NEXTGEN TOTALVIEW FOR HPC REFERENCE GUIDE

Version 2019.1
Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Rogue Wave Software, Inc., assumes no responsibility and will not be liable for any errors, omissions, damage, or loss that might result from any use or misuse of the Documentation.

ROGUE WAVE SOFTWARE, INC., MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ROGUE WAVE SOFTWARE, INC., HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DOCUMENTATION, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT SHALL ROGUE WAVE SOFTWARE, INC., BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT, PUNITIVE, OR EXEMPLARY DAMAGES IN CONNECTION WITH THE USE OF THE DOCUMENTATION.

The Documentation is subject to change at any time without notice.

Rogue Wave Software, Inc.
Address: 5500 Flatiron Parkway, Boulder, CO 80301 USA
Product Information: (303) 473-9118 (800) 487-3217
Fax: (303) 473-9137
Web: http://www.roguewave.com
Overview

This Reference Guide describes all of the commands and variables supported by the TotalView debugging engine. Everything described is supported in the Command Line Interface (CLI), which can be run in a command terminal, and is also available in the Command Line view of NextGen TotalView for HPC. However, many commands relate to functionality that does not appear in the NextGen TotalView for HPC interface. So while you can issue these commands and determine their effect through other CLI commands, you may not see any result reflected in the NextGen UI.

Note also that this release of NextGen TotalView for HPC supports a limited number of platforms. See the document Supported Platforms, either on the Rogue Wave web site or in the product distribution, for specific information.

The table that follows provides general guidance on the functionality that NextGen TotalView for HPC supports.

<table>
<thead>
<tr>
<th>Supported by the New UI</th>
<th>Not Supported by the New UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most commands related to basic debugging actions for stepping through code</td>
<td>Parallel debugging environments such as MPI, OpenMP, and HPC systems</td>
</tr>
<tr>
<td>Attaching to a running process</td>
<td>Remote debugging</td>
</tr>
<tr>
<td>Debugging with core files</td>
<td>Memory debugging with MemoryScape</td>
</tr>
<tr>
<td>Breakpoints. Other action points such as watchpoints are not supported.</td>
<td>CUDA debugging</td>
</tr>
<tr>
<td>Many commands for examining and manipulating the data in variables</td>
<td>Xeon Phi debugging</td>
</tr>
</tbody>
</table>
### Supported by the New UI

<table>
<thead>
<tr>
<th>Supported by the New UI</th>
<th>Not Supported by the New UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>TotalView startup options unless they relate to unsupported functionality such as parallel and remote debugging</td>
<td>Custom thread and process groups reverse debugging with ReplayEngine, including Replay on Demand and the saving and loading of recording files</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
</tbody>
</table>

The information in this guide is organized in parts:

- **Part I, “Using the CLI,”** on page 1 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 340 discusses formatting and transformations that display data in a clear and concise format to facilitate easier debugging sessions.
- **Part III, “Running TotalView,”** on page 366 documents all possible command-line options as well as those that customize the behavior of the tvdsvr.

### Resources

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

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

The documentation for TotalView for HPC could be useful if you are using features not yet supported in the Next-Gen TotalView for HPC UI by invoking commands through the Command Line Interface (CLI). The commands themselves are described in this Reference Guide, but the TotalView for HPC 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 TotalView for HPC distribution, or on the Rogue Wave web site.
This part of the reference guide describes the TotalView Command Line Interface (CLI).

**Chapter 1, “CLI Command Summary,” on page 2**
Summarizes all CLI commands.

**Chapter 2, “CLI Commands,” on page 15**
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.

**Chapter 3, “CLI Namespace Commands,” on page 207**
Describes commands found in the TV:: namespace. These commands are seldom used interactively, as they are most often used in scripts.

**Chapter 4, “Batch Debugging Using tvscript,” on page 264**
Discusses how to create batch scripts that run TotalView unattended.

**Chapter 5, “TotalView Variables,” on page 281**
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.
Chapter 1

CLI Command Summary

This chapter contains a summary of all TotalView debugger CLI commands. The commands are described in detail in Chapter 2, “CLI Commands,” on page 15 and Chapter 3, “CLI Namespace Commands,” on page 207.

**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_subset subset_specification]

**dbarrier**
Creates a barrier breakpoint at a source location

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

Creates a barrier breakpoint at an address

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

**dbreak**
Creates a breakpoint at a source location

dbreak breakpoint-expr [-p | -g | -t] \([ -l lang] -e expr \)[ -pending]

Creates a breakpoint at an address

dbreak -address addr [-p | -g | -t] \([ -l lang] -e expr \)[ -pending]

**dcache**
Clears the remote library cache

dcache -flush

**dcalltree**
Displays parallel backtrace data

dcalltree [-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*

**dddetach**

Detaches from the processes

*dddetach*

**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

```bash
denable action-point-list
```

Enables all disabled action points in the current focus

```bash
denable -a
```

**dexamine**

Display memory contents

```bash
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

```bash
dflush
```

Removes all suspended *dprint* computations

```bash
dflush -all
```

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

```bash
dflush susp-eval-id
```

**dfocus**

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

```bash
dfocus p/t-set
```

Executes a command in this P/T set

```bash
dfocus [p/t-set command]
```

**dga**

Displays global array variables

```bash
dga [-lang lang_type] [handle_or_name] [slice]
```

**dgo**

Resumes execution of target processes

