color)` on Linux: `EnumS`, `EnumS(Color)`, `EnumS<Color>::EnumS`, and of course, `EnumS<Color>::EnumS(Color)`. Since the ***function-name*** is demangled (if necessary), `_ZN5EnumSI5ColorEC1ES0_` (the G++ mangled name of the subroutine) also matches.

function-regexp

The function name regular expression for the skip rule. If ***function-regexp*** matches the fully qualified name of the subroutine symbol using Tcl's “advanced” regular expression matching, then the rule matches the subroutine.

Otherwise, the rule does not match the subroutine.

Using a ***function-regexp*** can be useful when matching complex C++ function names, matching several functions with one rule, or to accommodate demangling differences across compilers or platforms. For example, the following ***function-regexp*** string matches all instances of `template<class T> EnumS<T>::EnumS(T)`:

```
^EnumS<(.*)>::EnumS\(\1\)
```

Note that it is best to wrap all Tcl regexp arguments in curly braces to prevent any expansion within the regexp string. For example, create the above rule as follows:

```
dskip -rfu {^EnumS<(.*)>::EnumS\(\1\)}
```

filename

The name of the file for the skip rule. If ***filename*** matches the base name or the compilation directory joined with the base name of the file symbol, then the rule matches the file.

Otherwise, the rule does not match the file.

For example, the following two ***filename*** strings match the file name `/user/smith/build/linux/test.cxx`:

```
test.cxx
/user/smith/build/linux/test.cxx
```

Note the match is performed purely on the contents of the file name strings. No attempt is made to resolve the ***filename*** string or the file symbol's name in the file system, for example, via realpath.

file-glob

The file name glob pattern for the skip rule. If ***file-glob*** contains a directory path delimiter ('/') and it matches the compilation directory joined with the base name of the file symbol using Tcl's string match, then the rule matches the file. However, if ***file-glob*** does not contain a directory path delimiter ('/') and it matches the base name of the file symbol using Tcl's string match, then the rule matches the file. Otherwise, the rule does not match the file.

For example, the following ***file-glob*** strings match the file name `/user/smith/build/linux/test.cxx`:

```
*.cxx
/user/*
```

Note the match is performed purely on the file name strings. No attempt is made to resolve the ***filename*** string or the file symbol's name in the file system. Also, unlike the shell's glob matching (which does access the file system), the directory path delimiter ('/') matches the glob special characters "*" and "?".

id

The skip ID or list of skip IDs on which the command operates. If one or more skip IDs are given, at least one skip ID from the list must match an existing rule. However, additional skip IDs that do match an existing rule are ignored. For example, if exactly one skip rule with ID "1" exists, then **dskip info 1 2** is not an error, but **dskip info 2** is an error.

Description

The **dskip** command allows you to create and manage single-stepper "skip" rules that modify the way source-level single stepping works. You can add rules that match a function, all functions in a source file, or a specific function in a specific source file. Functions can be matched by the function name or a regular expression (Tcl "regexp"). Files can be matched by the file name or a glob pattern (Tcl "string match").

When a rule is created its skip ID is returned, which starts at 1, is incremented by 1, and is never reused. The **dskip** command creates skip over rules by default, unless the **through** qualifier is specified.

The skip rules allow you to identify functions that you are not interested in debugging. TotalView implements two skip rule variants, **over** and **through**, as follows:

1. A matching and enabled skip **over** rule changes the behavior of all source-level step-into operations, such as the **dstep** command or the **Step** button or menu items in the graphical user interface.

   A skip over rule tells TotalView to not step into the function, but instead step over the function. Skip over is most useful to avoid stepping into library functions, such as C++ STL code.

   For example, consider the code fragment:

```
10 void MyFunc()
11 {
12 EnumS<Color> v ( Red );
13 if (v == Blue || OtherFunc())
14 std::cout << "Hello, world" << std::endl;
15 }
```

   If a skip over rule existed for `EnumS<Color>::EnumS(Color)`, then a **dstep** at line 12 would step over the call to the constructor (not into it) and the program would stop at line 13.

2. A matching and enabled skip **through** rule changes the behavior of all source-level single-stepping operations, such as the **dstep** and **dnext** commands or the **Step** and **Next** buttons or menu items in the graphical user interface.

   A skip through rule tells TotalView to ignore any source-line information for the function, so that single stepping does not stop at source lines within the function. However, if the function being skipped through calls another function, that call is handled according to original single-stepping operation. Skip through is most useful for callback or thunk functions.

   For example, consider the code fragment:

```
20 template<class Func>
21 inline void Callback(Func func)
22 {
23 func();
24 }

30 void MainFunc()
31 {
32 Callback(MyFunc);
33 }
```

If a skip through rule existed for `Callback`, then a **dstep** at line 32 would step through the code in `Call-back` and the program would stop inside `MyFunc`.

Enabling, Disabling, Deleting, and Printing Skip Rules

The **dskip** command **enable**, **disable**, **delete** and **info** subcommands take an optional list of skip IDs.

When a skip rule is disabled, it is completely ignored during single-stepping operations.

The **dskip info** command is the only command that generates output. It prints details about the specified rules in tabular format with the following column headings:

Column	Description
Id	The skip rule ID number.
Enb	Enabled or disabled.
How	Skip over or skip through.
Glob	File is a ***file-glob*** or ***filename***.
File	The ***filename***, ***file-glob***, or none.
RE	Function is a ***function-regexp*** or ***function-name***.
Function	The ***function-name***, ***function-regexp***, or none.

Name Matching

Each skip rule contains a function name or function regular expression, and/or a file name or file glob pattern. During certain single-stepping operations, TotalView attempts to map program locations to symbol objects in the symbol table for the program, specifically subroutine and file symbols. If the symbols are found, it then attempts to match the names of the symbols against the names, regular expressions, and glob patterns contained within the skip rules.

For example, when executing a source-level step-into operation, if a **call** instruction is encountered, TotalView maps the address of the function that is being called to a subroutine and file symbol, and attempts to match the symbol names' against the skip rules. If an enabled skip over rule is matched, the call instruction is stepped over instead of stepped into.

The way in which TotalView matches the symbol names against the rules depends on a how the rule was created. Further, the rule might be matched against symbol name variations.

- When matching subroutine symbols against function skip rules, TotalView uses either the subroutine symbol's base name or fully qualified name.

  A fully qualified subroutine name contains any combination namespace, class name, template types, base name, and function signature.

- When matching file symbols against file skip rules, TotalView uses either the file symbol's base name or compilation directory joined with the base name.

  The compilation directory is a canonicalized version of the directory hint and directory path information in the symbol table. For example, if the compiler path was `../../src` and the directory hint (the compiler's "pwd") was `/user/smith/build/linux`, the compilation directory would be `/user/smith/src`.

Examples

```
dskip through function Callback
```

Skip through functions named "Callback".

```
dskip -rfunction {^EnumS<(.*)>::EnumS\(\1\)}
```

Skip over functions with names that match the regular expression.

```
dskip file /user/smith/build/linux/test.cxx
```

Skip over all functions contained in the source file named `/user/smith/build/linux/test.cxx`.

```
dskip over -gfile *.cxx
```

Skip over all functions contained in source files with names that match the file glob pattern, in this case all files with a `.cxx`extension.

```
dskip through -rfunction {^EnumS<(.*)>::EnumS\(\1\)} -gfile *.cxx
```

Skip through functions that match the regular expression that are also contained in source files with a `.cxx` extension.

dstacktransform

Maintains rules that change the displayed stack frames

Format

Enables or disables the stack transform facility.

dstacktransform [enable | **disable***id* | *transform_name* **]**

Prints the current state of rules and transforms.

dstacktransform **[list**]

Prints the enabled/disabled state of the stack transform facility.

dstacktransform **[status**]

Removes the rule with the given *id* from the stack transform facility.

dstacktransform **[remove** *id*]

Adds a new transform.

dstacktransform add [**-name** | **-n** *string*] [**-implementation** | **-i***path*]

Adds a new transform rule.

dstacktransform add [-filter *test_function_list*] [**-transform** | **-t** *name*] [**-operation** | **-o** *operation_name* **[-position** | **-p** *integer*] [**-before** | **-b** *integer*]

Arguments

id

A rule's ID number.

transform_name

A transform's name.

-filter | **-f** *test_function_list*

Required argument for all rules. Its value *test_function_list* is a comma-separated list of filter functions.

-implementation | **-i** *path*

The path of a script or shared library that supports the `modify` operation.

NOTE: This is not implemented in the current release. The only implemented transforms are provided by the debugger and are listed as *built-in* using **dstacktransform list.**

-name | **-n** *string*

A transform's name.

-transform | -t *name*

Indicates the transform name that provides the implementation of the `modify` operation. This name can also be used to enable and disable groups of rules.

-operation | -o *operation_name*

The action to be invoked if the filter matches the current frame. The allowed operations are `remove`, `modify`, and `next`:

`remove`: Hides the current stack frame.

`modify`: Invokes a transform function to change the name and behavior of the stack frame. Currently, user transforms cannot use the modify operation.

`next`: Ends processing of the current stack frame without changes. Rules lower in the transform list are not evaluated, and processing the stack continues with the next stack frame.

The default is `remove`, which is used if no **-operation** option is provided.

-position | -p *integer*

Puts the new rule at the given ordinal position in the transform's list of rules.

-before |-b *integer*

Puts the new rule before the rule with the given *id*. If no **-position** or **-before**option is given, the new rule is placed at the end of the transform's rule list.

Description

The stack transform facility (STF) maintains a list of rules that modify the list of stack frames shown in the stack trace view and the **dwhere** command. These rules are tested, starting at the top of the list, until a rule passes the tests and is applied. The various subcommands of **dstacktransform** maintain the list of rules.

Each rule has a filter that runs one or more tests on elements of each stack frame. Some tests compare the function name or image path with a regular expression. Other tests look for a loader symbol in the image associated with the frame. If the tests pass, an operation is run that controls the application of rules or modifies the stack by adding, removing, or re-writing one or more frames on the stack. If the tests pass and an operation is run, no further action is taken on that frame and processing continues to the next stack frame.

Rules can refer to *transforms*. Transforms contain code that is run when *modify* operations are invoked by rules.

NOTE: Currently, you cannot load your own transforms, and the debugger supplies a default transform for Python frames. Although you can define new transforms without implementing code, these are useful only to group a number of rules under a common name for enabling and disabling.

About a Rule’s Filter

The **-filter** or **-f**argument is required for all rules. The value of the filter is a comma-separated list of filter functions. Each function can be one of:

`function(<regular_expression>)`: This matches the function name in the stack frame against the given regular expression.

`image(<regular_expression>)`: This matches the image path in the stack frame against the given regular expression.

`symbol(<string>)`: Looks for the string in the list of loader symbols defined in the image associated with the stack frame.

All the filter functions must match for a rule's operation to be invoked.

NOTE: The TCL interpreter that reads these commands will try to interpret common regular expression syntax. For example, TCL considers text inside square brackets as a function to be evaluated. To prevent TCL from trying to evaluate regular expression character sets as functions, surround either the regular expression or the entire filter function in double-quotes " or curly braces {}.

About a Rule’s ID

When a rule is created, it is assigned a unique *id* by the debugger. This id is returned by the **dstacktransform add**command. Use this *id* to manage the rule, for instance, to disable it:

```
> set id [dstacktransform add -filter "function('_start')"]
> dstacktransform disable $id
```

The List Subcommand

The **list** subcommand prints the current state of rules and transforms. The rules are applied to each stack frame in the order shown in the list.

This example adds a rule and then prints the list of rules followed by the list of transforms:

```
> sta -f function('_init') -p 1
4

> dstacktransform list
Transformation Status: Enabled

Rules
ID Transform Operation Filter
4 remove function('_init')
1 RW_Python modify function('PyEval_EvalFrameEx')
2 RW_Python remove symbol('Py_DebugFlag')
```

```
Transforms
Name Implementation
RW_Python <built-in>
```

Command alias

Alias	Definition	Description
ste	**dstacktransform enable**	Enables the stack transform facility.
std	**dstacktransform disable**	Disables the stack transform facility.
sta	**dstacktransform add**	Adds a new transform or rule.
str	**dstacktransform remove**	Removes a transform.
stl	**dstacktransform list**	Prints the current rules and transforms and their states.
sts	**dstacktransform status**	Prints the enabled/disabled state of the stack transform facility.

Examples

```
dstacktransform enable 10
ste 10
```

Enables the rule identified by the *id* `10`.

```
dstacktransform disable 10
std 10
```

Disables the rule identified by the *id* `10`.

```
dstacktransform add -t MY_PYTHON -f "function('_start')" -o remove -p 0
```

Adds a new rule associated with the `MY_PYTHON` transform. The rule has a filter named `function('_start')`, which tries to match the pattern `_start` with the function name in the current stack frame. If they match, the frame is removed. This new rule is placed at the top of the list of rules.

After adding the rule above, enter **dstacktransform list** to view it and other rules:

```
> dstacktransform list
Transformation Status: Enabled

Rules
ID Transform Operation Filter
5 MY_PYTHON remove function('_start')
4 remove function('_init')
1 RW_Python modify function('PyEval_EvalFrameEx'),symbol('Py_DebugFlag')
2 RW_Python remove symbol('Py_DebugFlag')
3 RW_Python remove function('wrap'),symbol('SWIG_globals')

Transforms
Name Implementation
RW_Python <built-in>
```

RELATED TOPICS

dwhere**Command**

Transforming the Stack in the *TotalView User Guide*

dstatus

Shows current status of processes and threads

Format

dstatus

dstatus [**-g**]

dstatus [**-group_by** { *control_group* | *share_group* | *process_state* | *hostname* | *replay* | *pheld* | *thread_state* | *pc* | *function* | *line* | *dpid* | *dtid* | *apid* | *theld* | *stop_reason* | *sysid* | *utid_ktid* | *tname* }

Arguments

-group_by | **-g**

Displays an aggregated view of the processes and threads in the current focus. The processes and threads are aggregated based on the order of the properties chosen in the comma-separated list of properties in the property list.

The aggregation is shown using either a compressed process list for process-level properties (`plist`) or a compressed thread list for thread-level properties (**ptlist**). See Compressed List Syntax (ptlist) for a description of a **ptlist**.

Process level properties:

control_group (abbreviated as **cgroup**)

The control group of the process

share_group (abbreviated as **sgroup**)

The share group of the process

process_state (abbreviated as **pstate**)

The state of the process

replay

The replay mode of the process. A process can be in three replay states: `Replay Unavailable`, `Replay`, or -`Record`.

pheld

Process hold state: `Held` or `UnHeld`.

dpid

The debugger-assigned process ID

hostname

The hostname of the process

Thread level properties:

thread_state (abbreviated as **tstate**)

The state of the thread

pc

The Program Counter of the thread

function

The function where the thread's pc is currently

line

The line number for the current thread's pc

dtid

The debugger-assigned thread ID

apid

The action point identifier that the thread's pc is on. If the thread is not at an action point, it is grouped as `No Action`

theld

The thread's held state, either `Held` or `Not Held`

stop_reason

The stop code and stop message for a stopped thread

systid

Either the user thread ID (**utid**) or the kernel thread ID (**ktid**) if no **utid** exists

utid_ktid

"**utid / ktid**" or just **ktid** if no **utid** exists. User level thread IDs are assigned by a runtime library such as a **pthreads** implementation. Kernel thread IDs are assigned by the operating system.

tname

Thread name or "<unnamed>" if no thread name has been defined. Some runtime implementations, such as **pthreads**, allow the user to programmatically assign a name to a thread.

-pcount

Alias for the -**ptlist_element_count** argument

-ptlist_element_count *number*

Displays, at maximum, *number* elements (comma separated **plist**s or **ptlist**s) in the process/thread compressed list that is shown in a reduced **dstatus** display. If a reduction results in exceeding the **ptlist_element_count**, an ellipsis is appended. For instance, if **ptlist_element_count** is set to `5`:

```
[p1-4.1, p2.2, p3-4.3, p5.4, p6.1-2, ...]
```

To change the default value, use the TotalView State variable **ptlist_element_threshold**. For example:

```
dset TV::ptlist_element_threshold 10
```

-levels

The number of levels to show for a set of properties. If no levels are specified, then each property is reduced on a new line with indentation. If the number of levels is less than the number of specified properties, then the remaining properties are shown in a single reduction on one line.

-v

Show verbose output in the reduced display. Without **-v**, full paths of filenames and line numbers are not displayed.

-detail

Force full detailed information for the current state of each process and thread in the current focus. This option affects the amount of information displayed from grouping by function.

-thread_name

Show a thread's name, if one exists.

Description

With the **-group_by** option, the **dstatus** command displays an aggregated view of the process and thread state in the current focus. To make the display more useful, you can reduce it based on specific properties, provided as arguments as described above. The full detail shows the current state of each process and thread in the current focus. **ST** is aliased to `dfocus g dstatus` and acts as a group-status command. Type `help ptset` for more information.

If you have not changed the focus, the default 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. When you limit thread state display by certain properties, the output is displayed as a compressed thread list, or **ptlist**.

Compressed List Syntax (ptlist)

A compressed **ptlist** consists of a process and thread count, followed by square-bracket-enclosed list of process and thread ranges separated by dot (`.`). If the thread range is missing, it's merely a compressed list of processes and it is referred to as a **plist**.

If the process range starts with the letter `p`, the process IDs are TotalView DPIDs (debugger unique process identifiers); otherwise, they are the MPI rank for the process, `MPI_COMM_WORLD`.

The thread IDs are always TotalView DTIDs (debugger unique thread identifiers). For example, the compressed **ptlist** `5:13[0-3.1-3, p1.1]` indicates that there are five processes and 13 threads in the list. The process and thread range `0-3.1-3` indicates MPI rank processes `0` through `3`, each with DTIDs `1` through `3`. The process range `p1.1` indicates process DPID `1` and thread DTID `1`, normally the MPI starter process named `mpirun`.

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=0x0d9cae6
  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*(TOI).

```
d1.<> dfocus g dstatus -group_by thread_state, function
```

First reduces the focus by *thread_state*, then further breaks down and reduces the results according to the function the threads are in within each thread state. This call might output this reduced display:

```
  Focus: 4:20[p1-4.1-5]
  Breakpoint : 4:4[p1.2, p3-4.2, p2.3]
  snore : 4:4[p1.2, p3-4.2, p2.3]
  Stopped : 4:16[p1-4.1, p2.2, p1.3, p3-4.3, p1-4.4-5]
  .___newselect_nocancel : 4:13[p1-4.1, p2.2, p3-4.3, p1.4]
  snore : 2:3[p1.3, p2.4-5]
```

The above output displays the reduction produced by the **group_by** command as a series of **ptlist**s. (See above, Compressed List Syntax (ptlist)).

```
dfocus group dwhere -group_by function
```

This **dwhere** call output shows that all the processes have the first three frames in their backtrace but then they diverge and one process is in function `rank0` while the other three processes are in `rankn`.

```
  +/ : 10:10[0-9.1]
  +_start
  +__libc_start_main
  +main
  +rank0 : 1:1[0.1]
  +rankn : 3:3[1.1, 5.1, 8.1]
```

RELATED TOPICS

Viewing Processes and Threads in the *TotalView User Guide*

dwhat **Command**

dwhere **Command**

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 defines the number of **dstep** operations to perform. If you do not specify ***num-steps***, the default is 1.

NOTE: You can use the dskip command to create and manage single-stepper “skip” rules that modify the way source-level single stepping works. You can add rules that match a function, all functions in a source file, or a specific function in a specific source file.

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 steps only the thread of interest (TOI).

NOTE: On systems having identifiable manager threads, the **dfocus t dstep** command allows the manager threads as well as the TOI 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 *TOI*. TotalView selects one matching thread from each matching process. TotalView then runs all processes in the group and waits until the *TOI* 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 (POI). Regardless of which threads are in the group of interest, TotalView only waits for threads that are in the same share group as the *TOI*. 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 *TOI* 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 *TOI* reaches the goal.

Thread group

TotalView runs all threads in the process that are in that group to the same goal as the *TOI*. 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 *TOI* is allowed to run. (This is not supported on all systems.)

Command alias

Alias	Definition	Description
s	**dstep**	Runs the *TOI* 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 *TOI*, 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 *TOI* one statement, and runs all threads in the process that are at the same PC as the *TOI* 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 *TOI* one statement. Other threads in the control group run freely.
sw	**{dfocus W dstep}**	Steps worker threads in the process. This steps the *TOI* 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 *TOI* 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.

RELATED TOPICS

Setting Breakpoints in the *TotalView User Guide*

Creating a Process by Single Stepping in the “Manipulating Processes and Threads” chapter of the *Classic TotalView User Guide*

dstepi **Command**

dnext **Command**

dfocus **Command**

dstepi

Steps machine instructions, stepping into subfunctions

Format

dstepi [-back][*num-steps* **]**

Arguments

-back

(ReplayEngine only). Steps backward to previously executed instructions, possibly moving into subroutines that were called before 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 defines the number of **dstepi** operations to perform. If you do not specify *num-steps*, the default is 1.

For more information, see **dstep** on page 185.

Command alias

Alias	Definition	Description
si	**dstepi**	Runs the *thread of interest*(TOI) 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 *TOI*, 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 *TOI* one instruction, and runs all threads in the process that are at the same PC as the *TOI* 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 *TOI* one instruction. Other threads in the control group run freely.
siw	**{dfocus W dstepi}**	Steps worker threads in the process. This steps the *TOI* 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 *TOI* 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.

RELATED TOPICS

Setting Breakpoints in the *TotalView User Guide*

Creating a Process by Single Stepping in the “Manipulating Processes and Threads” chapter of the *Classic TotalView User Guide*

Stepping and Setting Breakpoints in the “Manipulating Processes and Threads” chapter of the *Classic TotalView User Guide*

Using Stepping Commands in the *Classic TotalView User Guide*

dstep **Command**

dnext **Command**

dfocus **Command**

dunhold

Releases a held process or thread

Format

Releases a process

dunhold -process

Releases a thread

dunhold -thread

Arguments

-process

Releases processes in the current focus. You can abbreviate the **-process** option argument to **-p**.

-thread

Releases 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 focus process
UHP	**{dfocus g dunhold -process}**	Releases the processes in the focus group
uht	**{dfocus t dunhold -thread}**	Releases the focus thread
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 *TOI*.

RELATED TOPICS

Starting Processes and Threads in the “Manipulating Processes and Threads” chapter of the *Classic TotalView User Guide*

Holding and Releasing Processes and Threads in the “Manipulating Processes and Threads” chapter of the *Classic TotalView User Guide*

dhold **Command**

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 TotalView Variables on page 297for 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*** restores 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.

RELATED TOPICS

dset **Command**

duntil

Runs the process until a target place is reached

Format

Runs to a line

duntil [-back] *line-number*

Runs to an absolute address

duntil [-back] -address *addr*

Runs into a function

duntil [-back]*proc-name*

Arguments

-b | -back

(ReplayEngine only). Moves back in execution time to the most recent point at which the other argument (***line-number***, ***address***, or ***proc-name***) was executed.

line-number

A line number in your program.

-address*addr*

An absolute address in your program.

proc-name

The name of a procedure, function, or subroutine in your program.

Description

The **duntil** command runs the threads that are members of the focus group until execution reaches the goal specified by the line number, absolute address, or function arguments. Threads already stopped at the goal are not run. Execution ends when a thread reaches the goal or stops for any other reason.

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

Process group

Runs the entire group, and the CLI waits until all processes in the group contain at least one thread that has arrived at the goal. This lets you *sync* all the processes in a group in preparation for group-stepping them.

Thread group

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 in the way processes and threads run using the **duntil** command and other stepping commands:

- **Process Group Operation**: TotalView examines the *TOI* to see if it is already at the goal. If it is, TotalView does not run the POI. Similarly, TotalView examines all other processes in the share group, and runs only processes without a thread at the goal. It also runs members of the control group 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 outside the share group.
- **Process-Width Thread Group Operation**: TotalView identifies all threads in the entire focus process not already at the goal. Only those threads run.

Command alias

Alias	Definition	Description
un	**duntil**	Runs the *TOI* 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 *TOI* until it reaches the target, and runs all threads in the process that are at the same PC as the *TOI* 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.