```bash
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
\texttt{dheap -hoard [ subcommands ]}

Displays Memory Debugger information
\texttt{dheap -info [ -backtrace ] [ start_address [ end_address ] ]}

Indicates whether an address is within a deallocated block
\texttt{dheap -is_dangling address}

Locates memory leaks
\texttt{dheap -leaks [ -check_interior ]}

Enables or disables Memory Debugger event notification
\texttt{dheap [-no]notify}

Paints memory with a distinct pattern
\texttt{dheap -paint [ subcommands ]}

Enables and disables the ability to catch bounds errors and use-after-free errors retaining freed memory blocks
\texttt{dheap -red_zones [ subcommands ]}

Enables and disables allocation and reallocation notification
\texttt{dheap -tag_alloc subcommand start_address [ end_address ]}

Displays the Memory Debugger's version number
\texttt{dheap -version}

\texttt{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.
\texttt{dhistory [ -info ] [ -get_time ] [ -go_time time ] [ -go_live ] [ -enable ] [ -disable ]}

\texttt{dhold}

Holds processes
\texttt{dhold -process}

Holds threads
\texttt{dhold -thread}

\texttt{dkill}

Terminates execution of target processes
\texttt{dkill [ -remove ]}

\texttt{dlappend}

Appends list elements to a TotalView variable
dlappend *variable-name value* [ ... ]

dlist
Displays code relative to the current list location

*dlist* [-n *num-lines*]

Displays code relative to a named location

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

Displays code relative to the current execution location

*dlist* -e [-n *num-lines*]

dll
Manages shared libraries

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

dload
Loads debugging information

*dload* [-g *gid*] [-mpi *starter_value*] [-r *hname*]

[-replay | -noreplay]

[-env *variable=value*] ... [-e] *executable*

[-parallel_attach_subset* subset_specification*]

dmstat
Displays memory use information

*dmstat*

dnext
Steps source lines, stepping over subroutines

*dnext* [-back] [ *num-steps*]

dnexti
Steps machine instructions, stepping over subroutines

*dnexti* [-back] [ *num-steps*]

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

*dout* [-back] [ *frame-count*]

dprint
Prints the value of a variable or expression

*dprint* [-nowait] [-slice *slice_expr*] [-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

```
```

**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 | { -rffunction | -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_process_startup process_id

spurs
  Manages threads using commands modeled after the GDB SPU Runtime System (SPU) library.
    spurs add [ directory directory-list ... ]
**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

Command Overview

This chapter lists all of CLI commands with a brief description.

**NOTE >>** This chapter describes some functionality that exists in the underlying debugging engine TotalView, but may not be supported in the NextGen TotalView for HPC user interface. To access these features, use the Command Line view. See “Overview” on page iii 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**: Create and manage 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.
• **spurs**: Manages threads using commands modeled after the GDB SPU Runtime System (SPU) library.

**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.
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 step22 {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 step22 {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 descriptor 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 console. 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

```tcl
set save_stat [ capture st ]
```
Saves the current process status to a Tcl variable.

```tcl
set arg [ capture p argc]
```
Saves the printed value of argc into a Tcl variable.

```tcl
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:

```tcl
int2_array(1) = -8 (0xffffB)
int2_array(2) = -6 (0xffffa)
int2_array(3) = -4 (0xffffc)
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

  drunCommand
drerunCommand
dactions

Displays information, and saves and reloads action points

Format

Displays information about action points.

```
```

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 the File > Preference Action Points Page information in the TotalView for HPC online Help.

**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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>dactions</td>
<td>Displays all action points</td>
</tr>
</tbody>
</table>

**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
```

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] Disabled
d1.==> dactions -full
```

**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)
```
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
Address 0: [Enabled] 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 -l 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 /* 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++)
```

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 [-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:
CLI Commands

```
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 *NextGen TotalView for HPC User Guide*
- Saving Action Points to a File in the *NextGen TotalView for HPC 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 NextGen TotalView for HPC 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, the TotalView expression system evaluates the expression. See “Using the Evaluate Window” in the TotalView for HPC User Guide for more information.
**Command alias**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>as</td>
<td>dassign</td>
<td>Changes a scalar variable's value</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{dassign scalar\_y 102} \]

Stores the value 102 in each occurrence of variable \texttt{scalar\_y} for all processes and threads in the current set.

\[ \text{dassign i 10*10} \]

Stores the value 100 in variable \texttt{i}.

\[ \text{dassign i i*i} \]

Does not work and the CLI displays an error message. If \texttt{i} is a simple scalar variable, you can use the following statements:

\[ \text{set x \{lindex \{capture dprint i\} 2\}} \]
\[ \text{dassign i \{expr $x * $x\}} \]

\[ \text{f \{p1 p2 p3\} as scalar\_y 102} \]

Stores the value 102 in each occurrence of variable \texttt{scalar\_y} contained in processes 1, 2, and 3.

**RELATED TOPICS**

- Changing the Value of Variables
- Changing a Variable’s Data Type
**dattach**

Brings currently executing processes under TotalView control

**Format**


**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 285 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 for HPC 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. For additional information, see the Parallel Page in the File > Preferences Dialog Box in the in-product help for TotalView for HPC.

- `-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 Page in the File > Preferences Dialog Box in the in-product help for TotalView for HPC.

  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 help for TotalView for HPC.

- `-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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td><code>dattach</code></td>
<td>Brings the process under TotalView control</td>
</tr>
</tbody>
</table>

**Examples**

`dattach mysys 10020`

Loads debugging information for `mysys` and brings the process known to the run-time system as PID 10020 under TotalView control.

`dattach -e 123 10020`

Loads file 123 and brings the process known to the run-time 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 run-time 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 Processes in the NextGen TotalView for HPC User Guide
- Examining Core Files in the NextGen TotalView for HPC 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_hitwidth][-stop_when_donewidth] [ -pending ]
```

Creates a barrier breakpoint at an absolute address

```
  dbARRIER -address addr [-stop_when_hitwidth ][ -stop_when_donewidth] [ -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 Chapter 9 of the TotalView for HPC User Guide.

For more information on breakpoint expressions, see `dbreak` on page 41, particularly Breakpoint Expressions.

`-address addr`

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

`-stop_when_hitwidth`

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 41.

**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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>dbarrier</td>
<td>Defines a barrier.</td>
</tr>
<tr>
<td>baw</td>
<td>{dfocus pW dbarrier -stop_when_done process}</td>
<td>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.</td>
</tr>
<tr>
<td>BAW</td>
<td>{dfocus gW dbarrier -stop_when_done group}</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

- Setting Breakpoints and Barriers in the *TotalView for HPC User Guide*
- Barrier Points in the *TotalView for HPC User Guide*
- Using Groups, Processes, and Threads in the *TotalView for HPC User Guide*
- Creating a Satisfaction Set in the *TotalView for HPC User Guide*
- Holding and Releasing Processes and Threads in the *TotalView for HPC User Guide*
- `dactionsCommand`
- `dbreakCommand`
- `denableCommand`
- `ddisableCommand`
**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 Chapter 9, “Qualifying Symbol Names” in the *TotalView for HPC 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 151 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 151 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 151 for more information.

`-l lang`

Sets the programming language used when you are entering expression `expr`. Enter either: c, c++, f77, f90, 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. The language statements and operators you can use are described in Chapter 8, “Setting Action Points” in the *TotalView for HPC User Guide*.

`-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 \texttt{breakpoint-expr} or the absolute address \texttt{addr}. You can enter a breakpoint expression that are sets of addresses at which the breakpoint is placed, and are as follows:

- \texttt{[[##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 \texttt{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.

- \texttt{class\ class\_name}
  
  Specifies that the breakpoint should be planted in all member functions of class \texttt{class\_name}.

- \texttt{virtual\ class::signature}
  
  Specifies that the breakpoint should be planted in all virtual member functions that match \texttt{signature} and are in the class or derived from the class.

### Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>break</td>
<td>Sets a breakpoint</td>
</tr>
<tr>
<td>bt</td>
<td>{dbreak t}</td>
<td>Sets a breakpoint only on the thread of interest</td>
</tr>
</tbody>
</table>

### Examples

For all examples, assume that the current process set is \texttt{d2.<} when the breakpoint is defined.

\texttt{dbreak 12}

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

\texttt{dbreak -address 0x1000764}

Suspends process 2 when execution reaches address 0x1000764.

\texttt{b 12 -g}
Suspends all processes in the current control group when execution reaches line 12.

dbreak 57 -1 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

Setting Breakpoints and Barriers in the TotalView for HPC User Guide
Defining Evaluation Points and Conditional Breakpoints in the TotalView for HPC User Guide
Using Groups, Processes, and Threads in the TotalView for HPC User Guide
dactionsCommand
dbreakCommand
denableCommand
ddisableCommand
**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


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.
- **-sortcolumn**
  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>
}
{
...
}
}
```
The \texttt{-show_details} and \texttt{-hide_backtrace} switches pull in opposite directions. The \texttt{-show_details} switch shows the maximum data, including all processes and threads. The \texttt{-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.

\textbf{Command alias}

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ct</td>
<td>\texttt{dcalltree}</td>
<td>Prints data to console</td>
</tr>
<tr>
<td>ctd</td>
<td>\texttt{dcalltree-data}</td>
<td>Puts data in a Tcl associative array</td>
</tr>
<tr>
<td>ctshb</td>
<td>\texttt{dcalltree-hide_backtrace}</td>
<td>Prints data only on root and leaf nodes</td>
</tr>
<tr>
<td>ctd</td>
<td>\texttt{dcalltree-show_details}</td>
<td>Prints more complete data</td>
</tr>
</tbody>
</table>

\textbf{Examples}

d\texttt{focus group dcalltree}

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

\begin{tabular}{ll}
Processes & Location \tabularnewline
--------- & -------- \\
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 0x2b342535f6e2 <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] ... \\
\end{tabular}
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 forker 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#705 0x00401c9b <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 

Adding the -hide_backtrace switch reduces the clutter somewhat:

Processes Location PC Host Rank ID Status
--------- -------- -- ---- ---- -- ------
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
CLI Commands

dcalltree

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 TotalView for HPC 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


dcheckpoint -by pe -halt

   Checkpoints the Parallel Environment job. All associated processes halt.