RELATED TOPICS

Executing to a Selected Line in the “Stepping through and Executing your Program” chapter of the *Classic TotalView User Guide*

Groups in TotalView in the *TotalView User Guide*

Using Run To and duntil Commands in the “Stepping (Part I)” section of the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

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 program enters a procedure, and decreases by one when the 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 allows access to 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 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.

RELATED TOPICS

ddown **Command**

dwait

Blocks command input until the target processes stop

Format

dwait

Arguments

This command has no arguments.

Description

The **dwait** command waits 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**] [[**-l***lang*] **-e** *expr*] [**-type***type*]

Defines a watchpoint for an absolute address

dwatch -address *addr***-length** *byte-count* [**-g**| **-p**] [[**-l***lang*] **-e***expr*] [**-type** *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 need to specify only the amount of storage to watch if you want to override the default value.

-g

Stops all processes in the process's control group when the watchpoint triggers.

-p

Stops the process that hit this watchpoint.

-t

Stops 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 specify only 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 (**{ }**).

-type*type*

The data type of **$oldval**/**$newval**in the expression. If you do not use this option, TotalView uses the variable's datatype. If you specify 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.

NOTE: Watchpoints are not available on Macintosh computers running macOS.

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.

To obtain a variable's address 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.

See Watchpoints in the *TotalView User Guide* for 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 watch a certain number of bytes, starting at the named location.

NOTE: 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.

RELATED TOPICS

Watchpoints in the **TotalView User Guide**.

Writing Code Fragments in the “Evaluating Expressions” chapter of the *Classic TotalView User Guide*

dactions **Command**

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 displays information about a symbol. For a variable name, **dwhat** displays the type, location, storage class, and other relevant information for each variable of that name in the scope of the current focus. For a type name, **dwhat** displays general information about the data type.

NOTE: To view information on CLI variables or aliases, 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 the CLI display for various 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_sectime_t(32 bits)
        tv_usecint(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_intint(32 bits)
    field2_charchar(8 bits)
    field2_chars$string[2](16 bits)
    <padding>$char(8 bits)
    field3_intint(32 bits)
    field2_uchar unsigned char(8 bits)
    <padding>$char[3](24 bits)
    field_substruct sub_st(320 bits){
       field1_intint(32 bits)
       <padding>$char[4](32 bits)
       field2_longlong(64 bits)
       field2_ulongunsigned long(64 bits)
       field3_uintunsigned int(32 bits)
       en1enum en_check (32 bits)
       field3_doubledouble(64 bits)
    }
  ...
  }
```

RELATED TOPICS

dstatus **Command**

dwhere **Command**

dwhere

Displays the current execution location and call stack

Format

Displays locations in the call stack

dwhere [**-level** *start-level*] [*num-levels* | *-all*] [**-a** | **-args**] [**-no_args** [**-locals**] [**-no_locals**] [**-registers**] [**-no_registers**] [**-fp_registers**] [**-no_fp_registers**] [**-show_pc**] [**-noshow_pc**] [**-show_fp**] [**-noshow_fp**][**-show_image**] [**-noshow_image**] [**-group_by** *property*]

Displays all locations in the call stack

dwhere -all[**-args**] [**-locals**] [**-registers**] [**-noshow_pc**][**-noshow_fp**][**-show_image**]

Arguments

-all

Shows all levels of the call stack. This is the default.

-level | **-l** *start-level*

Sets the level at which **dwhere** starts displaying information. Frame levels start from **0**, and this is the default.

num-levels

Restricts output to this number of levels of the call stack. Defaults to the value of debugger state variable **MAX-_LEVELS**.

-args | **-a**

Displays argument names and values in addition to program location information. By default, the arguments are not shown.

-no_args

Does not display argument names and values with program location information. This is the default.

-locals

Displays each frame's local variables as well as program location information. This option also displays the arguments to the function as well unless the **-no_args** option is specified. By default, the local variable information is not shown.

-no_locals

Does not display each frame's local variables with program location information. This is the default.

-show_pc

Displays the program counter (PC) value in the program location information. This is the default.

-no_show_pc

Does not show the PC. By default, the PC value is shown.

-show_fp

Displays the frame pointer (FP) value in the program location information. This is the default.

-no_show_fp

Does not show the FP. This may be useful when comparing **dwhere** output. By default, the FP value is shown.

-registers

Displays each frame's registers with program location information. By default, the register information is not shown.

-no_registers

Does not display each frame's registers. This is the default.

-fp_registers

Displays the floating point registers and their values as well as program location information. By default, the floating point register information is not shown.

-no_fp_registers

Does not display the floating point registers. This is the default.

-show_image

Displays the associated image at the location, if the source line cannot be found. This is the default.

-no_show_image

Does not display the associated image at the location when the source line cannot be found. This may be useful when comparing **dwhere** output. By default, **dwhere** displays the associated image information if the source line cannot be found.

-group_by| -g ***property***

Aggregates stack backtraces of the focus threads, outputting a compressed **ptlist** that identifies the processes and threads containing equivalent stack frames in the backtrace. For information on the **ptlist** syntax, see Compressed List Syntax (ptlist)or type "**help ptlist**" in the CLI.

This option requires a ***property*** argument to control the "equivalence" relationship of stack frames across the threads. See "The -group_by Option" below for more information.

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; the default shows information at the thread level.

Arguments control the amount of command output in two ways:

- The ***num-levels*** argument determines how many levels of the call stacks are displayed, counting from the uppermost (most recent) level. Without this argument, the CLI shows all levels in the call stack, which is the default.
- The **-a** option displays 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_LEVELSvariable contains the default maximum number of levels displayed when you do not use the ***num-levels*** argument.

The **dwhere** command displays the current execution location(s) and the backtrace(s) for the threads in the current focus, defaulting to thread level. If backtraces for multiple threads are requested, the stack displays are aggregated.

Lines denoting evaluation frames for compiled expressions or interpreted function calls are labeled with a suspended evaluation id. This id can be used to manipulate suspended evaluations with **dflush** and **TV::expr**.

Output is generated for each thread in the target focus. The output is printed directly to the console.

The -group_by Option

The **-group_by** option requires a ***property*** argument, which controls the "equivalence" relationship of stack frames across the threads. When you use the --**group_by** option, **dwhere** aggregates the stack frames of each of the focus threads, forming a tree of equivalent stack frames.

Starting at the base of the stack (closest to `main()` or the thread's start function), the **dwhere** command assigns each frame a distance from a synthetic root frame indicated by `/`. Two frames are equivalent only if all of the following apply:

- Their distance from the root is equal.
- They have the same parent frame.
- The selected property of frames is equivalent.

The following ***property*** values are supported, with their abbreviations in parentheses:

- *function (f)*: Equivalence based on the name of the function containing the PC for the frame.
- *function+line (f+l)*: Equivalence based on the name of the function and the file and line number containing the PC for the frame.
- *function+offset (f+o)*: Equivalence based on the name of the function containing the PC for the frame and offset from the beginning of the function to the PC for the frame.

Looking at backtraces purely by the ***function*** property is the most coarse grained grouping of threads. Choosing a more fine-grained grouping, such as a line number within the function, provides more detail about where in the code a given thread is executing, but it may also result in a much larger set of equivalent frames.

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.

RELATED TOPICS

dwhat **Command**

dstatus **Command**

dworker

Adds or removes a thread from a workers group

Format

dworker{ *number* | *boolean* }

Arguments

number

If positive, marks the thread of interest (TOI) as a worker thread by inserting it into the workers group.

boolean

If **true**, marks the *TOI* as a worker thread by inserting it into the workers group. If **false**, marks the thread as 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 *TOI* as a nonworker thread by removing it from the workers group. If *number* is **true** or is a positive value or *boolean* is **true**, this command marks the *TOI* 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

RELATED TOPICS

Organizing Chaos and **Creating Groups** in the “About Groups, Processes, and Threads” chapter of the *Classic TotalView User Guide*

Setting Group Focus in the “Group, Process, and Thread Control” chapter of the *Classic TotalView User Guide*

dgroups **Command**

exit

Terminates the debugging session

Format

exit [-force]

Arguments

-force

Exits 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 confirms that you wish to exit, then exits. If you entered the CLI from the TotalView GUI, this command also closes the GUI window.

NOTE: If you 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 and do the same thing.

Examples

```
exit
```

Exits TotalView, leaving any attached processes running.

RELATED TOPICS

quit **Command**

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. With no 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. Press Enter to see more data or **q** to return to the CLI prompt.

When you enter a topic name, the CLI attempts to complete an entry. You can also 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:
...
```

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

Closes all TotalView processes without asking permission.

Description

The **quit** command terminates the TotalView session.

After you enter this command, the CLI confirms that you wish to exit, then exits. If you entered the CLI from the TotalView GUI, this command also closes the GUI window.

NOTE: If you 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 and do the same thing.

Examples

```
quit
```

Exits TotalView, leaving any attached processes running.

RELATED TOPICS

exit **Command**

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, allowing you to set all your terminal's properties. However, this is most often used to set erase and kill characters.

If you start the CLI from a terminal 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 returns help information on your current settings.

The output from this command is returned as a string.

Examples

```
stty
```

Prints information about your terminal settings, equivalent to having entered **stty** while interacting with a shell.

```
stty -a
```

Prints information on all 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. For 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

Removes all aliases.

Description

The **unalias** command removes a previously defined alias. You can delete all aliases 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.

RELATED TOPICS

alias **Command**

CLI Namespace Commands

This chapter lists all CLI commands that are not in the top-level namespace.

Commands by Category

NOTE: This chapter describes some functionality that exists in the underlying debugging engine TotalView, but may not be supported in the TotalView user interface. To access these features, use the Command Line view. See on page 1for more details.

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.

All Commands

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 examines and sets the following action point properties and states:

address

The provisional and relocated block address pair of the action point. The command focus is used to relocate the provisional address. If the action point is planted in multiple locations (for instance, when it's in both host CPU code and GPU CUDA code), this is a list of pairs, where each pair is the provisional and relocated block address.

For example, this breakpoint is planted in three locations, two of which appear as "not mapped" because the command's focus is not on those threads:

```
address: {0xff00000090003998 0x00403998} {0xff000000911f6550 <NotMapped>}
{0xff000000911f7d50 <NotMapped>}
```

block_count

The number of address blocks associated with an action point.

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 action point can be set at many addresses even if you are placing it on a single line.

Internally, an address block represents one of these addresses.

block_enabled

Each block can be enabled or disabled separately. This property type returns a list within which 1 indicates if the block is enabled and 0 if 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 action point is conflicted, and 0 if it is not.

context

A string that represents the scope in which the action point was created.

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)

function

A list of soids (symbol object ID) indexed by block id, where the soid is for a subroutine or loader symbol.

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

A list of soids indexed by block id, where the soid is identifies a line number symbol where the action point is set. This property is not valid for watchpoints.

location

The string representing the breakpoint expression.

pending

A value (either 1 or 0) identifying whether the action point has at least one valid block (0) or no valid blocks (1).

pending_eval

A value (either 1 or 0) identifying whether the action point is a pending eval point (1) or is not a pending eval point. This property applies to eval points only.

pending_in_address_space

A value (either 1 or 0) identifying whether the action point has at least one relocatable block (0) or no relocatable blocks (1).

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)

shaded_by_better_match

A list of values (either 1 or 0) indexed by block id, indicating whether the block is not shaded (0) or shaded (1) by a better match. (A *shaded* block is one that has been marked as nullified by TotalView.).

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**.

value_type

A string that represents the type of the value being watched. Valid for watch points only.

Examples

```
TV::actionpoint set 5 share 1 enable 1
```

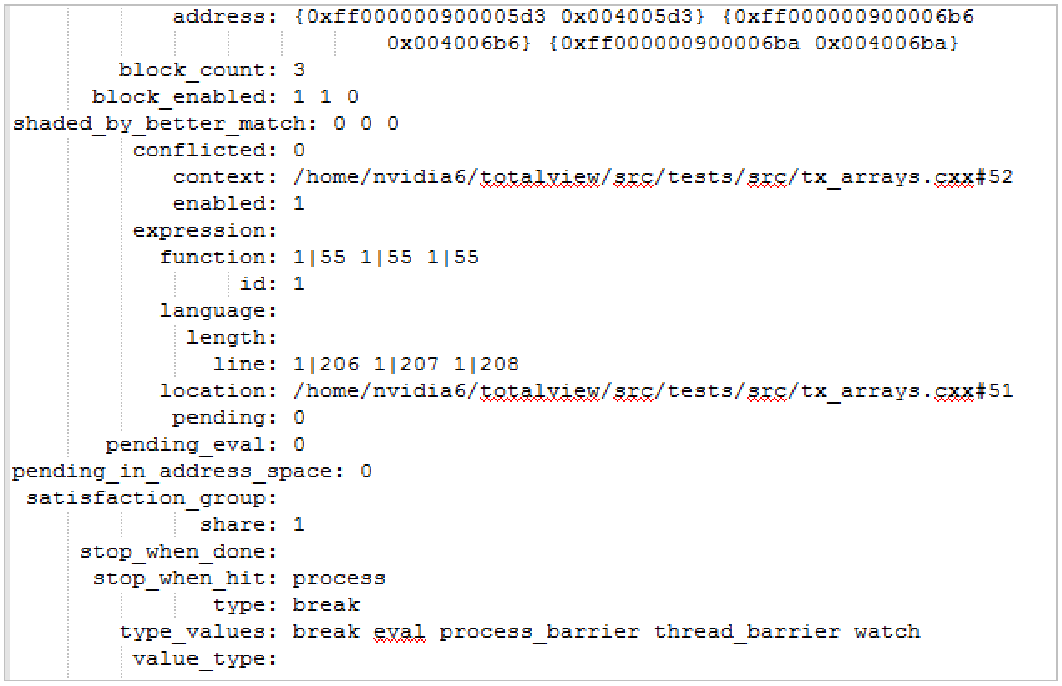
Shares and enables action point 5.

```
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: {0xff000000900005d3 0x004005d3} {0xff000000900006b6
                            0x004006b6} {0xff000000900006ba 0x004006ba}
        block_count: 3
      block_enabled: 1 1 0
shaded_by_better_match: 0 0 0
         conflicted: 0
            context: /home/nvidia6/totalview/src/tests/src/tx_arrays.cxx#52
            enabled: 1
         expression:
           function: 1|55 1|55 1|55
                 id: 1
           language:
             length:
               line: 1|206 1|207 1|208
           location: /home/nvidia6/totalview/src/tests/src/tx_arrays.cxx#51
            pending: 0
       pending_eval: 0
pending_in_address_space: 0
 satisfaction_group:
              share: 1
     stop_when_done:
      stop_when_hit: process
               type: break
        type_values: break eval process_barrier thread_barrier watch
         value_type:
```

RELATED TOPICS

dactionsCommand

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.

RELATED TOPICS

hex2dec**Command**

dll

Manages shared libraries

Format

TV::dll *action* [*dll-id-list* | *-all*] [*other-args*]

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 libraries that it opened.

commands

Displays available subcommands. 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. This subcommand takes no arguments.

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. These are the IDs returned by the **ddlopen** command.

-all

Performs the specified action for all libraries opened with 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.

```
d1.<> ddlopen /usr/lib64/libnuma.so
Process 1 has loaded the library /usr/lib64/libnuma.so
1
d1.<> ddlopen /usr/lib64/libz.so
Process 1 has loaded the library /usr/lib64/libz.so
2
d1.<> TV::dll get -all id
1 2
d1.<> TV::dll get 2 name
/usr/lib64/libz.so
```

First opens two shared libraries, then retrieves some properties: first, the **id** for both; then the **name** of the second library.

RELATED TOPICS

ddlopen**Command**

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 raises 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.

RELATED TOPICS

dprintCommand

TV::exprCommand

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.<>
```

RELATED TOPICS

dprintCommand

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.

RELATED TOPICS

- focus_processes**Command**
- focus_threads**Command**
- dfocus**Command**
- **Using Groups, Processes, and Threads** in the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

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.

RELATED TOPICS

focus_processes**Command**

focus_threads**Command**

dfocus**Command**

Using Groups, Processes, and Threads in the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

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.

RELATED TOPICS

focus_processes**Command**

focus_threads**Command**

dfocus**Command**

Using Groups, Processes, and Threads in the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

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.

RELATED TOPICS

focus_groups**Command**

dworker**Command**

Using Groups, Processes, and Threads in the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

process**Command**

thread**Command**

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.

RELATED TOPICS

dec2hex**Command**

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 with MemoryScape*™ 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).

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.

parallel_attach_subset

Contains the specification for MPI ranks to be attached to when an MPI job is created or attached to. See **-parallel_attach_subset** *subset_specification*.

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 **x86-64**.

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 mapping sizes 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
```

RELATED TOPICS

Using Groups, Processes, and Threads in the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

focus_processes**Command**

group**Command**

thread**Command**

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_onlyor TV::dll_read_no_symbolslists.

For more information, see "*Preloading Shared Libraries*" in the "Debugging Programs" chapter of the *Classic TotalView User 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 new-line, 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**.

NOTE: 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.

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.

loader_sym_by_regexp: Look up loader symbols using a regular expression to match the base name. For example:

```
TV::scope lookup $scope_id loader_sym_by_regexp {print$}
```

finds all of the loader symbols ending in "print" contained in the given 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.

RELATED TOPICS

Initializing TotalView in the "Loading and Managing Sessions" chapter of the *Classic TotalView User Guide*

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

Table 3lists 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 Perforce Software provided will be the ones that you will use.

Table 3: Symbol Properties

Symbol Kind	Has base_ name	Has type_ index	Property		
aggregate_-type	X	X	aggregate_kind artificial external_name	full_pathname id kind	length logical_scope_owner scope_owner
array_type	X	X	artificial data_addressing element_addressing external_name full_pathnameid	index_type_index kind logical_scope_owner lower_bound scope_owner stride_bound	submembers target_type_index upper_bound validator
block	X		address_class artificial full_pathname	id kind length	location logical_scope_owner scope_owner
char_type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner target_type_index
code_type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
ds_ undis-covered_ type	X	X	artificial id	kind logical_scope_owner scope_owner	target_type_index
enum_type	X	X	artificial enumerators external_name	full_pathname id kind	logical_scope_owner scope_owner value_size
file	X		artificial compiler_kind delayed_symbol demangler	full_pathname idkind language	logical_scope_owner scope_owner

Table 3: Symbol Properties

Symbol Kind	Has base_ name	Has type_ index	Property		
float_type	X	X	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
function_-type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
image	X		artificial full_pathname	id	kind
int_type	X	X	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
label	X		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_sym-bol			address_class artificial full_pathname	id kind length	location logical_scope_owner scope_owner
member	X		address_class artificial full_pathname id	inheritance kind location logical_scope_owner	ordinal scope_owner type_index
module	X		artificial full_pathname	id kind	logical_scope_owner scope_owner
named_-constant	X		artificial full_pathname id	kind length logical_scope_owner	scope_owner type_index value
namespace	X		artificial full_pathname	idkind	logical_scope_owner scope_owner

Table 3: Symbol Properties

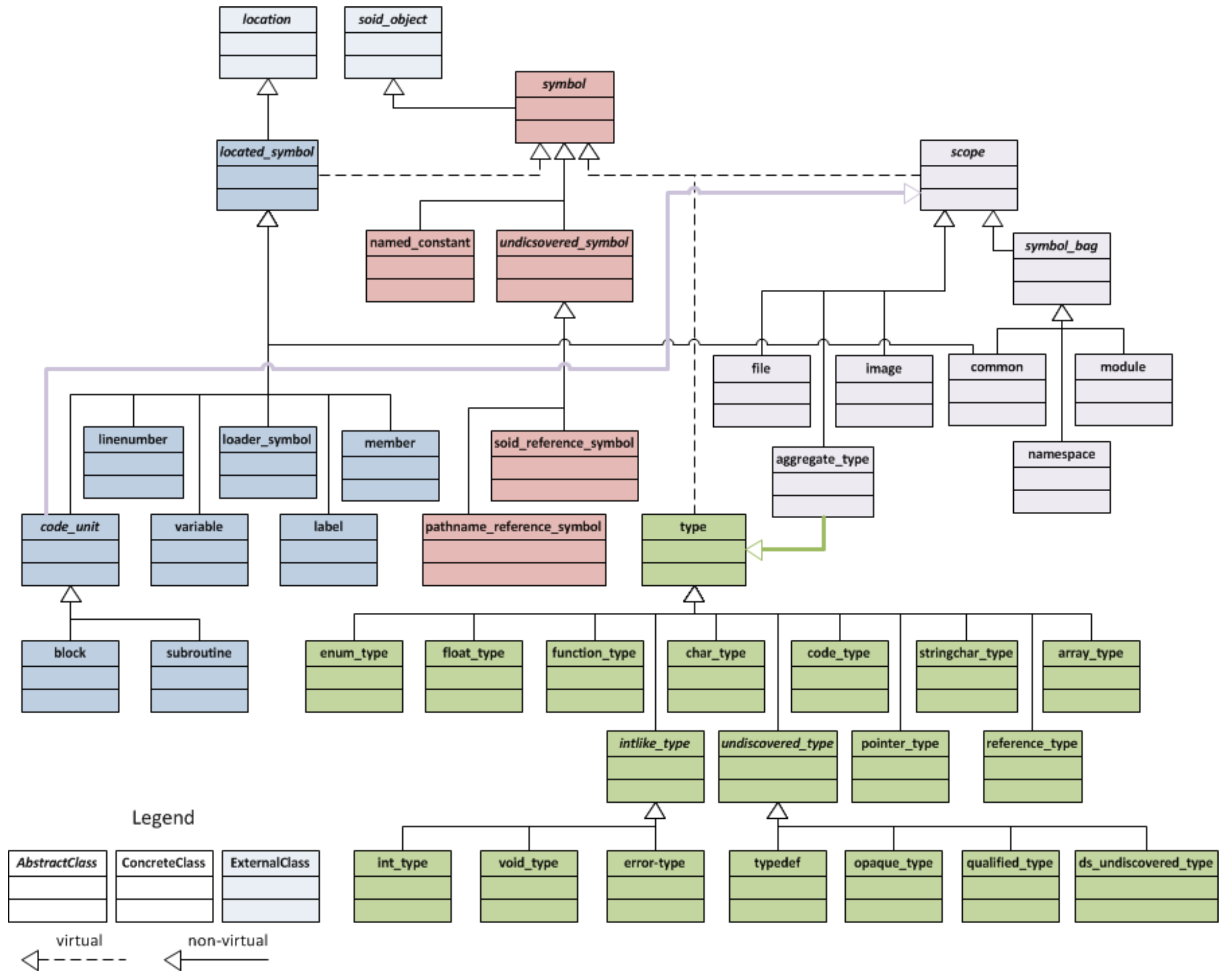
Symbol Kind	Has base_ name	Has type_ index	Property		
opaque_type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
pathname_-reference_symbol	X		artificial id full_pathname	kind lookup_scope logical_scope_owner	resolved_symbol_-pathname scope_owner
pointer_type		X	artificial external_name full_pathname id	kind length logical_scope_owner scope_owner	target_type_index validator
qualified_-type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	qualification scope_owner target_type_index
soid_reference_symbol	X		artificial full_pathname id	kind logical_scope_owner resolved_symbol_id	scope_owner
stringchar_-type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner target_type_index
subroutine	X		address_class artificial full_pathname id	kind length location logical_scope_owner	return_type_index scope_owner static_chain static_chain_height
typedef	X	X	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner target_type_index
variable	X		address_class artificial full_pathname id	is_argument kind location logical_scope_owner	ordinal scope_owner type_index

Table 3: Symbol Properties

Symbol Kind	Has base_ name	Has type_ index	Property		
void_type	X	X	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
wchar_type	X	X	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner target_type_index

Figure 1 on page 255shows how these symbols are related.

Figure 1, Symbols Architecture



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.

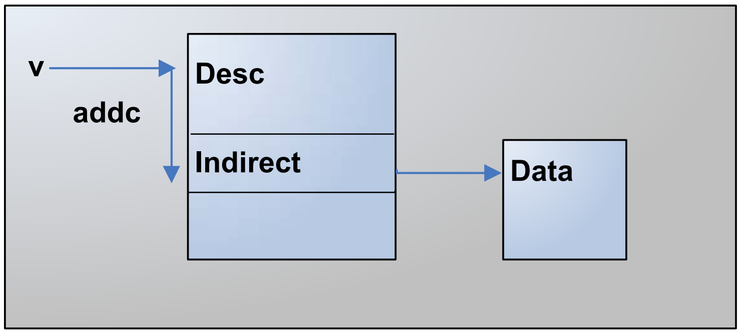
compiler_kind

The compiler or family of compiler used to create the file; for example, **gnu**, **xlc**, **intel**, and so on.

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 2, Data Addressing



delayed_symbol

Indicates if a symbol has been full or partially read-in. The following constants are or'd and returned: **skim**, **in-dex**, **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.

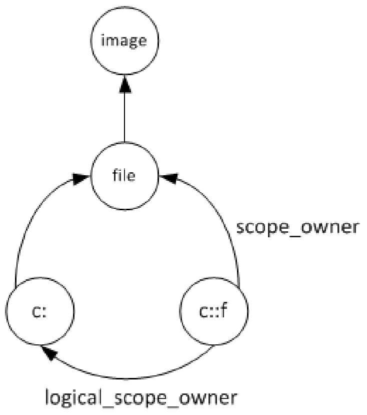
location

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 3, 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.

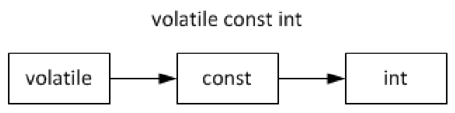
ordinal

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 4, 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. Table 1 lists the properties associated with a namespace. Figure 5 on page 261illustrates how these namespaces are related.

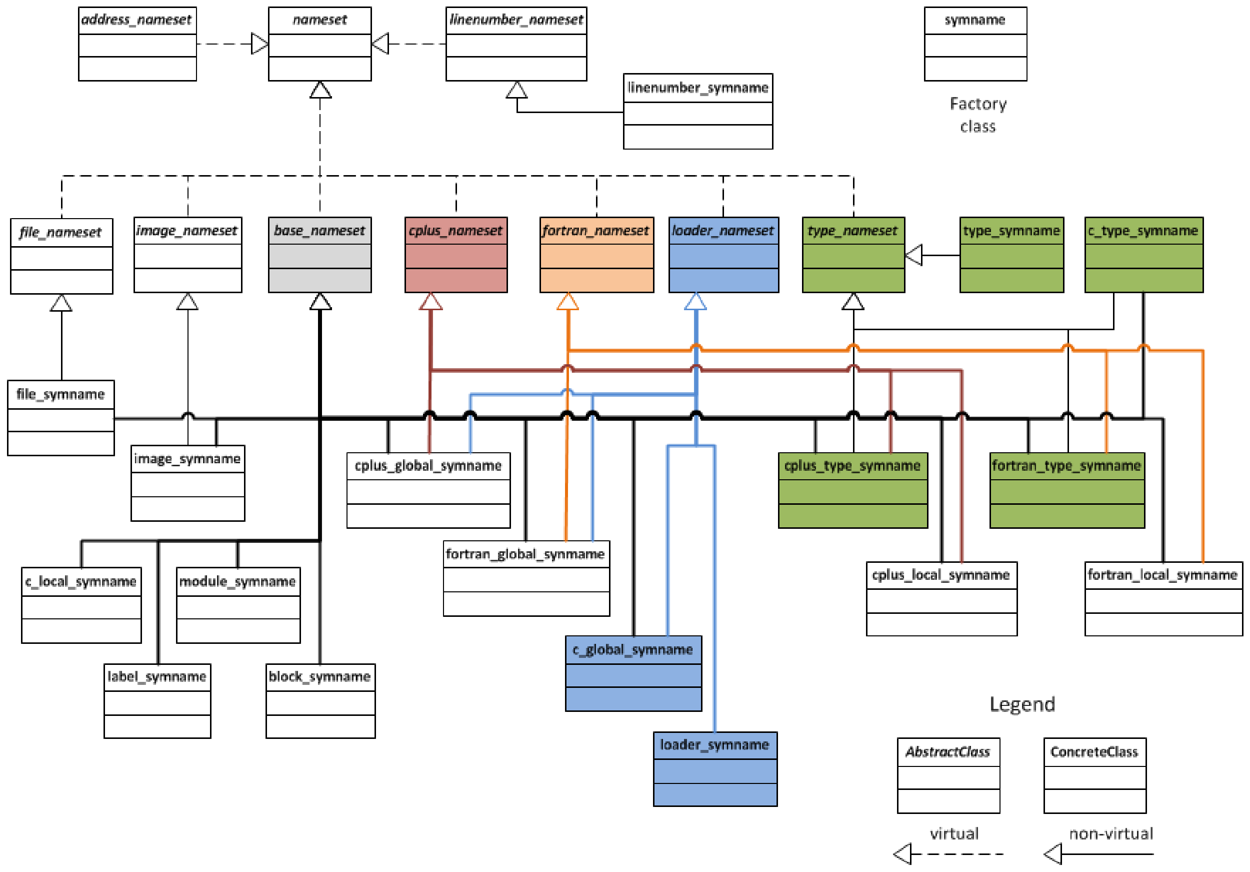
Table 4: Namespace Properties

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	
fortran_global_symname	base_name	loader_file_path

Table 4: Namespace Properties

Symbol Namespaces	Properties	
	fortran_module_name	loader_name
	fortran_parent_function_name	
fortran_local_symname	base_namefortran_parent_func-tion_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	

Figure 5, Namespace Architecture



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. Its 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* sub-commands. 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 examines 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 **x86-64**.

thread_ktid

The kernel thread ID.

thread_name

The name given to a thread by the application.

thread_utid

A user thread ID.

Examples

```
f p3 TV::thread get -all id
```

Return 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

RELATED TOPICS

Using Groups, Processes, and Threads in the *Group, Process, and Thread Control* chapter of the *Classic TotalView User Guide*

focus_threads**Command**

group**Command**

process**Command**

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* sub-commands. 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. For commands without this argument applied 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**, **Set**, **Umap**, **Uset** 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 create 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:

Common Properties

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++.

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** 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".

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 should 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 datatype. The callback procedure must return a Boolean value, where ***true*** means the symbol is valid and can be transformed.

compiler

Reserved for future use.

Array Properties

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** procedure.

This callback defines a TotalView addressing expression that computes the starting address of an array's first element.

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** procedure.

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** procedure.

List Properties

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** 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** 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** 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** 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_first_element_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** 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** procedure.

Struct Properties

struct_member_count_callback

Names the procedure that obtains the total number of members in a struct. The call structure for this callback is:

struct_member_count_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

struct_member_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the specified member of a struct. The call structure for this callback is:

struct_member_addressing_callback *id index*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure and *index* specifies the zero-based position of the member within the struct.

struct_member_type_callback

Names the procedure that obtains the type id of the specified member of a struct. The call structure for this callback is:

struct_member_type_callback *id index*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure and *index* specifies the zero-based position of the member within the struct.

struct_member_name_callback

Names the procedure that obtains the name of the specified member of a struct. The call structure for this callback is:

struct_member_name_callback *id index*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure and *index* specifies the zero-based position of the member within the struct.

Red/Black Tree Properties

The implementation of map/multimap and set/multiset STL types uses red/black trees. These properties are common to all these types.

rbtree_head_addressing_callback

Names the procedure that defines an addressing expression to obtain the head element of the map. The call structure for this callback is:

rbtree_head_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

rbtree_head_type_callback

Names the procedure that obtains the type id of the head of a map. The call structure for this callback is:

rbtree_head_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

rbtree_element_left_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the left sub-tree of the current element of a map. The call structure for this callback is:

rbtree_element_left_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

rbtree_element_right_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the right sub-tree of the current element of a map. The call structure for this callback is:

rbtree_element_right_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

rbtree_element_parent_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the parent of the current element of a map. The call structure for this callback is:

rbtree_element_parent_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

rbtree_element_count_addressing_callback

Names the procedure that determines the total number of elements in a map. The call structure for this callback is:

rbtree_element_count_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** 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 map.

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. Unfortunately, this can be very time consuming.

rbtree_element_count_type_callback

Names the procedure that obtains the type id of the member that specifies the number of elements in the map. The call structure for this callback is:

rbtree_element_count_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

If your data structure does not have a count element, this property is not required.

rbtree_left_most_addressing_callback

Names the procedure that defines an addressing expression to obtain the left-most element of the map. The call structure for this callback is:

rbtree_left_most_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

Map/Multimap Properties

For map and multimap STL types these properties are used in combination with those for red/black trees above.

map_element_key_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the key of an element of a map. The call structure for this callback is:

map_element_key_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

map_element_key_type_callback

Names the procedure that obtains the type id of the key of a map. The call structure for this callback is:

map_element_key_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

map_element_type_callback

Names the procedure that obtains the type id of the element in the red/black tree that contains the key/value pair. The call structure for this callback is:

map_element_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the validate_callback procedure.

map_element_value_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the value of an element of a map. The call structure for this callback is:

map_element_value_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

map_element_value_type_callback

Names the procedure that obtains the type id of the value of a map. The call structure for this callback is:

map_element_value_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

map_iterator_end_value

Specifies if a map is terminated by NULL or the head of the map. Enter one of the following: **NULL** or **MapHead**

Set/Multiset Properties

For set and multiset STL types these properties are used in combination with those for red/black trees above.

set_element_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access an element of a set. The call structure for this callback is:

set_element_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

set_element_type_callback

Names the procedure that obtains the type id of an element in the set. The call structure for this callback is:

set_element_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

set_iterator_end_value

Specifies if a set is terminated by NULL or the head of the set. Enter one of the following: **NULL** or **SetHead**

Hashtable Properties

The implementations of unordered map/multimap and unordered set/multiset STL types use hash tables. These properties are common to all these types.

hashtable_head_addressing_callback

Names the procedure that defines an addressing expression to obtain the head element of the map. Depending on the implementation, this element may be the address of the bucket list or the beginning element of a forward list. The call structure for this callback is:

hashtable_head_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

hashtable_element_count_addressing_callback

Names the procedure that determines the total number of elements in a hashtable. The call structure for this callback is:

hashtable_element_count_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** 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 map.

hashtable_element_count_type_callback

Names the procedure that obtains the type id of the member that specifies the number of elements in the map. The call structure for this callback is:

hashtable_element_count_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

hashtable_element_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the next element. The call structure for this callback is:

hashtable_element_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

hashtable_begin_index_addressing_callback

Names the procedure that determines the index of the first used bucket in a hashtable. The call structure for this callback is:

hashtable_begin_index_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the first used bucket in the hashtable. This allows a small optimization since the transformation can skip empty buckets at the start of the bucket table. If your data does not supply this value you can use **{nop}**.

hashtable_begin_index_type_callback

Names the procedure that determines the type of the value that contains the index of the first used bucket in a hashtable. The call structure for this callback is:

hashtable_begin_index_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

hashtable_bucket_count_addressing_callback

Names the procedure that determines the total number of buckets in a hash table. The call structure for this callback is:

hashtable_bucket_count_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the number of buckets in a hashtable.

This property can be **{nop}** when the hash table elements can be found without scanning the bucket list, for example, when the elements are also stored in a forward list.

hashtable_bucket_count_type_callback

Names the procedure that obtains the type id of the member that specifies the number of buckets in a hash table. The call structure for this callback is:

hashtable_bucket_count_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

If you are not scanning the bucket list for the hashed values, this property is not required.

Unordered Map/Multimap Properties

For unordered map and unordered multimap STL types these properties are used in combination with those for hash tables above.

umap_element_key_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the key of an element of a map. The call structure for this callback is:

umap_element_key_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

umap_element_key_type_callback

Names the procedure that obtains the type id of the key of a map. The call structure for this callback is:

umap_element_key_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

umap_element_type_callback

Names the procedure that obtains the type id of the element in the hashtable that contains the key/value pair. The call structure for this callback is:

umap_element_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

umap_element_value_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the value of an element of a map. The call structure for this callback is:

umap_element_value_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

umap_element_value_type_callback

Names the procedure that obtains the type id of the value of a map. The call structure for this callback is:

umap_element_value_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

Unordered Set/Multiset Properties

For unordered set and unordered multiset STL types these properties are used in combination with those for hash tables above.

uset_element_key_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access an element of a set. The call structure for this callback is:

uset_element_key_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

uset_element_key_type_callback

Names the procedure that obtains the type id of an element in the set. The call structure for this callback is:

uset_element_key_type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback** procedure.

Batch Debugging Using tvscript

TotalView supports batch debugging using the **tvscript** command.

About tvscript

Batch debug programs by starting TotalView using the **tvscript** command, which allows TotalView to run unattended. If you invoke **tvscript** using **cron**, you can schedule debugging for a certain time, for instance in the evening, so reports are available in the morning.

To perform complex actions, use a script file, which can 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 "noshow_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
```

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 options described in TotalView Command Syntax on page 386.

filename

The program being debugged.

-a *program_args*

Program arguments.

The command-line options most often used with **tvscript** are:

- **-mpi**(The MPI environments supported are those listed in the Parallel tab of the **File > New Program** dialog box.)
- **-starter_args**
- **-nodes**
- **-np** or **-procs**or **-tasks**

For more information on these command-line options, see TotalView Command Syntax on page 386.

Cray Xeon Phi

The syntax for using **tvscript** on Cray Xeon Phi Knights Corner (KNC) native nodes is:

tvscript [*options***] -mpi CrayKNC-aprun -np***number-of-processes***-starter_args** "**[***aprun-arguments***]***filename***[***program_args***]**"**aprun**

-np

The number of processes or tasks that the starter program will create.

-starter_args

Required, with the arguments following enclosed in quotes; the application executable (*filename*) to be debugged must follow the arguments for **aprun**.

aprun-arguments

The command arguments for **aprun** (except the **-k** argument).

filename

The program being debugged.

program_args

The arguments for the program being debugged.

aprun

Required; the executable at the end of the command line.

For example:

```
tvscript \
-create_actionpoint "tx_basic_mpi.c#98=>display_backtrace
-show_arguments, print myid" \
-mpi CrayKNC-aprun -np 16 \
-starter_args "tx_basic_mpi" \
aprun
```

Cray XK7

The syntax for using **tvscript** on Cray XK7 is:

tvscript [*options***] -mpi CrayXK7-aprun -np***number-of-processes***-starter_args**
"**[***aprun-arguments***]***filename***[***program_args***]**"**aprun**

-np

The number of processes or tasks that the starter program will create.

-starter_args

Required, with the arguments following enclosed in quotes; the application executable (*filename*) to be debugged must follow the arguments for **aprun**.

aprun-arguments

The command arguments for **aprun**.

filename

The program being debugged.

program_args

The arguments for the program being debugged.

aprun

Required; the executable at the end of the command line.

For example:

```
tvscript \
-create_actionpoint "tx_basic_mpi.c#98=>display_backtrace
-show_arguments, print myid" \
-mpi CrayXK7-aprun -np 16 \
-starter_args "tx_basic_mpi" \
aprun
```

tvscript Options

-create_actionpoint "*source_location_expr* [=>*action1*[, *action2*]...]"

Creates an action point at a source location using an expression. (See Action Point API on page 291for writing expressions.) When the action point is hit, **tvscript** can trigger one or more actions. Add one **-create_actionpoint** command-line option for each action point.

See **-event_action** for information about actions.

-event_action "*event_action_list*"

Performs an action when an event occurs. Events represent an unanticipated condition, such as **free_not_allocated** in the Memory Debugger. You can use more than one **-event_action** command-line option when invoking **tvscript**.

Here is how you enter an *event_action_list* :

event1=*action1*,*event2*=*action2*

or

event1=>*action1*,*action2*,*action3*

Table 5: Supported tvscript Events

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.

Table 5: Supported tvscript Events

Event Type	Event	Definition
	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.
	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.
Reverse debugging	**stopped_at_end**	The program is stopped at the end of execution and is about to exit.

For each occurring event, define the action to perform:

Action Type	Action	Definition
Memory debugging actions	**check_guard_blocks**	Checks all guard blocks and write violations into the log file.
	list_allocations	Writes a list of all memory allocations into the log file.
	list_leaks	Writes a list of all memory leaks into the log file.
	save_html_heap_status_source_view	Generates and saves an HTML version of the Heap Status Source View Report.
	save_memory_debugging_file	Generates and saves a memory debugging file.
	save_text_heap_status_source_view	Generates and saves a text version of the Heap Status Source View Report.
Source code debugging actions	**display_backtrace** [**-level***level-num*] **[** *num_levels* **]** **[** *options***]**	Writes the current stack backtrace into the log file. **-level** *level-num* sets the level at which information starts being logged. *num_levels* restricts output to this number of levels in the call stack. If 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_f**p **-[no]show_fp_registers** **-[no]show_image** **-[no]show_locals** **-[no]show_pc** **-[no]show_registers**

Action Type	Action	Definition
	print [**-slice** {*slice_exp*}] {*variable* | *exp*}	Writes the value of a variable or an expression into the log file. If the variable is an array, the **-slice** option limits the amount of data defined by slice_exp. A slice expression is a way to define the slice, such as **var[100:130]**in C and C++. (This displays all values from **var[100]**to **var[130]**.) To display every fourth value, add an additional argument; for example,**var[100:130:4]**. For additional information, see *"Examining Arrays"*in the *Classic TotalView User Guide*.
Reverse debugging actions	**enable_reverse_debugging**	Turns on ReplayEngine reverse debugging and begins recording the execution of the program.
	save_replay_recording_file	Saves a ReplayEngine recording file. The filename is of the form `<ProcessName>-<PID>_<DATE>.<INDEX>.recording`.

-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:

Type of Specifier	Specifier	Display ...
General display specifiers	**noshow_fp**	Does not show the frame pointer (FP)
	noshow_image	Does not show the process/library in backtrace
	noshow_pc	Does not show the program counter (PC)
	noshow_pid	Does not show the system process ID with process information
	noshow_rank	Does not show the rank of a process, which is shown only for a parallel process
	noshow_tid	Does not show the thread ID with process information

Type of Specifier	Specifier	Display ...
Memory debugging display specifiers	**noshow_allocator**	Does not show the allocator for the address space
	noshow_backtrace	Does not show the backtraces for memory blocks
	noshow_backtrace_id	Does not show the backtrace ID for memory blocks
	noshow_block_address	Does not show the memory block start and end addresses
	noshow_flags	Does not show the memory block flags
	noshow_guard_id	Does not show the guard ID for memory blocks
	noshow_guard_settings	Does not show the guard settings for memory blocks
	noshow_leak_stats	Does not show the leaked memory block statistics
	noshow_owner	Does not show the owner of the allocation
	noshow_red_zones_settings	Does not show the Red Zone entries for allocations (and deallocations) for the address space

-memory_debugging

Enables memory debugging and memory event notification. This option is required with any option that begins 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 returning it to the heap.

-mem_hoard_low_memory_threshold nnnn

Sets the low memory threshold amount. When memory falls below this amount, an event is fired.

-mem_paint_all

Paints memory blocks with a bit pattern when a memory is allocated or 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.

-script_log_filename *logFilename*

Overrides the name of the TVScript log file.

WARNING: Previous log files of the same name are overwritten.

-script_summary_log_filename *summaryLogFilename*

Overrides the name of the TVScript summary log file.

WARNING: Previous summary log files with the same name are overwritten.

-replay

Enables reverse debugging with ReplayEngine on the process through **tvscript**. The entire program's execution is recorded. To turn on recording for a **tvscript** event, use the **enable_reverse_debugging** action.

tvscript Example:

The following example is similar to that shown in Batch Debugging Using tvscript on page 280.

```
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 performs the following actions:

- Creates an action point at the beginning of **method1**. When **tvscript** reaches that breakpoint, it logs a backtrace and the method's arguments.
- Creates an action point 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.
- Logs the backtrace, the current routine's arguments, and its local variables when an error event occurs.
- Excludes the process ID and thread ID from the information that **tvscript** logs.
- Limits **tvscript** execution time to 30 seconds.
- Names the program being debugged and passes a value of 20 to the application.

tvscript Reverse Debugging Example:

```
tvscript \
-create_actionpoint "main=>enable_reverse_debugging" \
-event_action "stopped_at_end=>save_replay_recording_file" \
filterapp
```

This script performs the following actions:

- Creates an action point on method `main`. When the action point is hit, reverse debugging is enabled and recording of the program begins.
- Specifies that the recording file is to be saved if the **stopped_at_end** event is raised.

tvscript External Script Files

The section tvscript Command Syntaxdiscussed the command-line options used when invoking the **tvscript** command. You can also place commands in a file and provide them to **tvscript** using the**-script_file** command-line option. Using a script file supports the use of 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*

Logs a message to the log file set up by **tvscript**.

tvscript_slog *msg*

Logs a message to the summary log file set up by **tvscript**.

Process Functions API

tvscript_get_process_property *process_id property*

Gets 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 239for more information.

Thread Functions API

tvscript_get_thread_property *thread_id property*

Gets 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 263for more information.

Action Point API

tvscript_add_actionpoint_handler *actionpoint_id actionpoint_handler*

Registers a procedure handler to call when the action point associated with *actionpoint_id* is hit. This *actionpoint_id* is the value 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 the 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

Sets 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 with 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 that 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 to call 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 example **tvscript** file registers several action point handlers when an action point (breakpoint) is hit. When the handlers are called, they display information about the action point event. The script also installs an error handler, which is called if an error occurs during execution of the program. Run the script as follows:

tvscript -script_file *script_file program*

Where ***program*** is the name of the program to run under the control of **tvscript**. This **tvscript** example, `tvscript_example.tvd`, is available in the TotalView examples directory.

This script installs an error handler and an action point handler. When an error is encountered during execution, **tvscript** passes an array of information to the error handler and prints information to the log. Similarly, when an action point is hit, it passes an array of information to the action point handler and prints information to the log. These arrays are described in Event API on page 293.

```
# 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
  set temp [tvscript_get_process_property \
  $error_data_array(process_id) "syspid"];
  tvscript_log " Process ID: $temp";

  set temp [tvscript_get_thread_property \
  $error_data_array(thread_id) "systid"];
  tvscript_log " Thread ID: $temp";

  set temp $error_data_array(error_message);
  tvscript_log " Error Message: $temp";

  }

###############################################################
# Action point handlers

proc actionpoint_handler {event_data} {
tvscript_log "Inside actionpnt_handler: $event_data"
tvscript_slog "Inside 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
set temp [tvscript_get_process_property \
$event_data_array(process_id) "syspid"];
tvscript_log " Process ID: $temp";

set temp [tvscript_get_thread_property \
$event_data_array(thread_id) "systid"];
tvscript_log " Thread ID: $temp";

set temp [tvscript_get_process_property \
$event_data_array(process_id) "state"];
tvscript_log " Status: $temp";

set temp [tvscript_get_process_property \
$event_data_array(process_id) "executable"]
tvscript_log " Executable: $temp";

set 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

set temp $actionpnt_data_array(process_id)
tvscript_log " Process ID: $temp"

set temp $actionpnt_data_array(thread_id)
tvscript_log " Thread ID: $temp"

set temp $actionpnt_data_array(actionpoint_id)
tvscript_log " Action Point ID: $temp"

set temp $actionpnt_data_array(actionpoint_source_loc_expr)
tvscript_log "Action Point Expression: "
}

##############################################################
# Add event handlers

# Create a breakpoint on function main and register
# procedure "actionpoint_handler" to be called if the
# breakpoint is hit.
set actionpoint_id [tvscript_create_actionpoint "main"]
tvscript_add_actionpoint_handler $actionpoint_id \ "actionpoint_handler"

# Set up a generic actionpoint handler that is called
# whenever any action point is hit.
tvscript_add_event_handler "actionpoint" \ "generic_actionpoint_event_handler"

########################################################
# Add error handler that is called if an error
# occurs while running the program.
tvscript_add_event_handler "error" "error_handler"
```

TotalView Variables

This chapter contains a list of all CLI and TotalView variables, organized into sections that each correspond to a CLI namespace.

Top-Level (::) Namespace

ARGS(*dpmid*)

Contains the arguments to be passed the next time the process starts, with TotalView ID ***dpid***.