RELATED TOPICS

drestartCommand
**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:

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

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

**Command alias**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>co</td>
<td>dcont</td>
<td>Resume</td>
</tr>
<tr>
<td>CO</td>
<td>{dfocus g dcont}</td>
<td>Resume at group-level</td>
</tr>
</tbody>
</table>

**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 TotalView for HPC User Guide
dgoCommand
dwaitCommand
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,By),(Tx)>>>`
- `<<<(Bx,By,By),(Tx,Ty)>>>`
- `<<<(Bx,By,By),(Tx,By,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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuda</td>
<td>dcuda</td>
<td>Writes out the focus thread, as in <code>dcuda kernel</code>.</td>
</tr>
</tbody>
</table>

---

**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: 0x00000000f/0x00000000f/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: 0x00000000f/0x00000000f/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: 0x00000000f/0x00000000f/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 for HPC User Guide
**ddelete**

**Deletes action points**

**Format**

Deletes the specified action points

\[ \text{ddelete action-point-list} \]

Deletes all action points

\[ \text{ddelete -a} \]

**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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>de</td>
<td>ddelete</td>
<td>Deletes action points</td>
</tr>
</tbody>
</table>

**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**

```plaintext
ddetach
```

**Arguments**

This command has no arguments.

**Description**

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

You can detach any process controlled by the CLI; the process being detached 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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>det</td>
<td>ddetach</td>
<td>Detaches from processes</td>
</tr>
</tbody>
</table>

**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 TotalView for HPC 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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>di</td>
<td>ddisable</td>
<td>Temporarily disables action points</td>
</tr>
</tbody>
</table>

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

- \textbf{now}
  
  Includes RTLD_NOW in the \texttt{dlopen} command's mode argument. (Now immediately resolves all undefined symbols.)

- \textbf{lazy}
  
  Includes RTLD_LAZY in the \texttt{dlopen} command's mode argument. (Lazy tries to resolve unresolved symbols as code is executed, rather than now.)

- \textbf{local}
  
  Includes RTLD_GLOBAL in the \texttt{dlopen} command's mode argument. (Local makes library symbols unavailable to libraries that the program subsequently loads.) This argument is the default.

- \textbf{global}
  
  Includes RTLD_LOCAL in the \texttt{dlopen} command's mode argument. (Global makes library symbols available to libraries that the program subsequently loads.)

- \textbf{mode int}
  
  The integer arguments are ORed into the other mode flags passed to the \texttt{dlopen()} function. (See your operating system's documentation for information on these flags.)

filespec

The shared library to load.

- \textbf{list}
  
  Displays information about the listed DLL IDs. If you use \texttt{ddlopen} without arguments or use the -\texttt{list} option without a DLL ID list (\texttt{ddlopen -list}), TotalView displays information about all DLL IDs.

\textbf{dll-ids}

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

Description
The \texttt{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 TotalView for HPC.

For a \texttt{filespec} argument, TotalView performs a \texttt{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 \texttt{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 \texttt{ddlopen} command.

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

The \texttt{-now} and \texttt{-lazy} options indicate whether \texttt{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 \texttt{-local} and \texttt{-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 \texttt{-global} option. If you don't specify an option, the default is the \texttt{-local} option.)

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

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

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

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

The \texttt{ddlopen} command prints its output directly to the console.

Examples

\texttt{ddlopen "mpistat.so"}

1

Loads the \texttt{mpistat.so} library file. The return value (1) indicates the process into which TotalView loaded the library.
dfocus g ddlopen "mpistat.so(mpistat.o)"

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

ddlopen -lazy -global "mpistat.so"

Loads mpistat.so into process 1, and does not resolve outstanding application symbol requests to point to mpistat. However, TotalView uses the symbols in this library if it needs them.

ddlopen
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 TotalView for HPC User Guide
TV::dllCommand
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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>ddown</td>
<td>Moves down the call stack</td>
</tr>
</tbody>
</table>

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

dupCommand
**denable**

**Enables action points**

**Format**

Enables some action points

\[ \text{denable} \ action-point-list \ [\ -block\ number-list \] \]

Enables all disabled action points in the current focus

\[ \text{denable} \ -a \]

**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 \text{dactions} command to obtain a list of these numbers.

**Description**

The \text{denable} command reactivates action points that you previously disabled with the \text{ddisable} command. The \text{-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 \text{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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>en</strong></td>
<td><strong>denable</strong></td>
<td>Enables action points</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{denable} \ 3 \ 4 \]

Enables two previously identified action points.

\[ \text{dfocus} \ \{p1\ p2\} \ \text{denable} \ -a \]

Enables all action points associated with processes 1 and 2. This command does not affect settings associated with other processes.

\[ \text{en} \ -a \]

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 Points in the NextGen TotalView for HPC User Guide
ddisableCommand
dbarrierCommand
dbreakCommand
dwatchCommand
**dexamine**
Displays memory contents

**Format**

**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 word actid 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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>dexamine</td>
<td>Examines (dumps) memory</td>
</tr>
</tbody>
</table>

## Examples

d1.<> dexamine -f b {dbl_array[1]}
0x7fffffff0d70e8: 01000000000011011001100110011001100110011001100110011001100110011
0x7fffffff0d70f0:

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

d1.<> dexamine -wordsize 8 {dbl_array[1]}
0x7fffffff0d70e8: 0x4003333333333333
0x7fffffff0d70f0:

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]}
0x7fffffff0d70e8: 00400031463146314631463
0x7fffffff0d70f0:

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 TotalView for HPC), 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=0xffffbef6c8 [fork.cxx#1278]
#
# Use the dflush command with a suspened ID argument to remove
# all frames up to and including the one associated with
# suspened 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=0xffffbef400 [fork.cxx#310]
1 ***** Eval Function Call (8) ****************
2 nothing2 PC=0x00012520, FP=0xffffbef4f0 [fork.cxx#310]
3 ***** Eval Function Call (7) ****************
4 forker PC=0x00013fd8, FP=0xffffbef648 [fork.cxx#1120]
5 fork_wrap PC=0x00014780, FP=0xffffbef6c8 [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=0xffffbef648 [fork.cxx#1120]
1 fork_wrap PC=0x00014780, FP=0xffffbef6c8 [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 Chapter 21, “Group, Process and Thread Control” of the TotalView for HPC User Guide.
Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>dfocus</td>
<td>Changes the object upon which a command acts</td>
</tr>
</tbody>
</table>

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.
CLI Commands

\f \{\text{breakpoint(a) | watchpoint(a)}\} \text{ st}

Shows all threads that are stopped at breakpoints or watchpoints.

\f \{\text{stopped(a) - breakpoint(a)}\} \text{ st}

Shows all stopped threads that are not stopped at breakpoints.

Chapter 21 of the *TotalView for HPC User Guide* contains additional dfocus examples.

**RELATED TOPICS**

Using Groups, Processes, and Threads in the *TotalView for HPC 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:

```
dga
  lb_dist
    Handle -1000
    Ghosts yes
    C type $double[129][129][27]
    Fortran Type \n    $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 *TotalView for HPC User Guide*
**CLI Commands**

**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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td><strong>dgo</strong></td>
<td>Resumes execution</td>
</tr>
<tr>
<td>G</td>
<td><code>{dfocus g dgo}</code></td>
<td>Resumes group</td>
</tr>
</tbody>
</table>

**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.

`t g`

Continues a single thread.

**RELATED TOPICS**

- Starting Processes and Threads in the *TotalView for HPC User Guide*
- `dcont` Command
**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. With-
out 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 twodgroups -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] [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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gr</td>
<td>dgroups</td>
<td>Manipulates a group</td>
</tr>
</tbody>
</table>

**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
Using the Group Editor in the TotalView for HPC User Guide
Using Groups, Processes, and Threads in the TotalView for HPC User Guide
Setting Group Focus in the TotalView for HPC User Guide
dhalt

Suspects 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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>dhalt</td>
<td>Suspends execution</td>
</tr>
<tr>
<td>H</td>
<td>{dfocus g dhalt}</td>
<td>Suspends group execution</td>
</tr>
</tbody>
</table>

**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**

- Stopping Processes and Threads in the TotalView for HPC User Guide
- Updating Process Information in the TotalView for HPC User Guide
**dheap**

Controls MemoryScape with TotalView 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 HIA output
   dheap -verbosity [ subcommands ]

Displays the Memory Debugger version number
   dheap -version

Arguments
   -status
       Displays the current state of MemoryScape. This tells you if a process is capable of having its heap operations traced, and whether NextGen TotalView for HPC will notify you if a notifiable heap event occurs. If NextGen TotalView for HPC 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 tells NextGen TotalView for HPC to use the MemoryScape agent to record heap events the next time you start the program. The -disable option tells NextGen TotalView for HPC not to use the agent the next time you start your program.
   -apply_config { default | filename }
       Applies configuration settings in a file to MemoryScape. If you type default, MemoryScape 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, MemoryScape assumes that the extension is .hiarc. That is, while you can specify a file named foo.foobar, you cannot specify a file foo as MemoryScape 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, as 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 MemoryScape 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, NextGen TotalView for HPC 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 MemoryScape ignores when it creates a backtrace. As the backtrace includes procedure calls from within MemoryScape, setting a trim value removes them from the backtrace. The default is to exclude MemoryScape 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, NextGen TotalView for HPC 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 (`/`), MemoryScape writes information into a directory, with the `filenames` being the MemoryScape defaults.

**Optional_Subcommands**

**-reverse_diff**

Changes the order in which MemoryScape makes its comparison. That is, MemoryScape normally compares the first named file to the second. This tells MemoryScape to compare 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, MemoryScape writes the information as HTML.

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

Specifies if the comparison uses a process or a memory debugging export (.mdb) 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
The subcommands to this option let you control which agent events cause MemoryScape to stop program execution.

Subcommands
-status (optional)
Tells TotalView to display 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, MemoryScape displays all events. If you enable event filtering, then you can control which events MemoryScape 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 MemoryScape’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 MemoryScape’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 .mdb format. This format’s only purpose is to be imported back into MemoryScape.
-output filename
Names the file to which NextGen TotalView for HPC writes memory information.

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

-[no]check_interior
If you omit the no prefix, this option tells MemoryScape that a memory block should not be consi-
ered as leaked if a pointer is pointing anywhere within the block. NextGen TotalView for HPC 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 and things you need to know if you select html. 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, iso-
lated problems can occur.) This file can be quite large and can take a while to load.

When MemoryScape writes an HTML file, it also creates a subdirectory with a related name contain-
ing additional information. For example, if the output file name is foo.html, MemoryScape creates a
subdirectory named foo_files.

If you need to move the output file, you must also move the related subdirectory to the same direc-
tory.

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

-set_show_backtraces { on | off }
When set to on, NextGen TotalView for HPC 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, NextGen TotalView for HPC includes the source code for the place where the mem-
ory 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
Tells MemoryScape to export a backtrace view instead of a source view. If you also use the -data raw
option, MemoryScape 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 MemoryScape GUI.
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`:

```bash
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, MemoryScape 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, MemoryScape does not remove existing guard blocks.