Permitted Values: A string

Default: None

BARRIER_STOP_ALL

*C*ontains the value for the “stop_when_done” property for newly created action points. This property defines additional elements to stop when a barrier point is satisfied or 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 are:

group

Stops all processes in a thread’s control group when a thread reaches a barrier created using this default.

process

Stops the process in which the thread is running when a thread reaches a barrier created using this default.

none

Stops only 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**

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_WHEN_DONE

Contains the default value used when a barrier point is satisfied. You can also set this value using 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 are:

group

When a barrier is satisfied, stops all processes in the control group.

process

When a barrier is satisfied, stops the processes in the satisfaction set.

none

Stops only 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, TotalViewreleases 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 for use when editing text in the CLI. These editing commands are always available in the CLI window of the TotalView GUI. However, they are available only in the stand-alone CLI if the terminal in which it is running supports cursor positioning and clear-to-end-of-line. The commands 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 of the directories searched 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 entered is the delimiter. This example uses a "**+"** as a delimiter. (Traditionally, forward slashes are used as delimiters but are not used here, as a forward slash is also used to separate components of a pathname. For example, **/home/my_dir** contains forward slashes.)

Be aware that special characters must follow standard Tcl rules and conventions, for example:

```
dset EXECUTABLE_SEARCH_MAPPINGS {+^/nfs/compiled/u2/(.*)$+ = +/nfs/host/u2/\1+ }
```

This expression applies a mapping so that a directory named **/nfs/compiled/u2/project/src1** in the expanded search path becomes **/nfs/host/u2/project/src1**.

Default: {}

EXECUTABLE_SEARCH_PATH

Contains a list of paths, separated by a colon, to search for executables. For information, see "Setting Search Paths Using Classic TotalView Variables" in the Classic TotalView in-product help.

Permitted Values: Any directory or directory path.

Default: **${EXECUTABLE_PATH};${$PATH}:.**

GROUP(*gid*)

Contains a list of the TotalView IDs for all members in group ***gid***.

The first element indicates the type of group:

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 ***pid***s (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 does not display lockstep groups when you use **dset** with no arguments because 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 of 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

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 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 displays.

Permitted Values: A positive integer

Default: 512

MAX_LIST

Defines the number of lines that the **dlist** command displays.

Permitted Values: A positive integer

Default: 20

OBJECT_SEARCH_MAPPINGS

Contains pairs of regular expressions and replacement and replacement strings (called mappings) separated by colons. TotalView applies these mappings to the search paths when searching for source, object, and program files. For more information, see EXECUTABLE_SEARCH_MAPPINGS.

Default: {}

OBJECT_SEARCH_PATH

Contains a list of paths separated by a colon to search for your program's object files. For information, see "Search Path Variables That You Can Set" in the Classic TotalView in-product 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. Any information within brackets (**[ ]**) is assumed to be a Tcl command, so therefore evaluated before the prompt string is created.

Permitted Values: Any string. To access the value of **PTSET**, 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*. The share group is determined by 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 for 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 **false**, a newly created action point is active only 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. To override this value in the GUI, use 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, to search 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 (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, separated by a colon, to search for your program's source files. For information, see "Search Path Variables That You Can Set" in the Classic TotalView in-product help.

Permitted Values: Any directory or directory path.

Default: **${COMPILATION_DIRECTORY}: ${EXECUTABLE_PATH}: ${EXECUTABLE_DIRECTORY}: ${links{${EXECUTABLE_DIRECTORY}): .:${TOTALVIEW_SRC}**

STACK_TRACE_TRANSFORM_ENABLED

Controls whether TotalView filters the stack. Because not all applications can benefit from stack filtering, this variable is **false** by default.

NOTE: If TotalView detects that an application has a feature that can benefit from stack filtering, it enables this variable.

Permitted Values: **true** or **false**

Default: **false**

STOP_ALL

Indicates a default property for newly created action points, defining additional elements to stop when this action point is encountered

group

Stops the entire control group when the action point is hit

process

Stops the entire process when the action point is hit

thread

Stops only 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 containing the TotalView executable. This is a read-only variable and cannot be set. This variable is exported as **TVROOT**, and can be used in launch strings.

Permitted Values: The location of the TotalView installation directory

TOTALVIEW_TCLLIB_PATH

Contains a list of 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 version number

Default: Platform-specific

VERBOSE

Sets the error message information displayed by the CLI:

info

Prints errors, warnings, and informational messages. Informational messages include data on dynamic libraries and symbols.

warning

Prints only errors and warnings.

error

Prints only error messages.

silent

Does not print error, warning, and informational messages. This also shuts off printing results from CLI commands. This should be used only 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 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**, providing 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_dlopen

If **true**, TotalView asks about stopping processes that use the **dlopen** or **load** (AIX only) system calls dynamically load a new shared library.

If **false**, TotalView does not ask about stopping a process that dynamically loads a shared library.

Permitted Values:	**true** or **false**
Default:	**false**

TV::auto_array_cast_bounds

Indicates the number of array elements to display when the **TV::auto_array_cast_enabled**variable is **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 **true**, TotalViewautomatically dereferences 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

Defines if and how to dereference C and C++ pointers when performing a **View > Dive in All** operation, as follows:

yes_dont_push

While automatic dereferencing will occur, does not allow use of the **Undive** command to see the underefer-enced value when performing a **Dive in All** operation.

yes

Allows use of the **Undive** control to see undereferenced values.

no

Does 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 > Prefer-ences** 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, does not allow use of the **Undive** command to see the underefer-enced value when performing a **Dive in All** operation.

yes

Allows use of the **Undive** control to see undereferenced values.

no

Does 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 > Prefer-ences** dialog box.

Permitted Values: **no**, **yes**, or **yes_dont_push**

Default: **no**

TV::auto_deref_initial_c

Defines if and how to dereference C pointers when they are displayed, as follows:

yes_dont_push

While automatic dereferencing will occur, does not allow use of the **Undive** command to see the underefer-enced value.

yes

Allows use of the **Undive** control to see undeferenced values.

no

Does 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

Defines if and how to dereference Fortran pointers when they are displayed, as follows:

yes_dont_push

While automatic dereferencing will occur, does not allow use of the **Undive** command to see the underefer-enced value.

yes

Allows use of the **Undive** control to see undereferenced values.

no

Does 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

Defines if and how to 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 undereferenced 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

Defines if and how to dereference Fortran pointers when they are displayed:

yes_dont_push

While automatic dereferencing will occur, does not allow use of the **Undive** command to see the underefer-enced value.

yes

Allows use of the **Undive** control to see undereferenced values.

no

Does 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

If **true**, TotalViewautomatically loads action points from the file named *filename*.**TVD.v4breakpoints** where *file-name* is the name of the file being debugged. If **false**, breakpoints are not automatically loaded. If you set this to **false**, you can still load breakpoints using the **dactions -load** command.

Permitted Values: **true** or **false**

Default: **true**

TV::auto_read_symbols_at_stop

If **false**, TotalView does not automatically read symbols if execution stops when the program counter is in a library whose symbols were not read. If **true**, TotalView reads 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 unnec-essary and would affect performance.

Permitted Values: **true** or **false**

Default: **true**

TV::auto_save_breakpoints

If **true**, TotalView automatically writes information about breakpoints to a file named *filename* **.TVD.v4break-points**, 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 you set this variable to **false**, you can explicitly save this information by using the **dactions-save** command.

Permitted Values: **true** or **false**

Default: **false**

TV::barrier_stop_all

Contains the value of the "stop_all" property for newly created action points. This property defines additional elements to 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

Stops all processes in a thread's control group when a thread reaches a barrier created using this as a default.

process

Stops the process in which the thread is running when a thread reaches a barrier created using this default.

thread

Stops only 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 defines additional elements to stop when a barrier point is satisfied. You can also set this value using 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:

group

When a barrier is satisfied, stops all processes in the control group.

process

When a barrier is satisfied, stops the processes in the satisfaction set.

thread

Stops only 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**

Default:

TV::bulk_launch_base_timeout

Defines the base timeout period used to execute a bulk launch.

Permitted Values: A number from 1 to 3600 (1 hour)

Default: 20

TV::bulk_launch_enabled

If **true**, uses bulk launch features when automatically launching 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 to wait for a process to launch when automatically launching 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_tmpfile1_header_line

Defines the header line used in the first temporary file for a bulk server launch operation.

Permitted Values: A string

Default: None

TV::bulk_launch_tmpfile1_host_lines

Defines the host line used in the first temporary file when performing a bulk server launch operation.

Permitted Values: A string

Default: **%R**

TV::bulk_launch_tmpfile1_trailer_line

Defines the trailer line used in the first temporary file when performing a bulk server launch operation.

Permitted Values: A string

Default: None

TV::bulk_launch_tmpfile2_header_line

Defines the header line used in the second temporary file when performing a bulk server launch operation.

Permitted Values: A string

Default: None

TV::bulk_launch_tmpfile2_host_lines

Defines the host line used in the second temporary file when performing a bulk server launch operation.

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 performing a bulk server launch operation.

Permitted Values: A string

Default: None

TV::c_type_strings

If **true**, uses C type string extensions to display character arrays; when **false**, uses string type extensions.

Permitted Values: **true** or **false**

Default: **true**

TV::check_unique_id

In concert with **TV::checksum_libraries**, controls whether TotalView checksums an image or uses a unique ID to determine the uniqueness of images across nodes in a parallel debugging session.

When **true** (the default), TotalView attempts to extract a unique ID from an image file before checksumming it.The extracted unique ID is a build ID on Linux, or a UUID on macOS. Note that not all image files contain a unique ID by default, but linkers can often add one, for example, by using the linker option `--build-id`. On macOS, a UUID is added to the image file by default, but the linker can control whether it adds a UUID, a checksum, or a random value.

When **false**, *or* if an image file does not contain a unique ID, *or* there is an error extracting the unique ID, TotalView falls back to checksumming to determine image file uniqueness, according to the setting on **TV::checksum_libraries**.

*Permitted Values:***true** or **false**

Default: **true**

TV::checksum_libraries

Controls image checksumming across nodes in a parallel debugging session. If set to **auto** (the default) or **true**, TotalView checksums on both the server and the client. If **false**, TotalView checksums the image only on the client, which should be used only if all image files across all the nodes in the debug session are known to be identical. Performing a checksum only on the client can improve startup times in some cases, depending on the system.

Even when set to **auto** or **true**, whether TotalView performs a checksum to validate images is impacted by the setting for TV::check_unique_id.

> *Permitted Values:* **auto**, **true** or **false**
>
> *Default:* **auto**

TV::comline_patch_area_base

Allocates the patch space dynamically at the given ***address***. See "Allocating Patch Space for Compiled Expressions" in the *Classic TotalView User 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 the *Classic TotalView User Guide*.

> *Permitted Values:* A positive number
>
> *Default:* 0

TV::command_editing

Enables some Emacs-like commands for use while editing text in the CLI. These editing commands are always available in the CLI window of TotalView UI. However, they are available only within the stand-alone CLI if the terminal in which it is running 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 **true**, TotalViewenables compiled expressions. If **false**, TotalView interprets your expression.

On an IBM AIX system, you can use the **-aix_use_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: **false**

TV::compiler_vars

(SGI only) When **true**, TotalView shows variables created by your Fortran compiler as well as the variables in your program. When **false** (which is the default), TotalView does not show the variables created by your compiler.

SGI 7.2 Fortran compilers write debugging information that describes variables 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 with these 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 **true**, TotalViewkills attached processes and exits. When **false**, TotalView can sometimes better manage the way it kills parallel jobs when it works with management systems. This has been tested only with SLURM and 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

A read-only string containing the copyright information displayed when you start the CLI and TotalView.

TV::cppview

If **true**, the C++View facility allows the formatting of program data in a more useful or meaningful form than the concrete representation visible by default when you inspect data in a running program. For more information on using C++View, see C++View on page 371.

*Permitted Values:***true** or **false**

Default: **true**

TV::cuda_debugger

Indicates whether cuda debugging is currently enabled. This is a read-only variable.

Permitted Values: **true** or **false**

Default: **true**

TV::current_cplus_demangler

Setting this variable overrides the C++demangler used by default. Note that this value is ignored unless you also set the value of the TV::force_default_cplus_demangler variable. The following values are supported:

- **gnu_dot**: GNU C++ Linux x86
- **gnu_v3**: GNU C++ Linux x86
- **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 used by default. Note that this value is ignored unless you also set the value of the TV::force_default_f9x_demangler variable. The following values are supported:

- **intel**: Intel Fortran 9x

Permitted Values: A string naming the compiler

Default: Derived from your platform and information within your program

TV::data_format_double

Defines the format to use when displaying double-precision values. This is one of a series of variables that define how to display data. The format of each is similar:

{presentation format-1 format-2 format 3}

presentation

Selects which format to use 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 is either **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 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 used in the **printf()** function.

# (pound)

Displays a 0x prefix for hexadecimal and 0 for octal formats. This isn't used within floating-point formats.

0 (zero)

Pads 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)

Left-justifies 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 to use 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 to use 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 to use 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 to use 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 to use 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 to use 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 to use 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::dbfork

When **true**, TotalView catches the**fork()**, **vfork(),** and **execve()** system calls if your executable is linked with the **dbfork** library. See Linking with the dbfork Library on page 416.

Permitted Values: **true** or **false**

Default: **true**

TV::default_breakpoints_pending

When **true**, **dbreak** and **dbarrier** create pending action points, as if the **-pending** option had been set. The default is **false**. It is not recommended to set it to **true** because it suppresses catching input user errors.

For example, if you want to set a breakpoint on a function `foo`, but you typed `voo` instead, a pending breakpoint is immediately set on the function `voo`, which would not be your intention.

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: **ssh -x**

TV::default_parallel_attach_subset

Names the default subset specification listing MPI ranks to attach to when an MPI job is created or attached to.

Permitted Values: A string indicating the default subset specification.

Default: Initialized to the value specified with the **-default_parallel_attach_subset** command line option.

TV::default_stderr_append

When **true**, TotalView appends the target program's **stderr** information to the file set in the GUI, by the **-stderr** command-line option, or in the **TV::default_stderr_filename** variable. If no pathname is set, the value of this variable is ignored. If the file does not exist, TotalView creates it.

Permitted Values: **true** or **false**

Default: **false**

TV::default_stderr_filename

Names the file to which to write the target program's **stderr** information. If the file exists, TotalView overwrites it. If the file does not exist, TotalView creates it.

Permitted Values: A string indicating a pathname

Default: None

TV::default_stderr_is_stdout

When **true**, TotalView writes the target program's **stderr** information to the same location as **stdout**.

Permitted Values: **true** or **false**

Default: **false**

TV::default_stdin_filename

Names the file from which the target program reads **stdin** information.

Permitted Values: A string indicating a pathname

Default: None

TV::default_stdout_append

When **true**, TotalView appends the target program's **stdout** information to the file set in the GUI, by the **-stdout** command-line option, or in the **TV::default_stdout_filename** variable. If no pathname is set, the value of this variable is ignored. If the file does not exist, TotalView creates it.

Permitted Values: **true** or **false**

Default: **false**

TV::default_stdout_filename

Names the file to which to write the target program's **stdout** information. If the file exists, TotalView overwrites it. If the file does not exist, TotalView creates it.

Permitted Values: A string indicating a pathname

Default: None

TV::display_assembler_symbolically

When **true**, TotalView displays assembler locations as **label+offset**. When **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 that will not result in a query to stop the process when 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

Always reads 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.

Because this is the default behavior, include only 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 TotalViewloads libraries named in this variable, it reads only loader symbols. Because TotalView checks and processes the names in **TV::dll_read_all_symbols** list before it processes this list, it ignores 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 do not need to debug 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 does 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 ignores 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 are not known for these libraries, stack back-traces 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 stop the current process when it loads a library file with this suffix.

You must confirm that you want 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::dlopen_always_recalculate

When **true**, breakpoint specifications are reevaluated on every **dlopen** call (the default). When **false**, this variable enables **dlopen** event filtering in combination with the optional use of TV::dlopen_recalculate_on_match. For details on **dlopen** event filtering, see dlopen Options for Scalability on page 429.

*Permitted Values:***true** or **false**

Default: **true**

TV::dlopen_recalculate_on_match

Contains a *glob-list* of patterns used to match against the path name of a *dlopened* library. If TV::dlopen_always_recalculate is set to **true**, the value of this variable is ignored.

For a complete explanation of **dlopen** event filtering, including use-case examples, please refer to dlopen Options for Scalability on page 429.

*Permitted Values:*String

Default: **""**, the empty string

TV::dlopen_read_libraries_in_parallel

When **false**, (the default), TotalView handles **dlopen** events in the target application serially. (Note that for parallel applications, handling **dlopen** events serially can degrade debugger performance.)

When **true**, TotalView attempts to handle **dlopen** events in parallel.

On non-MRNet platforms, or if MRNet is not enabled, then the value of this variable is ignored. For more information, see "Handling dlopen Events in Parallel".

Permitted Values: **true** or **false**

Default: **false**

TV::dump_core

When **true**, a core file is created when an internal TotalView error occurs. This is used only 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::dwarf_global_index

When **true**, TotalView can use the DWARF global index sections (**.debug_pubnames**, **.debug_pubtypes**, **.debug_typenames**, etc.) in executable and shared library image files. It may be useful to set this flag to **false** if you have an image file that has incomplete global index sections, and you want to force TotalView to skim the DWARF instead, which may cause TotalView to slow down when indexing symbol tables. You can override this variable's value by using the following command-line options:

-dwarf_global_index sets this variable to **true**

-no_dwarf_global_index sets this variable to **false**

Permitted Values: **true** or **false**

Default: **true**

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:

- **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 states that all elements are displayed except for a parent function. For more information on these arguments, see symbol on page 249.

Permitted Values: One or more of the arguments listed above.

Default: **class_name+template_args+module+ parent_function+member+node**

TV::dynamic

When **true**, TotalViewloads symbols from shared libraries. This variable is available on all platforms supported by Perforce Software. (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 and is set by using the **-env** command-line option.For more information, see **-env** *variable=value* on page 391.

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::exec_handling

Defines how TotalView responds when a process being debugged calls **execve()**. This variable is comprised of an Tcl list of ***regexp*** and ***action*** pairs, called an ***exec-handling-list***. The ***regexp*** contains the name of the parent process, and ***action*** defines an action for TotalView to take. For more information, see "Controlling fork, vfork, and execve Handling" in the *TotalView User Guide*.

- *regexp*: A regular expression. The regular expression is not anchored, so use "**^**" and "**$**" to match the beginning or end of the process name.
- *action*: The action to take, as follows:

Action	Description
halt	Stop the process
go	Continue the process
ask	Ask whether to stop the process

Permitted Values:exec-handling-list

Default: None

TV::follow_clone

When a value greater than 0, allows TotalView to pickup threads created using the **clone()** system call. The supported values are:

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.

NOTE: Linux threads 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 **true**, TotalView uses the demangler set in the TV::current_cplus_demangler variable. Set this variable only if TotalView uses the wrong demangler which may occur if you are using an unsupported compiler, an 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 **true**, TotalView uses the demangler set in the TV::current_fortran_demangler variable. Set this variable only if TotalView uses the wrong demangler which may occur if you are using an unsupported compiler, an unsupported language preprocessor, or if your vendor has made changes to your compiler.

Permitted Values: **true** or **false**

Default: **false**

TV::fork_handling

Defines how TotalView responds when a process being debugged calls **fork()** or **vfork()** to attach to new processes. This variable is comprised of a Tcl list ***fork-handling-list*** of ***regexp*** and ***action*** pairs. The ***regexp*** contains the name of the parent process, and ***action*** defines an action for TotalView to take. For more information, see "Controlling fork, vfork, and execve Handling" in the *TotalView User Guide*.

- ***regexp***: A regular expression. The regular expression is not anchored, so use "**^**" and "**$**" to match the beginning or end of the process name.
- ***action***: The action to take, as follows:

Action	Description
attach	Attach to the new child processes.
detach	Detach from the new child processes.

Permitted Values: fork-handling-list

Default: None

TV::gdb_index

When **true**, TotalView can use the **.gdb_index** section in executable and shared library image files. It may be useful to set this to **false** if you have an image file that has an incomplete **.gdb_index** section and you want to force TotalView to skim the DWARF instead. You can override this variable's value by using the following command-line options:

-gdb_index sets this variable to **true**

-no_gdb_index sets this variable to **false**

Permitted Values: **true** or **false**

Default: **true**

TV::global_typenames

When **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 automatically displays the variable by using the non-opaque type definition when displaying variables declared with the opaque type.

If **false**, TotalView does ***not*** assume that type names are globally unique within a program. Use this variable only 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 **true**, TotalView checks for a **.note.gnu.build-id** NOTE section and a **.gnu_debuglink** section within your image files, in that order. If found, it looks for the file named in these sections. If **false**, TotalView ignores the contents of these sections, meaning that debug information from a separate file will not be loaded.

For more information, see the section "Maintaining Debug Information Separate from an Executable" in the *TotalView User Guide*.

Permitted Values: **true** or **false**

Default: **true**

TV::gnu_debuglink_build_id_search_path

This state variable contains a colon-separated search path that TotalView uses to search for separate debug information files using the build ID method.

Each component of this search path is expanded similarly to TV::gnu_debuglink_search_path. However, this variable is used only when searching for separate debug information files using the build ID method, and does not affect searching for separate debug information files using the debug link method.

A build ID is created using the **--build-id** option passed to the **ld** linker. By default, it is a SHA1,160-bit string (40 hex characters) stored in a **.note.gnu.build-id** NOTE section in an image file.

See the following URL for details: https://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html

For a build ID such as:

```
be0e052ddd73fd5f3f15975ac02f4ca903d9bf77
```

TotalView searches for:

`.build-id/be/0e052ddd73fd5f3f15975ac02f4ca903d9bf77.debug` relative to each expanded path component.

You can set or override this variable value in a startup file or with the command-line option -**gnu_debuglink_build_id_search_path**.

For more information, see the section “Using gnu_debuglink Files” in the *TotalView User Guide*.

Permitted Values: A colon-separated build ID search path

TV::gnu_debuglink_check_build_id
TV::gnu_debuglink_checksum

These two boolean variables work in concert to define how to validate a separate debug info file, if one exists, against a base image file that references it. (See the appendix section “Using gnu_debuglink Files” in the *TotalView User Guide*.) These settings may impact performance, depending on the system: Checksumming generally uses more resources than comparing build IDs, so the default setting attempts to compare the build IDs first before falling back and checksumming the separate debug info file.

Table 6 details how these settings work together.

Permitted Values: **true** or **false**

Default: Both settings default to **true**.

Table 6: Validating separate debug files

TV::gnu_debuglink_check _build_id	TV::gnu_debuglink _checksum	Result
True	**True**	***The default***. Compare build IDs if the base image file contains a build ID; otherwise, compare checksums. This option never compares both the build ID and the checksum.
True	**False**	Compare build IDs if the base image file contains a build ID. If the base image file has no build ID, then no comparison occurs, and TotalView uses the separate debug info file. If the base image file has a build ID but there is no match, or the separate debug info file has no build ID, the debug info file is rejected. Choose this option only if you know that the base image files contain a build ID or that the separate debug info files match.
False	**True**	Compare checksums only; never compare build IDs. This option may be useful if the build IDs are considered unreliable.
False	**False**	Compare neither build IDs nor checksums, and unconditionally use the separate debug info files. Choose this option only if you know that the separate debug info files match.

TV::gnu_debuglink_global_directory

Names the global directory containing separate debug files. For more information, see the appendix section “Using gnu_debuglink Files” in the *TotalView User Guide*.

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_search_path

Defines the search path to use when searching for debug files. You can use substituting variables when assigning values:

- **%D**: The directory containing the image file. Note that the directory ends in the target directory delimiter, for example "/".
- **%G**: The contents of the **TV::gnu_debuglink_global_directory** variable.
- **%/**: The target directory delimiter; for example "**/**".
- **%%**: A '%' character.

You can set or override this variable value in a startup file or with the command-line option **-gnu_debuglink_search_path**.

For more information, see the section "Using gnu_debuglink Files" in the *TotalView User Guide*.

Permitted Values: A string containing directory paths.

Default: **%D:%D.debug:%G%/%D**

TV::hia_local_dir

This variable affects only those cases where TotalView preloads the agent. It names the directory in which TotalView will look for the **hia** for a local job. The default is the value of **TV::hia_local_installation_dir**. Change this variable if you want TotalView to look for the agent in a different directory.

TV::hia_local_installation_dir

A read-only variable that names the directory where the **hia** distributed with the executing instance of TotalView is found.

TV::hia_remote_dir

This variable affects only those cases where TotalView preloads the agent. It names the directory on a remote host where TotalView will look for the **hia** that is to be used by the remote job. If the variable is not set, the server uses its default, which is the same as the default value of the server's **TV::hia_local_dir** but is interpreted in the remote file system.

TV::hpf

Deprecated.

TV::hpf_node

Deprecated.

TV::host_platform

A read-only value that returns the architecture upon which TotalView is running.

TV::ignore_control_c

When **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 be invoked only while TotalView is being initialized. This is a read-only variable.

Permitted Values: **true** or **false**

Default: **false**

TV::ipv6_support

When **true**, ipv6 support is enabled. If **false**, ipv6 support is disabled.

Permitted Values: **true** or **false**

Default: **false**

TV::jit_debugging

When **true**, Clang / LLVM JIT (just-in-time compiled) code support is enabled through the GDB JIT debugging interface.

When JIT code is dynamically loaded into a process, TotalView reads the symbol table information for the JIT code and places it into a separate symbol table image. JIT images are handled similarly to dynamically loaded shared libraries, except that they are given a synthetic name starting with "@TEMP@JIT@". The JIT images appear after the executable and shared library images in the process' image list.

When JIT code is dynamically unloaded from a process, TotalView removes the corresponding JIT image from the process' image list.

Pending breakpoints can be created for the JIT code before it is dynamically loaded into the process. For example, if a pending breakpoint exists for a JIT code function called “compute_factorial”, execution stops when that JIT function is reached.

When **false**, Clang / LLVM JIT code support is disabled and TotalView does not hook JIT events or read JIT code symbol tables.

NOTE: Support for JIT debugging is limited to CLANG LLVM on Linux-x86_64.

You can override this variable’s value by using the following command-line options:

-jit_debugging sets this variable to **true**

-no_jit_debugging sets this variable to **false**

Permitted Values: **true** or **false**

Default: **true**

TV::jnibridge

Internal use only.

TV::kcc_classes

When **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 **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 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 just 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 to 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 process's 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 kills 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 to write 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 TV::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) This variable tells TotalView which local server it should launch. 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.

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.

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 **true**, TotalView displays MPI message queues when you are debugging an MPI program. When **false**, these queues are not displayed. Disable these queues only if something is overwriting the message queues, thereby confusing TotalView.

Permitted Values: **true** or **false**

Default: **true**

TV::mrnet_enabled

When **true**, TotalView enables MRNet on platforms where it is supported (Linux-x86_64, Linux PowerLE, and Cray). To disable the MRNet infrastructure when debugging an MPI job, set this variable to **false**.

Permitted Values: **true** or **false**

Default: **true**

TV::mrnet_port_base

The start of the port range that MRNet attempts to use for listening sockets on Cray systems. This string is passed to MRNet instead of using the **MRNET_PORT_BASE** environment variable. This value is only used when TotalView uses MRNet on Cray systems.

Permitted Values: A port number

Default: {}

TV::mrnet_proxy_server

Controls the use of an MRNet proxy server.

- **auto**: Use an MRNet proxy server only if necessary. Default
- **true**: Use an MRNet proxy server, even if unnecessary. Useful for testing and debugging the proxy server.
- **false**: Do not use an MRNet proxy server, even if necessary. Likely to cause a fatal error if using a proxy server is necessary.

Permitted Values: **auto**, **true** or **false**

Default: **auto**

TV::mrnet_super_bushy

When **true**, TotalView creates a "super bushy" MRNet tree by launching one MRNet **tvdsvr** process per target MPI process, instead of the default in which it launches one **tvdsvr** process per node.

This option addresses the CUDA debug API limitation that allows a debugger process (such as the **tvdsvr**) to debug at most one target process using a GPU. Set this option to **true** if you are debugging an MPI job in which more than one CUDA process is running on a node.

Permitted Values: **true** or **false**

Default: **false**

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**.

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. A value of **true** identifies use of NPTL threads, while **false** means that the program is not using this package.

Permitted Values: **true**, **false**, or **auto**

Default: **auto**

TV::open_cli_window_callback

Contains the string that the CLI executes after you open the CLI by selecting the **Tools > Command Line** command. It is ignored when you open the CLI from the command line.

This variable 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::openmp_debug_enabled

When **true**, TotalView enables runtime support for OpenMP OMPD.

This variable can also be set in a TotalView **.tvdrc** file within your **.totalview** directory.

Permitted Values: **true** or **false**

Default: **false**

TV::openmp_ompd_filter_stack

OpenMP-specific variable to control whether TotalView filters the stack. This option allows a different setting for OpenMP programs as compared to other programs. When **auto**, this variable is set to the same value as TV::stack_trace_transform_enabled. If either **true** or **false**, its value is not affected by **TV::stack_trace_transform_enabled**.

Set this option to **true** to always enable stack filtering in an OpenMP program. A value of **false** disables stack filtering for OpenMP programs. Neither value impacts the value of **TV::stack_trace_transform_enabled**.

This variable can also be set in a TotalView **.tvdrc** file within your **.totalview** directory.

Permitted Values: **auto**, **true**, or **false**

Default: **auto**

TV::parallel

When **true**, enables 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

Automatically attaches to processes. Your choices are:

- **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_stop

Tells TotalView if it should automatically run processes when your program launches them. Your choices are:

- **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 on which you are running TotalView. This is a read-only variable.

Permitted Values: A string indicating a platform, such as **sun5**

Default: Platform-specific

TV::process_load_callbacks

Names the procedures that TotalView runs after it loads or attaches to a program and just before it runs the program. TotalView executes these procedures after it invokes the procedures in the **TV::image_load_callbacks** list.

The procedures in this list are called at most once per process load or attach, even though your executable may use many shared libraries. After attaching to the processes in a parallel job, the callback procedures listed in **TV::process_load_callbacks** are invoked on one representative process in each share group, and only when the share group is first created. If the parallel job is restarted, the callback procedures are not invoked because the share groups are not recreated. All processes in a parallel job are attached before calling the procedures. The calls to the procedures are queued and executed at a later time, and are not guaranteed to be during the lifetime of the processes.

Permitted Values: A list of Tcl 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::proxy_server_server_launch_-string

Defines the launch string used when launching a proxy server.

Permitted Values: A string

Default: **%B/tvdsvr%K -callback %L -set_pw %P -verbosity %V %F %X**

TV::recurse_subroutines:

Determines whether a data window displaying the subroutines associated with a source file initially displays just the subroutine names, or also the data values in the subroutine scopes. This situation most commonly occurs in the **Program Browser**, available in the Classic UI.

- **true**: Displays both the subroutine names and the data in their scope.
- **false**: Displays only the subroutine names.

For complex applications, determining the state of the data values in the scope of all subroutines can significantly slow down TotalView. If set to **false** so only the subroutine names appear, data values for a particular subroutine can still be viewed by explicitly diving into the subroutine.

Permitted Values: **true** or **false**

Default: **true**

TV::replay_history_mode

Controls how ReplayEngine handles the history buffer when it is full, as follows:

- **1**: Discards the oldest history and continue.
- 2: Stops 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 killing a multi-threaded or multiprocess program, specifies the number of threads or processes that must be running before a prompt launches confirming that you wish to kill the program. By default, this prompt appears if there is more than one thread or process running.

Permitted Values: A positive integer

Default: 1

TV::reverse_connect_wanted

Controls whether TotalView listens for reverse connection requests.

Permitted Values: **true** or **false**

Default: **true**

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 **true**, text searches are case-sensitive, succeeding only for an exact match for the entry in the **Edit > Find** dialog box. For example, searching **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 **true**, TotalView uses its single-process server launch procedure when launching remote **tvdsvr** processes. When **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 debugging a remote process. This command string is executed by **/bin/sh**. By default, TotalView uses the command **ssh -x** 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**.

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 to wait for a response from the TotalView Debugger Server (**tvdsvr**) that it has launched.

*Permitted Values:*An integer from 1 to 3600 (1 hour)

Default: 30

TV::server_response_wait_timeout

Specifies how long to 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 **true**, newly created action points are shared across the group. If **false**, a newly created action point is active only 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 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

A list that 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_expand_inlined_subroutines

Controls the behavior of reading delayed symbols while building a stack backtrace in order to find inlined subroutines. By default, this variable is set to **auto**, so that TotalView attempts to automatically detect whether the subroutine associated with a stack frame might contain inlined subroutines; if so, it reads the delayed symbols for the file containing the subroutine.

If you are sure your subroutines contain no inlined subroutines or you are experiencing debugging performance issues during stack backtraces, you can set this value to **false**. However, doing so might result in stack backtraces that are missing inlined subroutines.

A setting of **true** means that TotalView will *always attempt* to read delayed symbols for inlined subroutines, but this could result in poorer debugger performance when building a stack backtrace. However, doing so will ensure that stack backtraces always contain all inlined subroutines.

Permitted Values: **true**, **false**, or **auto**

Default: **auto**

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:

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 **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: **suf-fix: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-x86-64, the list is "**linux-x86_64**." 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 **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. 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. 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 **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 **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 **true**, TotalViewwill 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::xplat_remcmd

A command that needs to be executed before executing a process on a remote host, e.g., **runauth**. This string is passed to MRNet instead of using the **XPLAT_REMCMD** environment variable. This value is only used when TotalView uses MRNet.

Permitted Values: A command

Default: {}

TV::xplat_rsh

An rsh command that is passed to MRNet instead of using the **XPLAT_RSH** environment variable. This command is used to launch remote processes. If this variable isn't explicitly set and the **XPLAT_RSH** environment variable is empty, TotalView uses the value of **TV::launch_command**. This value is used only when Classic TotalView uses MRNet.

Permitted Values: A remote launch command

Default: {}

TV::xplat_rsh_args

A list of arguments that need to be given to the remote launch command. This string is passed to MRNet instead of using the **XPLAT_RSH_ARGS** environment variable. This value is only used when TotalView uses MRNet.

Permitted Values: A space-separated list of remote launch arguments

Default: {}

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 **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 **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_detection_alignment

Specifies the alignment and stride TotalView uses as it steps through memory looking for pointers during leak detection. If 0 (the default value), then TotalView defaults to using the size of a pointer, which varies according to platform and programming model. In normal circumstances you should not need to adjust the alignment.

Permitted Values: A non-negative integer number

Default: **0**

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

NOTE: The variables in this section have meaning (and in some cases, a value) only 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 **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 **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 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_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_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::hand_cursor_enabled

Specifies whether the cursor should change to a hand cursor when hovering over an element you can dive into in the source pane of the process window.

Permitted Values: **true** or **false**

Default: **true**

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 **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::old_root_window

When **true**, TotalView replaces the Root Window with the Root Window used in versions prior to Classic TotalView 8.15. You can override this value using the following command-line options:

- **-oldroot** sets this variable to **true**
- **-newroot**sets this variable to **false**

NOTE: Using the previous-version Root Window may affect performance of applications containing thousands of threads/processes.

Permitted Values: **true** or **false**

Default: **false**

TV::GUI::pop_at_breakpoint

When **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**, this check box is unselected by default.

Permitted Values: **true** or **false**

Default: true

TV::GUI::pop_on_error

When **true**, TotalView sets the **Open process window on error signa l**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::process_grid_wanted

When **true**, TotalView enables the Processes/Ranks Tab in the Process Window. Enabling this tab can significantly affect performance, particularly for large, massively parallel applications.

Permitted Values: **true** or **false**

Default: **false**

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

This value set defines toolbar display.

Permitted Values: **icons_above_text**,**icons_besides_text**,**icons**,or **text**

Default: **icons_above_text**

TV::GUI::tooltips_enabled

When **true**, variable tooltips are displayed 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

PART II Transformations

This part of the *TotalView Reference Guide* discusses formatting and transformations that display data in a clear and concise format to facilitate easier debugging sessions.

Creating Type Transformations

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 and includes information on 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.

The chapter also describes C++View (CV), a facility that allows you to format program data in a more useful or meaningful form than the concrete representation that you see in TotalView when you inspect data in a running program.

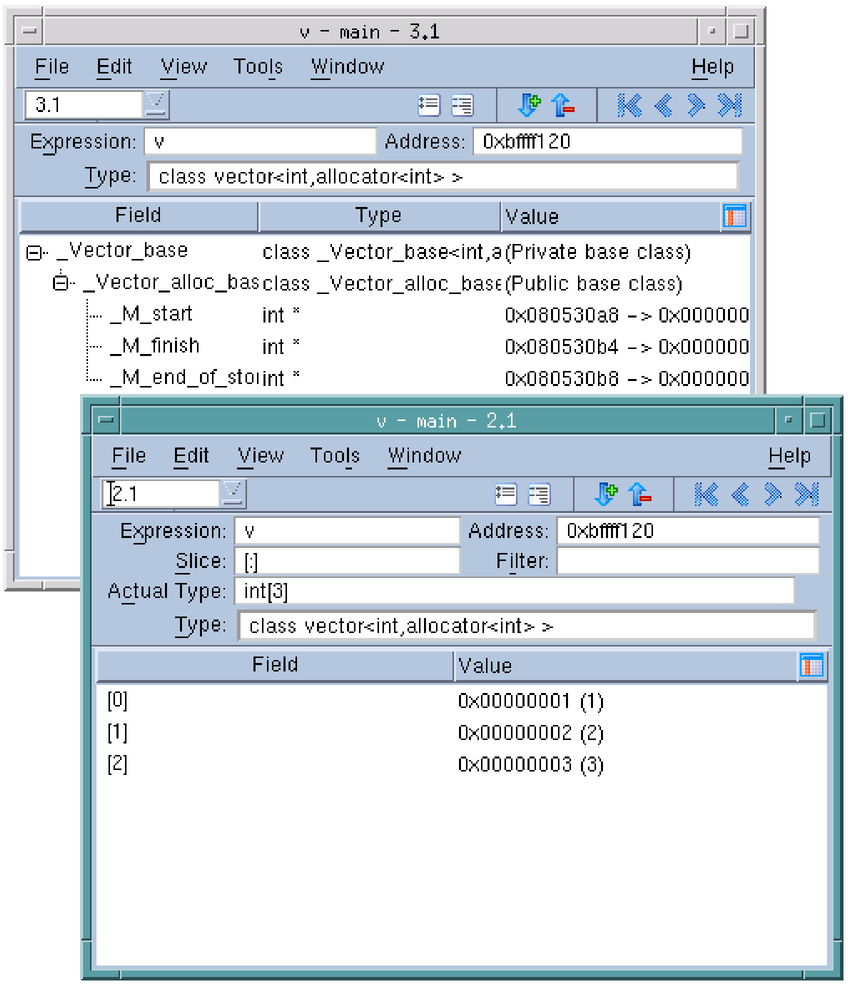
About the Type Transformation Facility

The Type Transformation Facility (TTF) customizes how 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.

Why Type Transformations

Modern programming languages allow you to use abstractions such as structures, class, and STL data types such as lists, maps, multimaps, sets, multisets, 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, Figure 6 shows a vector transformation.

Figure 6, A Vector Transformation



```
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_sto    int *                      0x080530b8 -> 0x000000
```

```
 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.

NOTE: By default, TotalView transforms STL strings, vectors, lists, maps, multimaps, sets, and multisets. The unordered STL types, unordered_map, unordered_multimap, unordered_set and unordered_multiset, are transformed for recent g++ compilers. If you do not want TotalView to transform your information, select the Options Tab within the File > Preferences Dialog Box and 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 } }
      }
   }
```

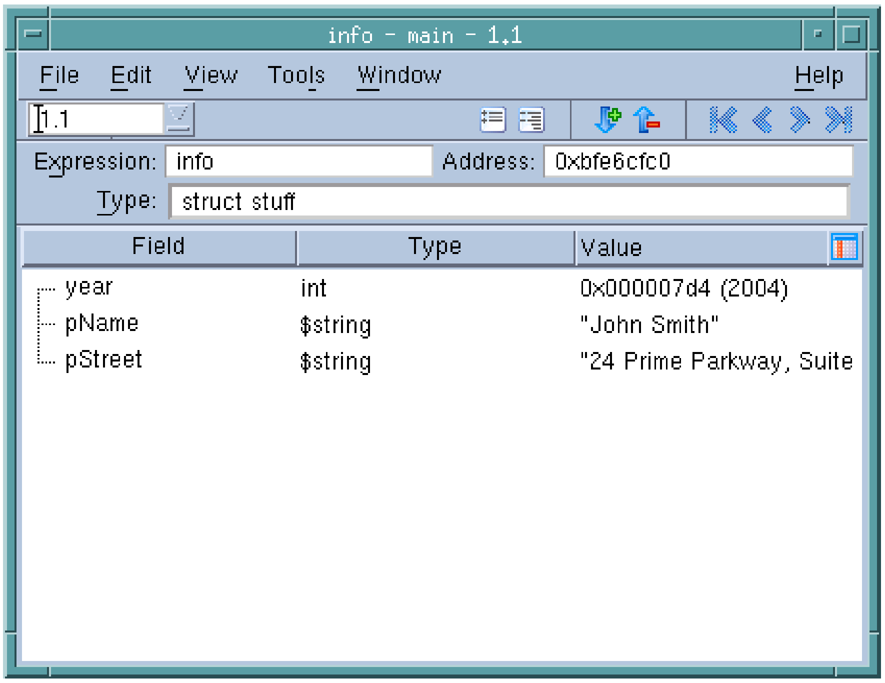
You can apply this transformation to your data in the following ways:

- 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 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 Variable Window when you dive on the **info** structure:

Figure 7, Transforming a 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

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 + offset}

Use the data whose location is an offset away from `expr`. This behaves just like pointer arithmetic in C and C++. The result is calculated based on the size of the type that `expr` points to:

```
result = expr + sizeof(*expr) * offset
```

{expr -> expr}

Names a subelement in a structure accessed using a pointer. This is used in the same way as the `->` operator 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

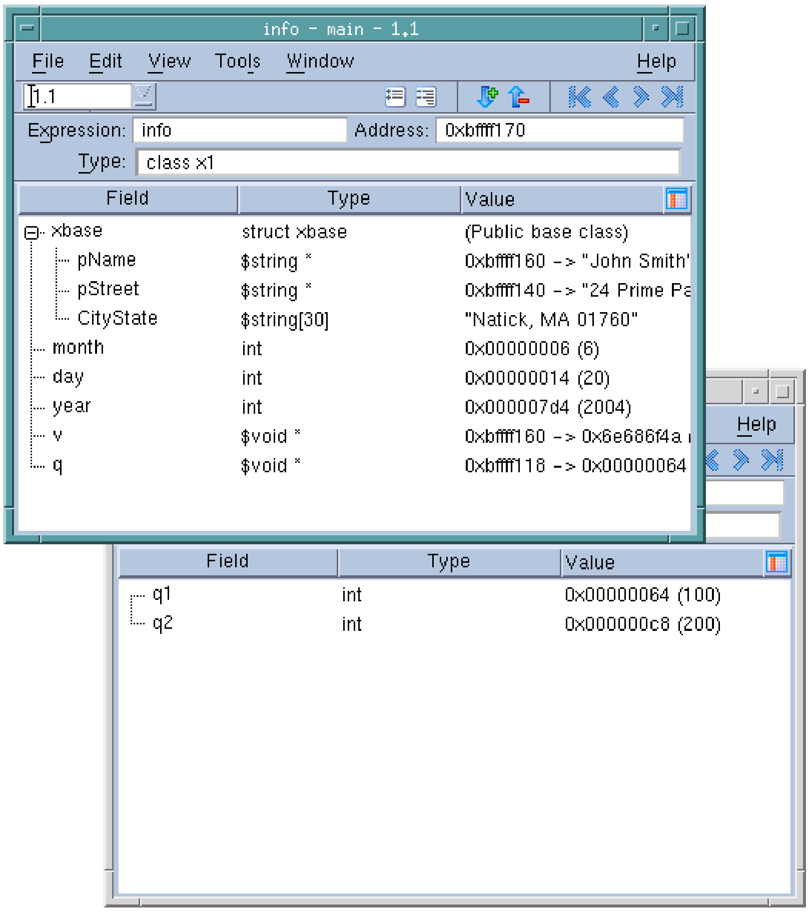
The example in this section changes the structure elements of the example in the previous section so that they are now class members. In addition, this example contains a class that is derived from a second class:

```
#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 8 shows the Variables Windows that TotalView displays for the **info** class and the **x** struct.

Figure 8, 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

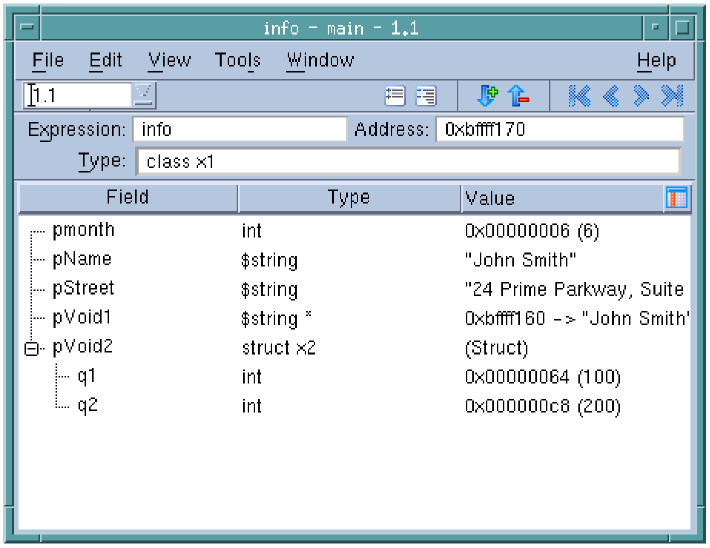
Field	Type	Value
q1	int	0x00000064 (100)
q2	int	0x000000c8 (200)

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.

Figure 9, 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
```

C++View

C++View (CV) is a facility that allows you to format program data in a more useful or meaningful form than the concrete representation that you see in TotalView when you inspect data in a running program. To use C++View, you must write a function for each type whose format you would like to control.

This section contains the following topics:

Writing a Data Display Function

The frame of reference in describing this is C++.

In order for C++View to work correctly, the code you write and TotalView must cooperate. There are two key issues here. The first is registering your function so that TotalView can find it when it needs to format data for display. This is straightforward: all you need to do is to define your function to have the right name and prototype. When TotalView needs to format the data of type T, it will look for a function with this signature:

```
 int TV_ttf_display_type ( const T * );
```

The **const** is deliberate to remind you that changes should not be made to the object being formatted for display. Many real-world applications are not entirely **const**-correct, and in cases where you must cast away the **const**, extreme caution is advised.

You will need to define a **TV_ttf_display_type** function for each type you want to format. A **TV_ttf_display_type** function may be at global scope, or it may be a class (static) method. It cannot be a member function.

The second issue concerns how the **TV_ttf_display_type** function which you will write communicates with TotalView. The API you will need to use is given in the header file **tv_data_display.h** included with your TotalView distribution in the ***<totalview-installation>*/src** directory.

Your **TV_ttf_display_type** will use the provided function **TV_ttf_add_row** to tell TotalView what information should be displayed. Its prototype is:

```
   int TV_ttf_add_row ( const char *field_name,
                        const char *type_name,
                        const char *address );
```

The **field_name** parameter is the descriptive name of the data field being computed. It will be shown by TotalView in a form similar to that of the name of a structure's field. The **type_name** parameter is the type of the data to be displayed. It must be the name of a legal type name in the program, or one of TotalView's types.

As a convenience, the header file provides these symbols for you:

TV_ttf_type_ascii_string

This tells TotalView to format a character array as a string (i.e., left to right) instead of an array (top to bottom).

TV_ttf_type_int

This is an alias for TotalView integer type **$int.**

The third parameter, **address**, is the address in your program's address space of the object to be displayed.

TV_ttf_add_row should be called only as a result of TotalView invoking your **TV_ttf_display_type** function. It may be called by a **TV_ttf_display_type** called by TotalView, or by one of the descendant callees of that **TV_ttf_display_type**.

Example

Here are the definitions of a couple of classes:

```
class A {
    int   i;
    char *s;
};

class B {
    A      a;
    double d;
};
```

We can define the display callback functions as follows:

```
int TV_ttf_display_type ( const A *a )
{
 /* NOTE: error checking of value returned from TV ttf add_row \
    omitted */
 (void) TV_ttf_add_row ( "i", TV_ttf_type_int, &(a->i) );
 (void) TV_ttf_add_row ( "s", TV_ttf_type_ascii_string, a->s );

 /* indicate success to TotalView */
 TV_ttf_format_ok;
}

int TV_ttf_display_type ( const B *b )
{
 /* NOTE: error checking of value returned from TV ttf add_row \
    omitted */
 (void) TV_ttf_add_row ( "a", "A", &(b->a) );
 (void) TV_ttf_add_row ( "d", "double", &(b->d) );

 /* indicate success to TotalView */
 return TV_ttf_format_ok;
}
```

For brevity and clarity, we have omitted all error checking of the value returned from **TV_ttf_add_row.** We will discuss the possible values that a **TV_ttf_display_type** may return later.