  - **-reset**
    
    Resets the guards to the MemoryScape’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 MemoryScape.

-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 MemoryScape 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, MemoryScape does not create a guard.

The default sizes are 8 bytes.

Setting the maximum size to zero (0) tells MemoryScape not to limit guard sizes. Zero is the default value.

-set_post_pattern pattern / -set_pre_pattern pattern

Defines the pattern MemoryScape 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**, MemoryScape 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**, MemoryScape 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**, MemoryScape tags or removes the tag from all addresses beginning with **start_address** to the end of the heap.

-hoard [ subcommands ]

Tells MemoryScape not to surrender allocated blocks back to your program's heap manager. If you do not specify a subcommand, MemoryScape 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, MemoryScape 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 MemoryScape settings for hoarding back to their initial values.

-set_all_deallocs { on | off }
  Tells MemoryScape whether to hoard deallocated blocks.

-reset_all_deallocs
  Resets MemoryScape 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 attempts 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 MemoryScape 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:
   
<table>
<thead>
<tr>
<th>Specifier</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:y</td>
<td>x &lt;= id &lt;= y</td>
</tr>
<tr>
<td>x</td>
<td>x &lt;= id</td>
</tr>
<tr>
<td>x:0</td>
<td>x &lt;= id</td>
</tr>
<tr>
<td>:y</td>
<td>1 &lt;= id &lt;= y</td>
</tr>
</tbody>
</table>
   
   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, MemoryScape 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 MemoryScape 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
MemoryScape considers a memory block as being referenced if the beginning or an interior portion of the block is referenced.

-no_check_interior
MemoryScape 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:

<table>
<thead>
<tr>
<th>Specifier Condition</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:y x &lt;= id &lt;= y</td>
<td>x &lt;= id</td>
</tr>
<tr>
<td>x x &lt;= id</td>
<td>x &lt;= id</td>
</tr>
<tr>
<td>x: x &lt;= id</td>
<td>x &lt;= id</td>
</tr>
<tr>
<td>x:0 x &lt;= id</td>
<td>x &lt;= id</td>
</tr>
<tr>
<td>:y 1 &lt;= id &lt;= y</td>
<td></td>
</tr>
</tbody>
</table>

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

-[no]notify
Using the -notify option tells TotalView to stop your program's execution when MemoryScape detects a notifiable event, and then print a message (or display a dialog box if you are also using the GUI) that explains what just occurred. MemoryScape can notify you when heap memory errors occur or when your program deallocates or reallocates tagged blocks. Using the -nonotify option tells TotalView not to stop execution. Even if you specify the -nonotify option, TotalView tracks heap events.

-paint [ subcommands ]
The painting feature allows the HIA to be instructed to fill, or paint, 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}
The on options enable block painting. They tell MemoryScape to paint 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 MemoryScape 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 MemoryScape 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 MemoryScape 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 align-
  ment of 0 indicates that the default alignment for the platform should be used. An alignment of 2 en-
  sures 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 con-
  trol 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 de-
  tected. 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 \texttt{id\_range}. For \texttt{id\_range} formats, see \texttt{-status} above.

\texttt{-reset\_mode} / \texttt{-reset\_pre\_fence} / \texttt{-reset\_post\_fence} / \texttt{-reset\_use\_after\_free} / \texttt{-reset\_alignment} / \texttt{-reset\_fence\_size} / \texttt{-reset\_exit\_value} / \texttt{-reset\_end\_aligned}

Unsets the TotalView settings for the above controls.

\texttt{-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, MemoryScape either tags all allocated blocks or removes the tag from all tagged blocks.

\textbf{Subcommands}

\texttt{-[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 \texttt{-no-hoard\_on\_dealloc} subcommand.

If you use this option, the memory tracker only hoards tagged blocks. In contrast, if you use the \texttt{dheap -hoard -set\_all\_deallocs} on command, MemoryScape hoards all deallocated blocks.

\texttt{-[no]notify\_dealloc} / \texttt{-[no]notify\_realloc}

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

\texttt{start\_address}

If you only specify a \texttt{start\_address}, MemoryScape either tags or removes the tag from the block that contains this address. The action it performs depends on the subcommand you use.

\texttt{end\_address}

If you also specify an \texttt{end\_address}, MemoryScape either tags all blocks beginning with the block containing the \texttt{start\_address} and ending with the block containing the \texttt{end\_address}, or removes the tag. The action it performs depends on the subcommand you use. If \texttt{end\_address} is 0 (zero) or you do not specify an \texttt{end\_address}, MemoryScape tags or removes the tag from all addresses beginning with \texttt{start\_address} to the end of the heap.

\texttt{-verbosity [ subcommands ]}

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

\textbf{Subcommands}

\texttt{-status}

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

\texttt{-reset}

Restores the verbosity setting to its default.

\texttt{-set verbosity}

Controls how much output MemoryScape writes to its output file. By default, this is file descriptor 1. Higher verbosity values tell MemoryScape to write more information. Setting \texttt{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 NextGen TotalView for HPC with which the HIA was released is also shown.