For now, we just return a simple success.

We could have made one or both of the display callbacks a class method:

```
class A {
 int    i;
 char  *s;
public:
 static int TV_ttf_display_type ( const A *a );
};

int A::TV_ttf_display_type ( const A *a )
{
 /* as before */
}
```

and similarly for class B.

Templates

C++View can also be used with template classes. Consider this container class:

```
template <class T> class BoundsCheckedArray {
private:
 int size;
 T   *array;

public:
 typedef T value_type;

 T ( int s ) { ... }
 ...
};
```

Writing a collection of overloaded display functions for each instantiated **BoundsCheckedArray** can rapidly become an overwhelming maintenance burden. Instead, consider whether you can write a template function.

One potential difficulty is getting the name of the type parameter to pass to **TV_ttf_add_row**. Here we follow the convention used by the container classes in the standard library which typedefs the template type parameter to the standard name **value_type**.

We can construct our template function like this:

```
template <class T>
int TV_ttf_display_type ( const BoundsCheckedArray<T> *a )
{
 char type [ 4096 ];

 snprintf ( type, sizeof ( type ), "value_type[%d]",        \
            a->get_size () );

 TV_ttf_add_row ( "array_values", type, a->get_array () );
 return TV_ttf_format_ok;
}
```

What we've done here is constructed the type of a fixed-sized array of the type named by the template type parameter. (In some cases you may need to use the compiler's demangler to get the name of the type. See also Tips and Tricks on page 379.)

This one definition can be used for any instance of the template class. In some cases, however, you may want a specialized implementation of the display function. As an illustration, consider this:

```
int TV_ttf_display_type ( const BoundsCheckedArray<char> *s )
{
 TV_ttf_add_row ( "string", TV_ttf_type_ascii_string,        \
                   s->get_array () );
   return TV_ttf_format_ok;
  }
```

Here we want to tell TotalView to display the array horizontally as a string instead of vertically as an array. For this reason, we want to pass **TV_ttf_type_ascii_string** to **TV_ttf_add_row** as the name of the type instead of the name constructed by the implementation of the general template display function. We therefore define a special version of the display function to handle **BoundsCheckedArray<char>**.

One remaining issue relating to templates is arranging for the various template display function instances to be instantiated. It is unlikely that display functions will be called directly by your program. (Indeed, we mentioned earlier that **TV_ttf_add_row** should not be called other than as a result of a call initiated by TotalView.) Consequently, the template functions may well not be generated automatically. You can either arrange for functions to be referenced, such as by calling them in a controlled manner, or by explicit template instantiation:

```
template int TV_ttf_display_type                        \
        ( const BoundsCheckedArray<int> * );
template int TV_ttf_display_type                        \
        ( const BoundsCheckedArray<double> * );
.
.
```

Precedence - Searching for TV_ttf_display_type

Only one call to a **TV_ttf_display_type** will be attempted per object to be displayed, even if multiple candidates are defined. For a type **T**, TotalView will look for the function in this order:

1. A class-qualified class (static) function returning **int** and taking a single **const T *** as its only argument.
2. A function at file scope, returning **int** and taking a single **const T *** as its only argument.
3. A global function, returning **int** and taking a single **const T *** as its only argument.
4. A TCL transformation

Namespace qualifications are not directly considered.

TV_ttf_add_row

TV_ttf_add_row will return one of the following values defined in the enum **TV_ttf_error_codes** given in the file **tv_data_display.h,** located in the ***<totalview-installation>*/include** directory in your distribution of TotalView.

The values returned by **TV_ttf_add_row** are:

TV_ttf_ec_ok

Indicates that the operation succeeded.

TV_ttf_ec_not_ active

Indicates that TV_ttf_add_row was called when the type formatting facility is not active. This is most likely to occur if **TV_ttf_add_row** is called other than as a result of a call to a **TV_ttf_display_type** initiated by TotalView.

TV_ttf_ec_invalid_characters

Indicates that either the field name or the type name contained illegal characters, such as **newline** or **tab**.

TV_ttf_ec_buffer_exhausted

Indicates that the internal buffer used by **TV_ttf_add_row** to marshal your formatted data for onward transmission to TotalView is full. See Tips and Tricks on page 379 for suggestions for reducing the number of calls to **TV_ttf_add_row**.

Return values from TV_ttf_display_type

The set of values your **TV_ttf_display_function** may return to TotalView is defined in the enum **TV_ttf_format_result** given in the file **tv_data_display.h** included with your distribution of TotalView. These values are:

TV_ttf_format_ok

Your function should return this value if it has successfully formatted the data and successfully registered its output using **TV_ttf_add_row**.

TV_ttf_format_ok_elide

As **TV_ttf_format_ok** but indicates that the output may be subject to type elision (see below).

TV_ttf_format_ failed

Return this if your function was unable to format the data. When displaying the data, TotalView will indicate that an error occurred.

TV_ttf_format_ raw

Use this to have your function tell TotalView to display the raw data as it would normally do, that is, as if there were no **TV_ttf_display_type** present for that type.

TV_ttf_format_ never

As **TV_ttf_format_raw.** In addition, this value tells TotalView never to call the display function again.

Elision

Elision is a feature that allows you to simplify how your data are presented. Consider the **BoundsCheckedAr-ray<char>** class and the specialized **TV_ttf_display_type** function we defined earlier:

```
int TV_ttf_display_type ( const BoundsCheckedArray<char> *s )
{
 (void) TV_ttf_add_row ( "string", TV_ttf_type_ascii_string,  \
                            s->get_array () );
 return TV_ttf_format_ok;
}
```

We used **TV_ttf_type_ascii_string** so that the array of characters is presented horizontally as a string, rather than vertically as an array. If our program declares a variable **BoundsCheckedArray<char> var1**, we will see output like this in the CLI:

```
d1.<> dprint var1
 var1 = {
  string = "Hello World!"
}
d1.<>
```

Note, however, that the variable **var1** is still presented as an aggregate or class. Conceptually this is unnecessary, and in this arrangement an extra dive may be necessary to examine the data. Additionally, more screen space is needed than is necessary.

You can use elision to promote the member of a class out one level. With elision, we will get output that looks like this:

```
d1.<> dprint var1
 string = "Hello World!"
d1.<>
```

TotalView will engage elision if your **TV_ttf_display_type** function returns **TV_ttf_format_ok_elide** (in place of **TV_ttf_format_ok**). In addition, for elision to occur, the object being presented must have only one field.

Other Constraints

An aggregate type cannot contain itself. (An attempt to do so would result in an infinite sized aggregate.) When generating a field of an aggregate **T** using **TV_ttf_add_row**, the named type may not be **T**, or anything which directly or indirectly contains a **T** as a member. If you do need to do something like that, use a pointer or reference.

As an illustration, consider this:

```
class A { ... };
class B { A a; ... };

int TV_ttf_display_type ( const A *a )
{
  (void) TV_ttf_add_row ( ... );
  return TV_ttf_format_ok;
}
```

```
int TV_ttf_display_type ( const B *b )
{
  (void) TV_ttf_add_row ( ... );
  (void) TV_ttf_add_row ( "a", "A", &(b->a) );
  return TV_ttf_format_ok;
}
```

Note the following:

- **TV_ttf_display_type (const A *a)** may not add an object of type **A** (direct inclusion) nor one of type **B** (indirect inclusion).
- When viewing an object of type **B**, TotalView will invoke **TV_ttf_add_row (const B *)**, and then **TV_ttf_add_row (const *A)**.

Safety

When you stop your program to inspect data, objects might not be in a fully consistent state. This may happen in a number of circumstances, such as:

- Stopping in a the middle of a constructor or destructor.
- Displaying an object in scope, but before its constructor has been called.
- Viewing a dangling pointer to an object, that is, a pointer to an object in memory that has been released by the program. This may be stack memory, but also heap memory. (If the target is running with memory debugging enabled, then TotalView does check that the object to be displayed does not lie in a deallocated region. If it does, then it does not call your **TV_ttf_display_type**, and will display the data in their raw form You should not, however, rely on this check.)

In the absence of C++View, this is not a problem, as displaying the data is just a matter of reading memory. However, with C++View, displaying data now involves executing functions in the target code. Your functions should be careful to check that the object to be displayed is in a consistent state. If you can't establish that with certainty, then it should not attempt to format the data, and instead it should return **TV_ttf_format_failed**.

Otherwise, your target program may crash when you attempt to display an object at an inappropriate time. As with any function call made from TotalView (expression list, evaluation window, etc.), TotalView recovers from this in a limited manner by posting an error message and restoring the stack to its original state. However, the target code may be left in an inconsistent or corrupted state, and further progress may not be possible or useful.

You may not place a breakpoint in a **TV_ttf_display_type** function. If you do, the callback will be aborted similarly, and TotalView will display an error.

Memory Management

You must make sure that the formatted data you want displayed by TotalView (the data whose address you supply as the third parameter to **TV_ttf_add_row)** remains allocated after the call to your **TV_ttf_display_type** returns. In practice this means that you shouldn't allocate these data on the stack. Your **TV_ttf_display_type** function may be called at anytime, including when your target program may be in the memory manager. For this reason it is inadvisable to allocate or deallocate dynamic memory in your **TV_ttf_display_type** functions. If the formatted data are manufactured, that is, generated by **TV_ttf_display_type** rather than already existing, then the memory for those data should be allocated during the target's normal course of execution.

You may find it convenient to have your program format data as part of its normal operations.That way there are no side-effects to worry about when TotalView calls your **TV_ttf_display_type** callback function.

The **field_name** and **type_name** string parameters to **TV_ttf_add_row** do not need to remain allocated after the call to TV_ttf_add_row.

Multithreading

Accessing shared data in multithreaded environments will usually need some sort of access control mechanism to protect its consistency and correctness. Your **TV_ttf_display_type** functions must be coded carefully if they need to access data that are usually protected by a lock or mutex. Attempting to take the lock or mutex may result in deadlock if the mutex is already locked.

Usually the threads in the program will have been stopped when TotalView calls the **TV_ttf_display_type** function. If the mutex is locked before TotalView calls **TV_ttf_display_type**, then an attempt by **TV_ttf_display_type** to lock the mutex will result in deadlock.

If you are designing a **TV_ttf_display_type** that needs to access data usually protected by a lock or mutex, consider whether you are able to determine whether the data are in a consistent state without having to take the lock. It might be enough to be able to determine whether the mutex is locked. If the data cannot be accessed safely, have the **TV_ttf_display_type** return **TV_ttf_format_failed** or **TV_ttf_format_raw** according to what fits best with your requirements.

Tips and Tricks

Consider constructing the type name on-the-fly. This can save time and memory. As an example, consider the **TV_ttf_display_type** for **BoundsCheckedArray<T>** we discussed earlier:

```
template <class T>
int TV_ttf_display_type ( const BoundsCheckedArray<T> *a )
{
  char type [ 4096 ];

  snprintf ( type, sizeof ( type ), "value_type[%d]", a->get_size () );
  (void) TV_ttf_add_row ( "array_values", type, a->get_array () );
  return TV_ttf_format_ok;
}
```

Note how we constructed an array type. The alternative would be to iterate **a->get_size ()** times calling **TV_ttf_add-d_row ()**. Depending on the number of elements, this could exhaust the API's buffer. In addition, there is a time penalty since TotalView will need to handle each line added by **TV_ttf_add_row** separately.

Constructing the array type as we did not only eliminates these disadvantages, it also provides other advantages. For example, as TotalView now knows that what is being presented is really an array, all the normal operations on arrays such as sorting, filtering, etc. are available.

Core Files

Because C++View needs to call a function in your program, C++View does not work with core files.

Using C++View with ReplayEngine

In general, C++View can be used with ReplayEngine just as with normal TotalView debugging. However, there are some differences you should be aware of. In both record mode and replay mode, TotalView switches your process into ReplayEngine's *volatile* mode before calling your `TV_ttf_display_type` function. When the call finishes, TotalView switches the process out of volatile mode. On entering volatile mode, ReplayEngine saves the state of the process, and on exiting volatile mode, ReplayEngine restores the saved status.

In most cases, executing `TV_ttf_display_type` in volatile mode behaves as you would expect. However, because ReplayEngine restores the earlier process state when it leaves volatile mode, any changes to process memory, such as writing to a variable, made while in volatile mode are lost.

This fact has implications for your program if your `TV_ttf_display_type` function modifies global or static data upon which either the function or the program relies. If `TV_ttf_display_type` does not change any global state, you will see no change in behavior when you engage ReplayEngine. However, if you generate synthetic values, such as the average, maximum or minimum values in an array, you cannot compute these in your `TV_ttf_display_type` function as the results will be lost when the function call terminates. Instead, consider generating them as a by-product of the program's normal execution as described in the section on Memory Management.

For more information on ReplayEngine, see The ReplayEngine User Guide.

The following code demonstrates how engaging ReplayEngine might affect calls to `TV_ttf_display_type`. This example is shipped with the ReplayEngine example files as `cppview_example_5.cc`.

```
/* Example program demonstrating TotalView's C++View with ReplayEngine. */
/* Run with (in both record and replay modes) and without ReplayEngine. */
/* Note how c in main is displayed in the various cases. */

#include <stdio.h>
#include "tv_data_display.h"

static int counter;

class C {
  public:
    int   value;

    C () : value ( 0 ) {};
}; /* C */

int
TV_ttf_display_type(const C *c)
{
  int  ret_val = TV_ttf_format_ok;
  int  err;

  // if Replay is engaged, this write to the global is lost because
  // the ttf function is evaluated in volatile mode
  counter++;

  // error checking omitted for brevity
  (void) TV_ttf_add_row ( "value", "int", &(c->value) );

  // show how many times we've been called.  Will always be zero
  // with Replay engaged because the update is lost when the
  // call to TV_ttf_display_type returns.
  (void) TV_ttf_add_row ( "number_of_times_called", "int", &counter );

  return ret_val ;
} /* TV_ttf_display_type */

int main(int argc, char *argv[])
{
  C c;

  c.value = 1;

  c.value++;      // should be 1  **before**  this line is executed

  c.value++;      // should be 2  **before**  this line is executed

  /* c.value should be 3 */

  return 0;
} /* main */
```

Compile and link the program with `tv_data_display.c` (see Compiling and linking tv_data_display.c). Follow this procedure:

1. Start the program under TotalView and enter the function `main`.
2. Dive on the local variable `c`, and note how the synthetic member `number_of_times_called` changes as you step through the program.

3. Restart, but this time with ReplayEngine engaged.
4. Notice the changes to the `value` member as you move forwards and backwards, and that the synthetic member `number_of_times_called` remains 0 because the increment in `TV_ttf_display_type` is lost when the function returns.

C

Although primarily intended for C++, C++View may be usable with C. C does not allow overloading so there may be at most one **TV_ttf_display_type** function with external linkage present. If you are interested in formatting only one type, then this restriction will not be constraining.

You may be able to work around this problem by defining separate **TV_ttf_display_type** functions as before, but placing each in a different file, and defining them to be static. Since the visibility of each definition is limited to the translation unit in which it appears, multiple functions can coexist.

This work-around, however, depends on the nature of the debug information emitted by the compiler. Some compilers do not place static functions in an indexable section in the debug information, or may try to optimize them out. If TotalView cannot find the function, it will not be called. TotalView cannot traverse the entire resolved symbol table to find these functions, as it would incur significant performance problems.

Compiling and linking tv_data_display.c

Your distribution includes the file **tv_data_display.c**. in the **<*totalview-installation*>/src** directory. This file contains the implementation of the interface between your **TV_ttf_display_type** functions and TotalView. This is distributed as source. You will need to compile this file and link it with your application.

You should take care to ensure that there is only one instance of **tv_data_display.c** present in your running application. One way in which multiple instances could creep in is if you link separate copies of the **tv_data_display.c** into independent shared libraries that your program uses. To avoid this type of problem, we strongly suggest that you build **tv_data_display.c** into its own separate shared library that can be shared by all the libraries your application uses. For example:

```
setenv TVSOURCE /usr/local/toolworks/totalview2020.01/src

setenv TVINCLUDE /usr/local/toolworks/totalview2020.01/include

gcc -g -Wall -fPIC -c $TVSOURCE/tv_data_display.c -I$TVINCLUDE  gcc -g \

-shared -Wl,-soname,libtv_data_display.so -o libtv_data_display.so tv_data_display.o
```

Some compilers or linkers will perform a type of garbage collection step and eliminate code or data that your application does not use. This affects C++View in two ways:

1. Your **TV_ttf_display_type** functions are unlikely to be called by your program.
2. Leading on from this, some of the entities in **tv_data_display.c** may not be reachable from your program.

As a result, the compiler or linker may identify your **TV_ttf_display_type** or **tv_data_display.c** as candidates for garbage collection and elimination. You can try to work around this problem by trying to create references to the **TV_ttf_display_type** functions.

Better still, we suggest identifying the flags for your compiler or linker that disable garbage collection. On AIX, for example, the linker flag **-bkeepfile:<filename>** tells the linker not to perform garbage collection in the file named **<filename>**.

C++View Example Files

Your TotalView distribution includes an examples directory, ***<totalview-installation>*/examples**, which includes the following C++View example files:

NOTE: Some compilers, such as some versions of gcc, do not emit debug information for typedefs in class scopes, and therefore TotalView cannot find the type underlying **value_type** so C++View may not work with those compilers.

cppview_example_1

A simple example showing two **TV_ttf_display_type** functions, one a function at global scope, the other a class function. It also demonstrates elision.

cppview_example_2

A simple example using templates, showing how the type named in the template can be passed to **TV_ttf_add_row**.

cppview_example_3

A more complex example using templates, showing how a **TV_ttf_display_type** function can be either generic or specialized for a particular instantiation of a template class. It also demonstrates elision.

cppview_example_4

A more complex example showing the use of STL container classes, elision, and the different values that **TV_ttf_display_type** can return.

cppview_example_5

This example adds a synthetic member to a class, and can be used to explore how C++View behaves under ReplayEngine.

Limitations

With the exception of Sun, compilers that emit STABS debug information do not handle C++ namespaces. This affects TotalView in general and C++View in particular, in that references to entities in namespaces are not always resolved.

Licensing

The C++View API library is distributed as two files. The first is **tv_data_display.c**, an ANSI C file that contains the implementation of the API used by your **TV_ttf_display_type** functions. The other is **tv_data_display.h**, which is a matching header file.

These files are licensed so as to permit unlimited embedding and redistribution.

PART III Running TotalView

This part of the *TotalView Reference Guide* contains information about command-line options used when starting TotalView and the TotalView Debugger Server.

- TotalView Command Syntax on page 386

  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.

- TotalView Debugger Server Command Syntax on page 400

  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

This chapter describes the syntax of the **totalview** command. Topics in this chapter are:

Command-Line Syntax

Format

totalview [*options* **]** **[** *executable* **[** *core-file* | *recording-file* **]] [-***a***[***args* **]]**

or

totalview [*options* **] -args** *executable* **[***args* **]**

Arguments

options

TotalView options.

executable

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 multi-process programs that call **fork()**, **vfork()**, or **execve()** should be linked with the **dbfork** library.

core-file

Specifies the name of a core file. Use this argument in addition to ***executable*** when you want to examine a core file with TotalView.

recording-file

Specifies the name of a saved replay recording session file. Use this argument in addition to ***executable*** when you want to replay the recording session with TotalView.

args

Default target program arguments.

Description

TotalView is a source-level debugger with features for debugging multiprocess programs and multithreaded programs, with multiple source files, executables, and shared libraries.

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.

-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***.

-bg *color*

Same as **-background**.

Default: **light blue**

-check_unique_id

(Default). TotalView attempts to extract a unique ID from an image file before checksumming it. For detail, see the state variable TV::check_unique_id.

-no_check_unique_id

TotalView does not try to extract a unique ID from an image file and instead relies on the setting for **-checksum_libraries**.

-checksum_libraries

(Default). TotalView checksums image files across nodes in a parallel debugging session. This setting is impacted by the setting for **-check_unique_id**. For detail, see the state variable TV::checksum_libraries.

-no_checksum_libraries

TotalView does not checksum image files across nodes in a parallel debugging session.

-classicUI

Launches the Classic TotalView UI rather than the modern UI

-control_c_quick_shutdown-ccq

(Default) 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.

-cuda

(Default) Enables CUDA debugging with TotalView.

-no_cuda

Disables CUDA debugging. Any CUDA kernels launched on a GPU device are not seen by the debugger, so the debugger can only debug the host code. **-nocuda** is the identical option.

-dbfork

(Default) Catches the **fork()**, **vfork()**, and **execve()** system calls if your executable is linked with the **dbfork** library.

-no_dbfork

Do not catch **fork()**, **vfork()**, and **execve()** system calls even if your executable is linked with the **dbfork** library.

-debug_file *console_outputfile*

Redirects TotalViewconsole output to a file named*console_outputfile*.

If *consoleoutputfile* is the string **UNIQUE**, the filename **tv_dump.***hostname.pid* is used. If *console_outputfile* contains the string **'$$'** (note the escaping single quotes), *hostname.pid* is substituted. **UNIQUE** and **'$$'** are useful for separating the console output when running multiple **tvdsvr** processes.

All TotalView console output is written to **stderr**.

-demangler= *compiler*

Overrides the demangler and mangler TotalView uses by default. The following indicate override options.

-demangler=gnu_dot:GNU C++ on Linux x86

-demangler=gnu_v3: GNU C++ Linux x86

-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*

Sets the name of the X Windows display to *displayname*. For example, **-display vinnie:0.0** displays 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 sets TotalView to 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 sets TotalView to ask if it should open a library that has any of the entries on this list as a suffix.

-dlopen_always_recalculate

(Default). Reevaluates breakpoint specifications on every **dlopen** call.

-no_dlopen_always_recalculate

Enables **dlopen** event filtering, deferring the evaluation of breakpoint specifications based on the value of the option **-dlopen_recalculate_on_match**).

This setting impacts scalability in HPC computing environments. For details, see Filtering dlopen Events on page 429.

-**dlopen_recalculate_on_match** *glob-list*

Default: "" (the empty string)

Contains a *glob-list* of patterns used to match against the path name of a *dlopened* library. If **-dlopen_always_recalculate** is set (the default), the value of this variable is ignored. When **-no_dlopen_always_recalculate** is set and a **dlopen** event occurs, TotalView matches the name of the *dlopened* library against the ***glob-list***. Be careful to quote the *glob-list* to prevent shell expansion. For example, if a *glob-list* contains special characters such as *****, **?, [,],** and **!**, be sure to quote the characters to prevent the shell from interpreting them. Note that some shells, like tcsh, will expand **!** even when it is enclosed in double or single quotes, which requires escaping the **!** with a backslash. For example, `'\!*/libfoo*'` will prevent a tcsh shell from expanding **!*** from history, and result in the *glob-list* being set to `'!*/libfoo*'`.

For a complete explanation of **dlopen** event filtering, including use-case examples, please refer to Filtering dlopen Events on page 429.

-dlopen_read_libraries_in_parallel

Enables **dlopen** events to be handled in parallel, reducing client/server communication overhead by using MRNet to fetch the library information.

-no_dlopen_read_libraries_in_parallel

(Default). Disables handling *dlopened* events in parallel.

This setting impacts scalability in HPC computing environments. For details, see Filtering dlopen Events on page 429.

-dump_core

Allows TotalView to dump a core file of itself when an internal error occurs. This is used to help Perforce Software debug problems.

-dwarf_global_index

(Default). Allows TotalView to use the DWARF global index sections (**.debug_pubnames**, **.debug_pubtypes**, **.debug_typenames**, etc.) in executable and shared library image files.

-no_dwarf_global_index

Forces TotalView to skim the DWARF instead of using them, which may cause TotalView to slow down when indexing symbol tables.

-e *commands*

Immediately executes 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

Uses only an Enterprise license.

-no_ent

Does not use an Enterprise license. You may combine this with **-no_team** or **--noteamplus.**

-env *variable=value*

Adds 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.