Description

The dheap command controls the MemoryScape product from the command line. You must have the MemoryScape product installed and a valid license.

Here are some of the things you can do with MemoryScape:

- Tell NextGen TotalView for HPC to use the MemoryScape 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 MemoryScape writes guard blocks, you can ask it to tell you 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.
- Use Red Zones to detect bounds and use-after-free errors.
- 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 MemoryScape. You must do this before your program begins executing. If you try to do this after execution starts, NextGen TotalView for HPC tells you that it will enable MemoryScape when you restart your program. For example:

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

You can tell MemoryScape 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, NextGen TotalView for HPC stops the process when it detects one of these events. MemoryScape is always monitoring heap events even if you turned notification off, but NextGen TotalView for HPC 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 NextGen TotalView for HPC creates, it is the only process where NextGen TotalView for HPC can control MemoryScape'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, NextGen TotalView for HPC simply attaches to them.

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:

<table>
<thead>
<tr>
<th>Flag Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>Operation in progress</td>
</tr>
<tr>
<td>0x0002</td>
<td>notify_dealloc: you will be notified if the block is deallocated</td>
</tr>
<tr>
<td>0x0004</td>
<td>notify_realloc: you will be notified if the block is reallocated</td>
</tr>
<tr>
<td>0x0008</td>
<td>paint_on_dealloc: MemoryScape will paint the block when it is deallocated</td>
</tr>
<tr>
<td>0x0010</td>
<td>dont_free_on_dealloc: MemoryScape will not free the tagged block when it is deallocated</td>
</tr>
<tr>
<td>0x0020</td>
<td>hoarded: MemoryScape is hoarding the block</td>
</tr>
</tbody>
</table>

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]
   :
   ...

RogueWave
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 re-start.

- **-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.
-**gotoBookmark ID**

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

-**goLive**

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

-**showBookmarks**

  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.

-**deleteBookmark ID**

  Deletes the bookmark with the given ID.

-**clearBookmarks**

  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 -newUI 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 **createBookmark**

  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.

-**goTime time** — deprecated: use **gotoBookmark**

  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 will 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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>replay</code></td>
<td><code>dhistory</code></td>
<td>Performs actions upon ReplayEngine.</td>
</tr>
</tbody>
</table>

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_jsl_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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hp</code></td>
<td><code>{dhold -process}</code></td>
<td>Holds the focus process</td>
</tr>
<tr>
<td><code>HP</code></td>
<td><code>{f g dhold -process}</code></td>
<td>Holds all processes in the focus group</td>
</tr>
<tr>
<td><code>ht</code></td>
<td><code>{f t dhold -thread}</code></td>
<td>Holds the focus thread</td>
</tr>
<tr>
<td><code>HT</code></td>
<td><code>{f g dhold -thread}</code></td>
<td>Holds all threads in the focus group</td>
</tr>
<tr>
<td><code>htp</code></td>
<td><code>{f p dhold -thread}</code></td>
<td>Holds all threads in the focus process</td>
</tr>
</tbody>
</table>

**Examples**

```bash
f W HT
```  

Holds all worker threads in the focus group.

```bash
f s HP
```  

Holds all processes in the share group.

```bash
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 TotalView for HPC User Guide
dunholdCommand
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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>dkill</td>
<td>Terminates a process's execution</td>
</tr>
</tbody>
</table>

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 a Session from Your Shell in the NextGen TotalView for HPC User Guide
- Starting Your Program in the CLI in the TotalView for HPC User Guide
- Restarting and Deleting Programs in the TotalView for HPC 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`:

```tcl
lappend foo 1 3 5
```

In contrast, the CLI command displays an error message:

```cli
dlappend foo 1 3 5
```

**Examples**

```cli
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

```plaintext
dlist [-n num-lines ]
```

Displays source code relative to a named place

```plaintext
dlist breakpoint-expr [-n num-lines ]
```

Displays source code relative to the current execution location

```plaintext
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 Chapter 9 of the TotalView for HPC 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 42. 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 \texttt{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 \texttt{TAB_WIDTH} variable controls the tab stop width, which defaults to 8. If \texttt{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:

- \texttt{@} An action point is set at this line.
- \texttt{>} The PC for the current stack frame is at the indicated line and this is the leaf frame.
- \texttt{=} 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 \texttt{dwhere} command that indicates the selected frame.

Here are some general rules:

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

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

\textbf{Command alias}

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>\texttt{dlist}</td>
<td>Displays lines</td>
</tr>
</tbody>
</table>

\textbf{Examples}

The following examples assume that the \texttt{MAX_LIST} variables equals 20, which is its initial value.

\texttt{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 1 -e
Lists the lines around the current execution location of thread 2 in process 1.

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

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

Loads debugging information

Format

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

Arguments

-g gid
Sets the control group for the process being added to the group ID specified by gid. This group must already exist. (The CLI GROUPS variable contains a list of all groups.)

{ -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:

```bash
dload -parallel_attach_subset {1 2 4-6 7-max:2} mpirun
```
will attach to ranks 1, 2, 4, 5, 6, 7, 9, 11, 13,...

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

**NOTE >>** Your license limits the number of processes that you can run at the same time.

**Command alias**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo</td>
<td>dload</td>
<td>Loads debugging information</td>
</tr>
</tbody>
</table>

**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.

```bash
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.

```bash
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.