-exec_handling *exec-handling-list*

Default: "" (the empty string)

Controls how TotalView responds when a process being debugged calls **execve()**.

This option's argument, ***exec-handling-list***, is a Tcl list of ***regexp*** and ***action*** pairs. The ***regexp*** contains the name of the parent process, and ***action*** defines an action for TotalView to take.

regexp: A regular expression. The regular expression is not anchored, so use "**^**" and "**$**" to match the beginning or end of the process name.

action: The action to take, as follows:

Action	Description
halt	Stop the process
go	Continue the process
ask	Ask whether to stop the process

When a process that is being debugged execs a new executable, TotalView iterates over ***exec-handling-list*** to match the original process name (that is, the name of the process before the exec happened) against each ***regexp*** in the list. If it finds a match, it uses the corresponding ***action***.

If a matching process name is not found in the ***exec-handling-list***, the value of the TV::parallel_stop CLI state variable preference is used.

For more information, see "Controlling fork, vfork, and execve Handling" in the *TotalView User Guide*.

-fork_handling *fork-handling-list*

Default: "" (the empty string)

Controls how TotalView launches or attaches to new processes.

This option's argument, ***fork-handling-list***, is a Tcl list of ***regexp*** and ***action*** pairs. The ***regexp*** contains the name of the parent process, and ***action*** defines how future fork system calls will be handled for this process.

> ***regexp***: A regular expression. The regular expression is not anchored, so use "**^**" and "**$**" to match the beginning or end of the process name.
>
> ***action***: The action to take, as follows:

Action	Description
attach	Attach to the new child processes.
detach	Detach from the new child processes.

When first launching or attaching to a process, TotalView iterates over ***fork-handling-list*** to match the process name against each ***regexp*** in the list. When it finds a match, it uses the corresponding ***action*** to determine how future fork system calls will be handled

If a matching process name is not found in ***fork-handling-list***, TotalView handles **fork()** based on whether the process was linked with the **dbfork** library and the setting of the TV::dbforkCLI state variable preference.

For more information, see "Controlling fork, vfork, and execve Handling" in the *TotalView User Guide*.

-foreground *color*

Sets the general foreground color (that is, the text color) to ***color***.

-fg *color*

Same as **-foreground**.

Default: **black**

-gdb_index

(Default). Allows TotalView to use the **.gdb_index** section in executable and shared library image files.

-no_gdb_index

Forces TotalView to skim the DWARF instead.

-global_types

(Default) Sets TotalView to 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 attempts 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

For a program or library with either (or both) a build ID or **.gnu_debug_link** section, TotalView looks for a separate debug file. If found, TotalView reads this file's debugging information. The related state variable is TV::gnu_debuglink.

-no_gnu_debuglink

Do not load information from a separate debug file even if the file has a build ID or **.gnu_debug_link** section.

-gnu_debuglink_build_id_search_path

Sets the TV::gnu_debuglink_build_id_search_path variable to specify a build ID search path string.

-gnu_debuglink_check_build_id

For a program or library with either (or both) a build ID or **.gnu_debug_link** section, compare build IDs if the base image file contains a build ID.

Works in concert with **-gnu_debuglink_checksum** to define how to validate a separate debug info file, if one exists, against a base image file that references it. See the variables **TV::gnu_debuglink_check_build_id** and **TV::gnu_debuglink_checksum** for detail on how these options' settings impact one another.

-no_gnu_debuglink_check_build_id

Do not check build IDs. Whether a separate debug info file is validated or not also depends on the setting for **-gnu_debuglink_checksum**.

-gnu_debuglink_checksum

Validates the debug file's checksum against the checksum contained in the image's **.gnu_debuglink** section.

Works in concert with **-gnu_debuglink_check_build_id** to define how to validate a separate debug info file, if one exists, against a base image file that references it. See the variables **TV::gnu_debuglink_check_build_id** and **TV::gnu_debuglink_checksum** for detail on how these options' settings impact one another.

-no_gnu_debuglink_checksum

Do not compare checksums. Whether a separate debug info file is validated or not also depends on the setting for **-gnu_debuglink_check_build_id**. Set this only if you are absolutely certain that the debug file matches.

-gnu_debuglink_global_directory

Sets the TV::gnu_debuglink_global_directory variable that names the global directory that stores debug files.

-gnu_debuglink_search_path

Sets the TV::gnu_debuglink_search_path variable to specify a search path string.

-ipv6_support

Directs TotalView to support IPv6 addresses.

-no_ipv6_support

(Default) Do not support IPv6 addresses.

-jit_debugging

Enables Clang / LLVM JIT debugging. See TV::jit_debugging for details.

-no_jit_debugging

Disables Clang / LLVM JIT debugging.

-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 theTV::kcc_classes variable for a description of the conversions that TotalView performs.

-no_kcc_classes

Does 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.v4breakpoints** file, providing the file exists.

-nlb

Does not automatically load action points from an action points file.

-load_session *session_name*

Loads into TotalView the session named in ***session_name***. Session names with spaces must be enclosed in quotes, for example, "my debug session". Sessions that attach to an existing process cannot be loaded using this option; rather, use the **-pid** option instead.

-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.)

-mrnet_super_bushy

Sets the state variable **TV::mrnet_super_bushy** to **true**. When set, TotalView creates a "super bushy" MRNet tree. For detail, see the state variable TV::mrnet_super_bushy.

-no_mrnet_super_bushy

Sets the state variable to **false**.

-nodes

Specifies the number of nodes upon which the MPI job will run.

-no_startup_scripts

Sets TotalView to not reference any initialization files during startup. Note that this negates *all* settings in *all* initialization files. Aliases are `-nostartupscripts` and `-nss`.

-nohand_cursor

By default, the cursor in the source pane of the process window turns into a hand cursor when hovering over an element you can dive on (a red box is also drawn around the applicable code). Specify this option to override this behavior and retain the usual arrow cursor.

-np

Specifies how many tasks that TotalView should launch for the job. This argument usually follows a **-mpi** command-line option.

-nptl_threads

Sets your application to use NPTL threads. You need use this option only if TotalView cannot determine that you are using this threads package.

-no_nptl_threads

Does not use the NPTL threads package. Use this option if TotalView thinks your application is using it and it isn't.

-openmp_debug

Enables or disables OpenMP runtime support by setting the variable **TV::openmp_debug_enabled**.

-patch_area_base *address*

Allocates the patch space dynamically at *address*. See "*Allocating Patch Space for Compiled Expressions*" in the *Classic TotalView User 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 *Classic TotalView User Guide*.

-pid *pid filename*

Attaches to process *pid* for executable *filename* when TotalView starts executing.

-procs

Specifies how many tasks that TotalView should launch for the job. This argument usually follows a **-mpi** command-line option.

-replay

Enables the ReplayEngine when TotalView begins. This command-line option is ignored if you do not have a license for ReplayEngine. You may also use the alias `-reverse_debugging`.

-reverse_connect

Enables listening for reverse connections when TotalView launches. This is the default.

-no_reverse_connect

Disables listening for reverse connections when TotalView launches.

See the related state variable TV::reverse_connect_wanted.

-rocm

(Default) Enables AMD ROCm debugging with TotalView.

-no_rocm

Disables AMD ROCm debugging. Any AMD kernels launched on a GPU device are not seen by the debugger, so the debugger can only debug the host code. **-norocm** is the identical option.

-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 the *Classic TotalView User Guide*.

-search_path *pathlist*

Specifies a colon-separated list of directories in which 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*

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) Specifying an action for a signal in an ***action_list*** changes the default action for that signal.

(2) Not specifying a signal in the ***action_list*** does not change its default action for the signal.

(3) Specifying a signal that does not exist for the platform results in TotalView ignoring it.

(4) Specifying an action for a signal more than once results in TotalView using the last action specified.

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*"

Passes ***arguments*** to the starter program. You can omit the quotation marks if ***arguments*** is just one string without any embedded spaces.

-stack_trace_expand_inlined_subroutines *option*

Controls the behavior of reading delayed symbols while building a stack backtrace in order to find inlined subroutines. Possible options are **auto**, **true**, or **false**. The default is **auto**, meaning that TotalView attempts to automatically detect whether the subroutine associated with a stack frame might contain inlined subroutines; if so, it reads the delayed symbols for the file containing the subroutine.

For more information, see the state variable TV::stack_trace_expand_inlined_subroutines.

-stderr *pathname*

Names the file to which TotalView writes the target program's **stderr** information while executing within TotalView. If the file exists, TotalView overwrites it. If the file does not exist, TotalView creates it.

-stderr_append

Appends the target program's **stderr** information to the file named in the **-stderr** command, specified in the GUI, or in the TotalView **TV::default_stderr_filename** variable. If the file does not exist, TotalView creates it.

-stderr_is_stdout

Redirects the target program's **stderr** to **stdout**.

-stdin *pathname*

Names the file from which the target program reads information while executing within TotalView.

-stdout *pathname*

Names the file to which TotalView writes the target program's **stdout** information while executing within TotalView. If the file exists, TotalView overwrites it. If the file does not exist, TotalView creates it.

-stdout_append

Appends the target program's **stdout** information to the file named in the **-stdout** command, specified in the GUI, or in the TotalView **TV::default_stdout_filename** variable. If the file does not exist, TotalView creates it.

-team

Uses only a Team license.

-no_team

Does not use an Enterprise license. You may combine this with **-no_ent** or **-noteamplus.**

-teamplus

Uses only a Team Plus license.

-no_teamplus

Does not use a Team PLus license. You may combine this with **-no_ent** or **-noteam.**

-theme *option*

Controls the UI theme. Options are **dark** or **light**.

Default: **light**

-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. 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 the TotalView documentation page 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. 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 the TotalView documentation page 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.

-verbosity *level*

Sets the verbosity level of TotalView messages to ***level***, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

-working_directory *pathname*

Sets the working directory for executing a target program, overwriting the default.

Default: The directory from which TotalView was invoked

-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 attempts to create an **xterm** window.

TotalView Debugger Server Command Syntax

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 start up processes on remote servers.

For more information on remote debugging, refer to *TotalView Remote Connections"* in the *TotalView User Guide*.

Topics in this chapter are:

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, use the **File > Debug a Program** menu item and 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 on remote debugging, refer to *TotalView Remote Connections"* in the *TotalView User Guide*.

-**debug_file** *console_outputfile*

Redirects TotalView Debugger Server console output to a file named ***console_outputfile***.

If ***console_outputfile*** is the string **UNIQUE**, the filename **tv_dump.*hostname*.*pid*** is used. If ***console_outputfile*** contains the string **'$$'** (note the escaping single quotes), ***hostname*.*pid*** is substituted. **UNIQUE** and **'$$'** are useful for separating the console output when running multiple **tvdsvr** processes.

Default: All console output is written to **stderr**.

-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

-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"*.

-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 on remote debugging, refer to *TotalView Remote Connections"* in the *TotalView User 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:

%A

Expands to the ALPS Application ID (`apid`), which is a unique identifier for an application started using ALPS `aprun` on Cray XT, XE, and XK. The token is used to construct server path references copied onto the compute nodes' ramdisk under the `/var/spool/alps/apid` directory by the ALPS Tool Helper library.

%B

Expands to the bin directory where **tvdsvr** is installed.

%C

Is replaced by the value of the server launch command variable, TV::launch_command. On most platforms, this is **ssh -x**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView uses this 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

Expands to the `tvdsvr` platform suffix string in situations where a different server must be used.

When MRNet is being used as the debugger infrastructure, `_mrnet` is appended to the normal `%K` expansion. On Cray XT with MRNet enabled, the **%K** token is expanded to **_mrnet**. This convention allows MRNet-specific debugger servers to be launched only when MRNet is being used as the debugger infrastructure.

%L

If TotalView is launching one process, this is replaced by the host name and TCP/IP port number (*host-name*:*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 on remote debugging, refer to *TotalView Remote Connections"* in the *TotalView User 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.

%X

Is replaced with the appropriate proxy server option, which is currently valid only for MRNet proxy servers. When launching an MRNet proxy server, **%X** will expand to **-mrnet_proxy_fd %U**. The **-mrnet_proxy_fd** option informs the server that it should run as a proxy server.

The **%U** is replaced with a Unix domain socket file descriptor by the remote server that is launching the proxy server. The MRNet proxy server uses the socket to read the MRNet tree instantiation parameters and write the results. In other contexts, **%X** is replaced with **-invalid_proxy**, which will cause a server launch failure.

%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 IV Platforms and Operating Systems

This part describes information that is unique to the computers, operating systems, and environments in which TotalView runs.

Platforms and Compilers

This chapter describes the compilers and parallel runtime environments used on platforms supported by TotalView. See 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 Operating Systems on page 421.

Topics in this chapter are:

- Compiling with Debugging Symbols
- Maintaining Debug Information Separate from an Executable
- Linking with the dbfork Library
- Compiling and Linking Split DWARF

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.

NOTE: 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 macOS

On macOS, in all cases use the standard compiler invocation, just being sure to include the **-g** option.

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	**gfortran -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 macOS, you can create 64-bit applications using GCC 4 by adding the **-m64** command-line option.

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

NOTE: 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
```

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**
GCC Fortran	**gfortran -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 ARM64 Platform

The following table lists the procedures to compile programs on ARM64 platforms.

Compiler	Compiler Command Line
GCC C	**gcc -g** ***program*****.c**
GCC C++	**g++ -g** ***program*****.cxx**
GCC Fortran	**gfortran -g** ***program*****.f**

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**

Maintaining Debug Information Separate from an Executable

Because debug information embedded in an executable can be very large, some versions of Linux support stripping this information from the executable and placing it into a separate file. This file is then referenced within the executable using either a build ID section or a debug link section (or both) to identify the location and name of the separate debug file. 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 way this works with TotalView is controlled by a series of state variables and command line options discussed in Controlling Separate Debug Files.

Create this file on Linux systems that have an **objcopy** that supports the **- -add-gnu-debuglink** and **- -only-keep-debug** command-line options. If **objcopy - -help** mentions these options, creating this file is supported. See **man objcopy** for more details.

To create a separate file containing debug information:

1. Create a **.debug** copy of the executable or shared library. This second file is a regular executable but will contain only debugging symbol table information, with no code or data.
2. Create a stripped copy of the image file, and add to the stripped executable a **.gnu_debuglink** section that identifies the **.debug** file.

NOTE: The technique for creating a build ID separate debug information file is different and more complex than that for creating a debug link. Consult your system documentation for how to create a separate debug information file using the build ID method.

3. Distribute the stripped image and **.debug** files separately.

For example:

```
objcopy --only-keep-debug hello hello.debug
objcopy --strip-all hello
objcopy --add-gnu-debuglink=hello.debug hello
```

The code above uses `objcopy` to:

1. Create a separate debug file for an executable **hello** named **hello.debug**, containing only debug symbols and information.

2. Strip the debug information from the **hello** executable.
3. Add a **.gnu_debuglink** section to the **hello** executable.

Controlling Separate Debug Files

The following command line options and CLI variables control how TotalView handles separate debug files.

- **Controls whether TotalView looks for either a build ID or a .gnu_debuglink section in image files**:
  - Command line options -gnu_debuglink and **-no_gnu_debuglink**
  - State variable TV::gnu_debuglink

    This option basically turns on or off the functionality to support separate debug files.
- **Sets the search path to use when looking for debug files referenced by a .gnu_debuglink section**:
  - Command line option -gnu_debuglink_search_path
  - State variable TV::gnu_debuglink_search_path
- **Sets the search path to use when looking for debug files referenced by a .note.gnu.build-id section**:
  - Command line option -gnu_debuglink_build_id_search_path
  - State variable TV::gnu_debuglink_build_id_search_path
- **Specifies the global debug directory**:
  - Command line option -gnu_debuglink_global_directory
  - State variable TV::gnu_debuglink_global_directory
- **Validates the separate .gnu_debuglink debug file**:
  - ***Validate using a build ID, if available***

    Command line option**-gnu_debuglink_check_build_id**

    State variableTV::gnu_debuglink_check_build_id
  - ***Validate using a checksum***

    Command line option **-gnu_debuglink_checksum**

    State variable TV::gnu_debuglink_checksum

Searching for the Debug Files

If the**TV::gnu_debuglink** variable is **true** and if an image file contains either a **.note.gnu.build-id** or a **.gnu_debug_link** section, TotalView searches for a separate debug information file that matches the image file. TotalView will first search for the debug file using the **.note.gnu.build-id** section in the image file, if it exists. If that search fails, TotalView will search for the debug file using the **.gnu_debuglink** section in the image file, if it exists.

For the build ID method:

1. The TV::gnu_debuglink_build_id search_path string is split at the colon (:) characters into a list of strings.
2. For each string on the list, "%D", "%G", and "%/" token expansion is performed to yield a list of directory names to search.
3. The list of directories is searched for the debug file path named by the **.note.gnu.build-id** section. The debug file path follows the pattern "**.build-id**/*xx*/*yyy...yyy***.debug**", where *xx* are the first two hex characters of the build ID bit string, and *yyy...yyy* is the rest of the bit string. Build ID bit strings are at least 32 hex characters.

For separate debug files referenced by a **.gnu_debuglink** section:

1. The TV::gnu_debuglink_search_path string is split at the colon (:) characters into a list of strings.
2. For each string on the list, "%D", "%G", and "%/" token expansion is performed to yield a list of directory names to search.
3. The list of directories is searched for the debug file named in the **.gnu_debuglink** section. If the file is found, and the checksum matches or TV::gnu_debuglink_checksum is **false**, then the debug file is used.

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** and the **TV::gnu_debuglink_search_path** is set to its default value, 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**

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.

NOTE: 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.

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 -oprogram 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/rs6000/lib/libdbfork.a**
- **-L/usr/totalview/lib -ldbfork**

Linux or macOS

Add one of the following arguments or command-line options to the command that you use to link your programs:

- **-L/usr/totalview/***platform***/lib -L/usr/totalview/***platform***/lib -ldbfork_64**

where *platform* is one of the following: **darwin-x86**, **linux-x86-64**, **linux-powerle**, or **linux-arm64**.

For example:

```
cc -o program program.c \
   -L/usr/totalview/linux-x86/lib -ldbfork_64
```

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
```

Compiling and Linking Split DWARF

The Split DWARF approach is to split the DWARF into two parts during compilation: One part remains in the object (.o) file and the other is written to a corresponding DWARF object (.dwo) file. Lightweight "skeleton" DWARF debug information is included in the .o file, and full DWARF debug information is in the .dwo file. Since the linker processes only the .o files the size of the object files processed is greatly reduced.

Using GNU DebugFission Split DWARF on Linux

On Linux, TotalView supports the GNU DebugFission variant of Split DWARF. The DWARF 5 variant of Split DWARF is not yet supported. For more information, check out the Saving Time and Space with Split DWARF white paper.

Building your application for GNU DebugFission Split DWARF:

- **Compiling:**
  - Use the **-gsplit-dwarf** compiler option to generate ".dwo" (DWARF object) files containing the full DWARF debug information, and an ".o" (object) file containing the code, data, and skeleton DWARF debug information. Note: the DWARF debug information in the ".o" file points to the ".dwo" file, therefore the ".dwo" file must not be deleted.
- **Linking:**
  - Use the **-fuse-ld=gold** compiler option to use the gold linker (**ld.gold**).
  - Use the **-Wl, --gdb-index** compiler option to pass the **--gdb-index** option to the gold linker. Note: The resulting executable or shared library image file contains a **.gdb_index** section that the debugger can use for faster startup.

Using Split DWARF on Solaris

Starting with the Solaris Studio 12.4 compilers, Oracle® supports a Split-DWARF variant (also known as excluded DWARF) that can greatly reduce link time, disk usage, and debugger start-up time for large applications. When enabled, the full DWARF information remains in the object (.o) file and is excluded from the executable or shared library file. The compiler generates a lightweight symbol table index in the executable or shared library file. TotalView uses the symbol table index information to build a skeleton symbol table to locate the source files, functions, types, and other externally defined objects. The full DWARF information contained in the object file is read on-demand, only when required during the debug session.

To enable Split DWARF in the Solaris Studio 12.4 (or later) compilers, use the following compiler options:

```
-xdebugformat=dwarf -xs=no
```

For more information on how Oracle implements Split DWARF, please refer to the Oracle documentation.

Operating Systems

This chapter describes the operating system features that can be used with TotalView, including the following topics:

- Supported Operating Systems
- Troubleshooting macOS Installations
- Mounting the /proc File System (SunOS 5 only)
- Swap Space
- Shared Libraries
- Debugging Your Program’s Dynamically Loaded Libraries
- Remapping Keys (Sun Keyboards only)

Supported Operating Systems

Because support for operating systems, platforms, and compilers changes often, we do not identify support for these in the main product documentation. For all support questions, please refer to the document TotalView Supported Platforms in the TotalView distribution, or on the TotalView documentation website.

Troubleshooting macOS Installations

NOTE: For TotalView installation prerequisites on a Mac, see macOS Installations in the *Installation Guide*.

At TotalView startup, the OS checks whether the Mach system call `-task_for_pid()` is working properly. If the call returns an error, no debugging is possible, and TotalView outputs an error message that begins "The Mach system call -task_for_pid() is not working properly." Because this error is varied and depends on the OS version, TotalView cannot distinguish the circumstances that lead to it; however, the error is sometimes related to Apple's security layer, System Integrity Protection (SIP).

SIP's protections are not limited to protecting the system from file system changes. Some system calls are restricted in their functionality, which can affect developing and debugging on macOS. For runtime protection the following restrictions exist:

- `task_for_pid()` fails with EPERM if called incorrectly, which may cause TotalView to crash
- **dyld** environment variables are ignore
- DTrace probes are unavailable

However, SIP does not block inspection by developers of their own applications while they are being developed. TotalView tools will continue to allow applications to be inspected and debugged during the development process.

For more information about SIP, please see Apple's developer documentation.

Mounting the /proc File System

(Classic UI Only)

To debug programs on SunOS 5 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
SunOS 5	`% /sbin/mount | grep /proc  /proc on / proc read/write/setuid on ...`

If you receive one of these messages from the **mount** command, the **/proc** file system is mounted.

Mounting /proc with 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
SunOS 5	**/etc/vfstab**	**/proc - /proc proc - no -**

Then, to mount the **/proc** file system, enter the following command:

```
 /sbin/mount /proc
```

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 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.

1. 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 547  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 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.

1. 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**, Linux: **/lib/ld-linux.so.?**).
- 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.

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 TotalView Command Syntax on page 386 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.

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.

Debugging Your Program's Dynamically Loaded Libraries

dlopen Options for Scalability

When a target process calls `dlopen()`, a **dlopen** event is generated and must be handled by TotalView. Because **dlopen** event handling can affect debugger performance for a variety of reasons, especially if the application loads many shared libraries or the debugger is controlling many processes, TotalView provides ways to configure **dlopen** for better performance and scalability in HPC computing environments:

- Filtering **dlopen** events to avoid stopping a process for each event
- Handling **dlopen** events in parallel to reduce client/server communication overhead with MRNet enabled

Filtering dlopen Events

Two state variables and their related command line options enable you to filter **dlopen** events to defer planting breakpoints in the *dlopened* libraries until the process stops for some other reason. Deferring **dlopen** event processing allows the debugger to handle all dynamically loaded shared libraries at the same time, which is much more efficient than handling them serially.

dlopen event filtering is controlled by the settings on two state variables, TV::dlopen_always_recalculate and TV::dlopen_recalculate_on_match, and their related command line options **-dlopen_always_recalculate** and **-dlopen_recalculate_on_match**.

Three possible **dlopen** filtering modes are available using these variables: *Slow*, *Medium*, and *Fast*, in which *Fast* provides the best performance, although it won't be suitable for some debugging situations. For detail, see Filtering Modes.

You can configure TotalView to filter **dlopen** events for all invocations of TotalView using your **.tvdrc** file. For example, to use *Fast* mode by default for every TotalView session, add the following to your **.tvdrc** file:

```
dset TV::dlopen_always_recalculate false
dset TV::dlopen_recalculate_on_match ""
```

Or, launch just an individual instance of TotalView with these settings by entering:

```
totalview -no_dlopen_always_recalculate -dlopen_recalculate_on_match ""
```

Filtering Modes

Filtering modes for **dlopen** include *Fast*, *Medium*, and *Slow*. In *Fast* mode, the process never stops for a **dlopen** event, not even "null" **dlopen** events. Using this option can result in significant performance gains, but may be impractical for some applications. In *Medium* mode, some libraries can be specified to either immediately reevaluate or defer evaluation of breakpoint specifications, rather than all or none. In *Slow* mode, every **dlopen** event results in the immediate reevaluation of breakpoint specifications.

- **Slow Mode: Reloads libraries on every dlopen event**

  Option:

  ```
  dset TV::dlopen_always_recalculate true
  ```

  Reloads libraries on every **dlopen** event, retaining TotalView's traditional breakpoint reevaluation semantics. This mode is compatible with CUDA and is a good choice when your session has pending breakpoints. However, this mode does not perform or scale as well as the other modes, because it requires the TotalView client to handle every (non-null) **dlopen** event for every process.

  If performance is not the primary concern, or the application or runtime environment does not perform many **dlopen** events, then this may be a good choice.

  In this mode, when the target stops with a **dlopen** event, the debugger server reports the event to the debugger client, where the library list is reloaded and checked to see if any additional breakpoint locations need to be planted in the newly loaded libraries.

- **Medium Mode: Reports or defers libraries that match defined patterns on a dlopen event**

  Options:

  ```
  dset TV::dlopen_always_recalculate false
  dset TV::dlopen_recalculate_on_match {glob-list}
  ```

  A *glob-list* is a colon-separated list of positive or negated Tcl glob match patterns used to determine if the *dlopened* library event should be reported or deferred. For example:

  *Immediately report **dlopen** events for libraries that match any of the patterns on the glob-list, but defer reporting other **dlopen** events:*

  ```
  "*/libcuda.so*:*/libmylib1*:*/libmylib2.so"
  ```

  *Defer reporting **dlopen** events for libraries that match any of the patterns on the glob-list, but immediately report other **dlopen** events*:

  ```
  "!*/libboring.so*:!*/libwhocares1*:!*/libwhocares2.so"
  ```

  The glob match rules are defined by the standard Tcl **string match** command. For details and examples, see glob-list Matching Rules. Note that the library names are typically absolute path names, for example "/lib64/libc.so", so the glob patterns must take that into account.

This mode strikes a balance between performance and enabling breakpoints to be planted in *dlopened* libraries, and is useful if you have specific shared libraries that you know you always, or never, want to defer. For example, Open MPI performs many **dlopen** calls in parallel programs, however most users are not interested in planting breakpoints or debugging the Open MPI libraries themselves. Therefore, it makes sense to defer reporting **dlopen** events for Open MPI libraries.

In *Medium* mode, the target process stops on every **dlopen** event (just as in *Slow* mode), but:

- If a newly loaded library matches a positive *glob-list* entry, the event is immediately reported to the client, but all other libraries are deferred. For example:

  ```
  dset TV::dlopen_always_recalculate false
  dset TV::dlopen_recalculate_on_match {*/libfoo.so:*/libbar.so}
  ```

  Here, when /home/jones/libfoo.so or /home/jones/libbar.so are loaded, the **dlopen** event is immediately reported and breakpoints are reevaluated because their names match a pattern in the *glob-list*. However, when /usr/lib64/libompi.so is loaded, breakpoints are deferred because its name does not match a pattern in the *glob-list*.

- If a newly loaded library matches a negated *glob-list* pattern, and the list contains *only negated patterns* (i.e., does not contain a combination of negated and positive patterns), the event is deferred for that library, but all other libraries not matching a negated pattern are immediately reported.

- If a newly loaded library matches a negated *glob-list* pattern, and the list contains a *combination of positive and negated patterns*, the event *might* be deferred, depending on other library names in the library list. See glob-list Matching Rules for details.

This setting requires:

- Adding patterns that match the names of any *dlopened* libraries to the TV::dlopen_recalculate_on_match list

- Adding "***/libcuda.so***" to the match list if you are debugging CUDA; otherwise TotalView will miss CUDA kernel launch events.

- **Fast Mode**: Does not stop for **dlopen** events

  Options:

  ```
  dset TV::dlopen_always_recalculate false
  dset TV::dlopen_recalculate_on_match ""
  ```

  This mode provides the best performance, deferring planting breakpoints in all *dlopened* libraries when a library is loaded. Breakpoints (pending or not) are planted in the *dlopened* libraries only when the process stops for some other reason; however, be aware that with this option, an application may have executed past the point at which you want to start debugging inside the *dlopened* library.

  Because the debugger does not plant the **dlopen** breakpoint in the process, the process never stops for a **dlopen** event, not even "null" **dlopen** events.

While this mode may be impractical for some debugging situations, the performance gains are significant.

Table 7 summarizes the pros and cons of each mode.

Table 7: dlopen Event Filtering Modes

Mode/ Speed	Option	
Slow	`TV::dlopen_always_recalculate true`	
	Pros: ■ Retains TotalView's traditional breakpoint reevaluation semantics. ■ Works best with pending breakpoints. ■ Compatible with CUDA.	**Cons**: ■ Does not perform or scale as well as the other modes because the TotalView client handles every (non-null) **dlopen** event for every process.

Medium	`TV::dlopen_always_recalculate false`
	`TV::dlopen_recalculate_on_match {`*`glob-list`*`}`	

	Pros: ■ Performs better by filtering out **dlopen** events. ■ Allows the TotalView client to process multiple **dlopen** events at a time, which is much more efficient. ■ Compatible with CUDA.	**Cons**: ■ Process stops at the **dlopen** breakpoint, even for "null" **dlopen** events. ■ An application *may* execute past the point at which you want to start debugging inside the *dlopened* library. ■ Requires adding to ***glob-list*** any libraries that should or should not cause breakpoint specifications to be reevaluated immediately when the library is loaded. ■ Requires adding to the ***glob-list*** ***/libcuda.so*** for CUDA support.

Table 7: dlopen Event Filtering Modes

Mode/ Speed	Option	

Fast	`TV::dlopen_always_recalculate false`
	`TV::dlopen_recalculate_on_match ""`	

	Pros: ■ Performs best by never stopping the process at **dlopen** events. ■ Allows the TotalView client to process multiple **dlopen** events at a time.	**Cons**: ■ Breakpoints are not recalculated when a particular library is loaded, which breaks pending breakpoints and traditional breakpoint semantics. ■ Breaks CUDA support.

glob-list Matching Rules

The glob-list is a colon-separated list of positive or negated Tcl glob match patterns. A glob match is negated if it starts with an exclamation point (!), which is removed from the pattern before testing for a match. The glob match rules are defined by the standard Tcl **string match** command. For example:

- **Positive match pattern**: `/lib/libfoo*`
- **Negated match pattern**: `!/lib/libfoo*`

Note that:

- The order of positive or negated *glob-list* patterns matter, if you are mixing positive and negated patterns.
- Spaces are included in a match, so stray spaces will impact the result.
- Empty patterns (see below) are allowed, but will result in no match.
- A trailing, negated empty pattern is allowed, which affects only the default result.
- Library names are typically absolute path names (e.g., "/lib64/libc.so"), so the glob patterns must take that into account.
- Tcl **string match** glob rules are not the same as shell glob rules, in that a "*" will match across directory boundaries. For example, the glob pattern "*/libfoo.so" will match "/lib/libfoo.so" and "/usr/lib/libfoo.so".

Mixed positive and negated patterns

While combining positive and negated patterns is likely to be rare, in some cases it is useful, for example, to defer reporting **dlopen** events for all shared libraries in a directory *except* one.

A ***glob-list*** that contains a combination of positive and negated glob patterns can return varied results:

1. When a library name matches a positive match pattern, the **dlopen** event is reported immediately, even if there are more library names on the library list that would have resulted in a negated match.
2. When a library name matches a negated match pattern, the reporting of the **dlopen** event *might* be deferred. If there are more library names on the library list (because loading the library resulted in loading its dependent libraries), they are also checked for a positive match. If a positive match is found for any of the dependent libraries, the **dlopen** event is reported immediately.
3. In both cases, once a library name matches a pattern, any remaining patterns on the ***glob-list*** are *not* checked.

Empty match patterns

The ***glob-list*** is allowed to have empty match patterns, which may either be a positive, empty match pattern ("") or a negated, empty match pattern ("!").

An empty match pattern never matches a library name, but might affect the default result:

- A positive empty match pattern is ignored and does *not* affect the default result.
- A negated empty match pattern is ignored but *might* affect the default result.

Results when there is no match

If no match is found in the *glob-list* for any library name on the library list, the default result is determined as follows:

1. If the last, non-empty pattern on the pattern list is a positive match pattern, reporting the **dlopen** event is deferred.
2. If the last, non-empty pattern on the pattern list is a negated match pattern (empty or not), the **dlopen** event is reported.
3. If the pattern list consists solely of positive empty match patterns (e.g., ":::"), reporting the **dlopen** event is deferred.

Examples

Defer all libraries except those in a specific directory

A ***glob-list*** can contain a path to a directory containing shared libraries:

```
dset TV::dlopen_always_recalculate false
dset TV::dlopen_recalculate_on_match {/home/jones/project/lib*}
```

In this case, TotalView calculates breakpoint specifications on all shared libraries except those in the `/home/jones/project/lib/` directory.

Negated and positive patterns

You can control the results, based on the combination of negated and positive patterns, the order of the patterns, or the use of negated empty match patterns.

Consider the following ***glob-list*** containing both a negated and a positive pattern:

```
dset TV::dlopen_always_recalculate false
dset TV::dlopen_recalculate_on_match {*/libopen-rte.so*:!/*/mware/*}
```

For these *dlopened* libraries:

- `/opt/mware/openmpi/lib/libopen-rte.so.4`

  matches the first positive glob pattern "`*/libopen-rte.so*`", so the **dlopen** event is reported immediately.

- `/opt/mware/openmpi/lib/openmpi/mca_gizmo.so`

  matches the second negated glob pattern "`/*/mware/*`", so reporting the **dlopen** event is deferred.

- `/home/jones/project/lib/libmine.so`

  does not match either glob pattern, therefore a default result is returned. Since the last glob pattern on the list is a negated pattern, the **dlopen** event is reported.

Pattern order

If the *glob-list* contains both negated and positive patterns, the order in which the patterns appear matters and can result in unintended behavior. Consider what would happen if the patterns used in the previous example were swapped:

```
dset TV::dlopen_always_recalculate false
dset TV::dlopen_recalculate_on_match {!/*/mware/*:*/libopen-rte.so*}
```

For these *dlopened* libraries:

- `/opt/mware/openmpi/lib/libopen-rte.so.4`

  matches the first negated glob pattern "/*/mware/*", so the **dlopen** event is deferred. The second glob pattern "*/libopen-rte.so*" is not checked, because once a library name matches a pattern, any remaining patterns on the *glob-list* are not checked.

- `/opt/mware/openmpi/lib/openmpi/mca_gizmo.so`

  matches the first negated glob pattern "/*/mware/*", so the **dlopen** event is deferred.

- `/home/jones/project/lib/libmine.so`

  does not match either glob pattern, therefore a default result is returned. Since the last glob pattern on the list is a positive pattern, the **dlopen** event is deferred.

Simply swapping the patterns resulted in deferring every **dlopen** event, which is probably not the intention.

Handling dlopen Events in Parallel

TotalView's default behavior is to handle *dlopened* libraries serially, creating multiple, single-cast client-server communications. This can degrade performance, depending on the number of libraries a process dlopens, and the number of processes in the job.

To handle *dlopened* libraries in parallel, use the **TV::dlopen_read_libraries_in_parallel** and its related command line option **-dlopen_read_libraries_in_parallel**.

This sets the state variable to **true**. Placing this **dset** command in the **tvdrc** file ensures that all instances of TotalView launch with this option:

```
dset TV::dlopen_read_libraries_in_parallel true
```

To set this option on an individual instance of TotalView, use the command line option when you start TotalView:

```
totalview -dlopen_read_libraries_in_parallel
```

NOTE: Enabling this option does not guarantee that **dlopen** performance will improve on all systems in all scenarios. Be sure to test the impact of this setting on your system and debugging environments.

Remember that MRNet must also be enabled for this to work.

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 **key-sym** 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'
```

Architectures

This chapter describes the architectures TotalView supports, including:

AMD and Intel x86-64

This section describes AMD's 64-bit processors and the Intel EM64T processors, including:

- x86-64 General Registers on page 439
- x86-64 Floating-Point Registers on page 440
- x86-64 FPCR Register on page 441
- x86-64 FPSR Register on page 442
- x86-64 MXCSR Register on page 443

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

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**

Register	Description	Data Type	Edit	Dive	Specify in Expression
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	**$code[]**	no	yes	**$rip**
FS_BASE		**$long**	yes	yes	**$fs_base**
GS_BASE		**$long**	yes	yes	**$gs_base**
TEMP		**$long**	no	no	**$temp**

x86-64 Floating-Point Registers

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**
FPOP	Floating-point operation	**$int**	no	no	**$fpop**

Register	Description	Data Type	Edit	Dive	Specify in Expression
FPI	Instruction address	**$int**	no	no	**$fpi**
FPD	Data address	**$int**	no	no	**$fpd**
MXCSR	SSE status and control	**$int**	yes	no	**$mxcsr**
MXCS-R_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**

NOTE: 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
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

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

Value	Bit Setting	Meaning
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

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

NOTE: The Power architecture supports the IEEE floating-point format.

Power General Registers

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	**$int/$long**	yes	yes	**$r0**
SP	Stack pointer	**$int/$long**	yes	yes	**$sp**
RTOC	TOC pointer	**$int/$long**	yes	yes	**$rtoc**
R3 - R31	General registers 3 - 31	**$int/$long**	yes	yes	**$r3 - $r31**
INUM		**$int/$long**	yes	no	**$inum**
PC	Program counter	**$code[]**	no	yes	**$pc**
SRR1	Machine status save/ restore register	**$int/$long**	yes	no	**$srr1**
LR	Link register	**$code[]**	yes	no	**$lr**
CTR	Counter register	**$int/$long**	yes	no	**$ctr**
CR	Condition register (see below)	**$int/$long**	yes	no	**$cr**

Register	Description	Data Type	Edit	Dive	Specify in Expression
XER	Integer exception register (see below)	**$int/$long**	yes	no	**$xer**
DAR	Data address register	**$int/$long**	yes	no	**$dar**
MQ	MQ register	**$int/$long**	yes	no	**$mq**
MSR	Machine state register	**$int/$long**	yes	no	**$msr**
SEG0 - SEG9	Segment registers 0 - 9	**$int/$long**	yes	no	**$seg0 - $seg9**
SG10 - SG15	Segment registers 10 - 15	**$int/$long**	yes	no	**$sg10 - $sg15**
SCNT	SS_COUNT	**$int/$long**	yes	no	**$scnt**
SAD1	SS_ADDR 1	**$int/$long**	yes	no	**$sad1**
SAD2	SS_ADDR 2	**$int/$long**	yes	no	**$sad2**
SCD1	SS_CODE 1	**$int/$long**	yes	no	**$scd1**
SCD2	SS_CODE 2	**$int/$long**	yes	no	**$scd2**
TID		**$int/$long**	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

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: ¥ - ¥, or infinity-infinity
0x00400000	**VXIDI**	Floating-point invalid operation exception: ¥ / ¥, or infinity divided by infinity
0x00200000	**VXZDZ**	Floating-point invalid operation exception: 0 / 0
0x00100000	**VXIMZ**	Floating-point invalid operation exception: ¥ * ¥, or infinity times infinity

Value	Bit Setting	Meaning
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
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

Value	Bit Setting	Meaning
0x00000002	**RN=PINF**	Round toward +infinity
0x00000003	**RN=NINF**	Round toward **-**infinity

ARM64

This section contains the following information:

NOTE: The ARM64 processor architecture supports the IEEE floating-point format.

ARM64 General Registers

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
X0-X30	General registers 0 - 30	**$long**	yes	yes	**$x0 -$x30**
SP	Stack pointer	**$long**	no	yes	**$sp**
PC	Program counter	**$long**	no	yes	**$pc**
PSTATE	Process state	**$long**	no	yes	**$pstate**
VFP	Virtual frame pointer	**$long**	no	yes	**$vfp**

ARM64 Floating-Point Registers

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 - F30	Floating-point registers 0 - 30	**$double**	yes	yes	**$f0 -$f30**

Register	Description	Data Type	Edit	Dive	Specify in Expression
FPSR	Floating-point status register	**$int**	yes	yes	**$fpsr**
FPCR	Floating-point control register	**$int**	yes	yes	**$pc**

ARM64 FPCR Register

For your convenience, TotalView interprets the bit settings of the ARM64 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
0x100	**IOE**	Invalid operation exception enable
0x200	**DZE**	Division by zero exception enable
0x400	**OFE**	Overflow exception enable
0x800	**UFE**	Underflow exception enable
0x1000	**IXE**	Inexact exception enable
0x8000	**IDE**	Input denormal exception enable
0x0 (bits 23 and 24 clear)	**RMode=RN**	Round to nearest
0x400000	**RMode=RP**	Round towards plus infinity
0x800000	**RMode=RM**	Round towards minus infinity
0xC00000	**RMode=RZ**	Round towards zero
0x1000000	**RMode=(per above)+FZ**	Flush-to-zero
0x2000000	**RMode=(per above)+DN**	Operations on NaN return default NaN
0x1000000	**RMode=(per above)+AHP**	Alternative half-precision

ARM64 FPSR Register

For your convenience, TotalView interprets the bit settings of the ARM64 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
0x1	**IOC**	Invalid operation exception occurred
0x2	**DZC**	Division by zero exception occurred
0x4	**OFC**	Overflow exception occurred
0x8	**UFC**	Underflow exception occurred
0x10	**IXC**	Inexact exception occurred
0x80	**IDC**	Input denormal exception occurred
0x8000000	**QC**	Saturation occurred
0x10000000	**V**	Overflow condition code
0x20000000	**C**	Carry condition code
0x40000000	**Z**	Zero condition code
0x80000000	**N**	Negative condition code

Intel x86

This section contains the following information:

NOTE: The Intel x86 processor supports the IEEE floating-point format.

Intel x86 General Registers

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**

Register	Description	Data Type	Edit	Dive	Specify in Expression
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

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**

Register	Description	Data Type	Edit	Dive	Specify in Expression
FPDOFF	Data offset	**$int**	no	no	**$fpdoff**
FPDSEL	Data selector	**$int**	no	no	**$fpdsel**
MXCSR	SSE status and control	**$int**	yes	no	$mxcsv
MXCS-R_MASK	MXCSR mask	**$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**

NOTE: 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

Value	Bit Setting	Meaning
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

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
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

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

NOTE: The SPARC processor supports the IEEE floating-point format.

SPARC General Registers

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**

Register	Description	Data Type	Edit	Dive	Specify in Expression
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

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**

Register	Description	Data Type	Edit	Dive	Specify in Expression
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

Value	Bit Setting	Meaning
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 “*Handling Signals*” in the *Classic TotalView User 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.

Appendix A

TotalView Glossary

This glossary defines terms specific to TotalView.

action point

A breakpoint. TotalView action points include standard breakpoints, watchpoints, eval points, and barriers.

action point identifier

A unique integer ID associated with an action point.

affected p/t set

The set of process and threads that are affected by the command. For most commands, this is identical to the target P/T set, but in some cases it might include additional threads. (See **p/t (process/thread) set** for more information.)

aggregated output

The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

arena

A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

array slice

A subsection of an array, which is expressed in terms of an upper bound, a lower bound, and a **stride**. Displaying a slice of an array can be useful when you are working with very large arrays.

autolaunching

When a process begins executing on a remote computer, TotalView can also launch a **tvdsvr** (TotalView Debugger Server) process on the computer that will send debugging information back to the TotalView process that you are interacting with.

automatic process acquisition

TotalView detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you don't have to attach to them manually. If the process is on a remote computer, automatic process acquisition starts the -TotalView Debugger Server (**tvdsvr**).

barrier point

An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

command history list

A debugger-maintained list that stores copies of the most recent commands issued by the user.

conditional breakpoint

A breakpoint containing an expression. If the expression evaluates to true, program stops. TotalView does not have conditional breakpoints. Instead, you must explicitly tell TotalView to end execution by using the $stop directive.

control group

All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program has two distinct executables that run independently of one another. Each would be in a separate control group. In contrast, processes created by the **fork()** function are in the same control group.

debugger server

See **tvdsvr process**.

debugger state

Information that TotalView or the CLI maintains to interpret and respond to user commands. This includes debugger modes, user-defined commands, and debugger variables.

dpid

Debugger ID. The ID used for processes.

eval point

A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

expression system

A part of TotalView that evaluates C, C++, and Fortran expressions. An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of a source language. Not all Fortran 90, C, and C++ operators are supported.

focus

The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you're using the default prompt).

gid

The TotalView group ID.

GOI

The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

group

When TotalView starts processes, it places related processes in families. These families are called "groups."

group of interest

The primary group that is affected by a command. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

HIA

The Heap Interposition Agent, used when memory debugging. The HIA intercepts calls to heap library functions that allocate and deallocate memory by using the malloc() and free() functions and related functions such as calloc() and realloc(). In most cases, the HIA is loaded automatically when your program starts. For some platforms, however, the HIA needs to be explicitly linked to your application. See Linking Your Application with the Agent in the *TotalView User Guide.*

host computer

The computer on which TotalView is running.

initial process

The process created as part of a load operation, or that already existed in the runtime environment and was attached by TotalView or the CLI.

initialization file

An optional file that establishes initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called **.tvdrc**.

lockstep group

All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group. By definition, all members of a lockstep group are in the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

manager thread

A thread created by the operating system. In most cases, you do not want to manage or examine manager threads.

native debugging

The action of debugging a program that is running on the same machine as TotalView.

pid

Depending on the context, this is either the process ID or the program ID. In most cases, this is the process ID.

POI

The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and so on.

process group

A group of processes associated with a multi-process program. A process group includes program control groups and share groups.

process/thread identifier

A unique integer ID associated with a particular process and thread.

process of interest

The primary process that TotalView uses when it is trying to determine what to step, stop, and so on.

program control group

A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent), and children that were forked with a subsequent call to the **execve()** function (processes that don't share the same source code as the parent). Contrast this with **share group**.

program event

A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

p/t (process/thread) set

The set of threads drawn from all threads in all processes of the target program.

pthread ID

The ID assigned by the Posix pthreads package. If this differs from the system TID, it is a pointer value that points to the pthread ID.

satisfaction set

The set of processes and threads that must be held before a barrier can be satisfied.

satisfied

A condition that indicates that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are either running because they have not yet hit the barrier, or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is **satisfied**, the held processes or threads are released, which means they can be run. Prior to this event, they could not run.

serial line debugging

A form of remote debugging where TotalView and the **tvdsvr** communicate over a serial line.

service thread

A thread whose purpose is to *service* or manage other threads. For example, queue managers and print spoolers are service threads. There are two kinds of service threads: those created by the operating system or runtime system and those created by your program.

share group

All the processes in a control group that share the same code. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.

single process server launch

A TotalView procedure that individually launches **tvdsvr** processes.

slice

A subsection of an array, which is expressed in terms of a lower bound, upper bound, and **stride**. Displaying a slice of an array can be useful when you are working with very large arrays.

stop set

A set of threads that TotalView stops after an action point -triggers.

stride

The interval between array elements in a slice and the order in which TotalView displays these elements. If the stride is 1, TotalView displays every element between the lower bound and upper bound of the slice. If the stride is 2, TotalView displays every other element. If the stride is -1, TotalView displays every element between the upper bound and lower bound (reverse order).

target computer

The computer on which the process to be debugged is running.

target process set

The target set for those occasions when operations can only be applied to entire processes, not to individual threads in a process.

target program

The executing program that is the target of debugger operations.

target p/t set

The set of processes and threads on which a CLI command acts.

thread of interest (TOI)

The primary thread affected by a command.

tid

The thread ID. On some systems (such as AIX where the threads have no obvious meaning), TotalView uses its own IDs.

trigger set

The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

triggers

The effect during execution when program operations cause an event to occur (such as arriving at a breakpoint).

tvdsvr process

The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

type transformation facility (TTF)

Abbreviated as TTF. A TotalView subsystem that allows you to change the way information appears. For example, an STL vector can appear as an array.

user thread

A thread created by your program.

watchpoint

An action point that stops execution when the value of a memory location changes.

worker thread

A thread in a workers group. These are threads created by your program that perform the task for which you've written the program.

workers group

All the worker threads in a **control group**. Worker threads can reside in more than one **share group**.

Index