```bash
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 NextGen TotalView for HPC 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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The amount of memory used to store your program's machine code instructions. The text segment is sometimes called the code segment.</td>
</tr>
<tr>
<td>data</td>
<td>The amount of memory used to store initialized and uninitialized data.</td>
</tr>
<tr>
<td>heap</td>
<td>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.</td>
</tr>
<tr>
<td>stack</td>
<td>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.</td>
</tr>
</tbody>
</table>
CLI Commands / dmstat

**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:**

<table>
<thead>
<tr>
<th>image_name</th>
<th>text data</th>
<th>dpids</th>
</tr>
</thead>
<tbody>
<tr>
<td>/lib/i686/libpthread.so.0</td>
<td>32172 27948</td>
<td>1</td>
</tr>
<tr>
<td>/lib/i686/libc.so.6</td>
<td>1050688 122338</td>
<td>1</td>
</tr>
<tr>
<td>/lib/ld-linux.so.2</td>
<td>70240 10813</td>
<td>1</td>
</tr>
</tbody>
</table>

**Examples**

```
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
```

**image information:**

<table>
<thead>
<tr>
<th>image_name</th>
<th>text data</th>
<th>dpids</th>
</tr>
</thead>
<tbody>
<tr>
<td>...../ry/forked_mem_exampleLINUX 2524 16778479</td>
<td>1 5 6 7</td>
<td></td>
</tr>
<tr>
<td>/lib/i686/libpthread.so.0</td>
<td>32172 27948</td>
<td>1 5 6 7</td>
</tr>
<tr>
<td>/lib/i686/libc.so.6</td>
<td>1050688 122338</td>
<td>1 5 6 7</td>
</tr>
<tr>
<td>/lib/ld-linux.so.2</td>
<td>70240 10813</td>
<td>1 5 6 7</td>
</tr>
</tbody>
</table>

`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.
RELATED TOPICS

For information on MemoryScape, see *Debugging Memory Problems with MemoryScape™* in the TotalView for HPC documentation.
**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 169.
**CLI Commands**

**/dnext**

**Command alias**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>dnext</td>
<td>Runs the <em>thread of interest</em> (TOI) one statement, while allowing other threads in the process to run.</td>
</tr>
<tr>
<td>N</td>
<td>{dfocus g dnext}</td>
<td>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.</td>
</tr>
<tr>
<td>nl</td>
<td>{dfocus L dnext}</td>
<td>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.</td>
</tr>
<tr>
<td>NL</td>
<td>{dfocus gl dnext}</td>
<td>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.</td>
</tr>
<tr>
<td>nw</td>
<td>{dfocus W dnext}</td>
<td>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.</td>
</tr>
<tr>
<td>NW</td>
<td>{dfocus gW dnext}</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 *NextGen TotalView for HPC User Guide*
- **Debugging Using Process Width** in the *NextGen TotalView for HPC User Guide*
- **Debugging Using Thread Width** in the *NextGen TotalView for HPC User Guide*
- **Creating a Process by Single Stepping** in the *TotalView for HPC User Guide*
- **Stepping and Setting Breakpoints** in the *TotalView for HPC User Guide*
- **Using Stepping Commands** in the *TotalView for HPC 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 169.
### Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>dnexti</td>
<td>Runs the TOI one instruction while allowing other threads in the process to run.</td>
</tr>
<tr>
<td>nil</td>
<td>{dfocus L dnexti}</td>
<td>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.</td>
</tr>
<tr>
<td>NIL</td>
<td>{dfocus gl dnexti}</td>
<td>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.</td>
</tr>
<tr>
<td>niw</td>
<td>{dfocus W dnexti}</td>
<td>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.</td>
</tr>
<tr>
<td>NIW</td>
<td>{dfocus gW dnexti}</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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.

**ftn**

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 NextGen TotalView for HPC User Guide
- Debugging Using Process Width in the NextGen TotalView for HPC User Guide
- Debugging Using Thread Width in the NextGen TotalView for HPC User Guide
- Creating a Process by Single Stepping in the TotalView for HPC User Guide
- Stepping and Setting Breakpoints in the TotalView for HPC User Guide
- Using Stepping Commands in the TotalView for HPC User Guide

\texttt{dnextiCommand}

\texttt{dstepCommand}

\texttt{dfocusCommand}
**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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ou</td>
<td>dout</td>
<td>Runs the <em>thread of interest</em> (TOI) out of the current function, while allowing other threads in the process to run.</td>
</tr>
<tr>
<td>OU</td>
<td>{dfocus g dout}</td>
<td>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.</td>
</tr>
<tr>
<td>ouL</td>
<td>{dfocus L dout}</td>
<td>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.</td>
</tr>
<tr>
<td>OUL</td>
<td>{dfocus gL dout}</td>
<td>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.</td>
</tr>
<tr>
<td>ouw</td>
<td>{dfocus W dout}</td>
<td>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.</td>
</tr>
<tr>
<td>OUW</td>
<td>{dfocus gW dout}</td>
<td>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.</td>
</tr>
</tbody>
</table>

For additional information on the different kinds of stepping, see the *dstep* on page 169 command 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 TotalView for HPC User Guide
# dprint

Evaluates and displays information

## Format

Prints the value of a variable

```
dprint [-nowait] [ -slice "slice_expr"] variable
```

Prints the value of an expression

```
dprint [-nowait] [ -group_by] [-slice "slice_expr"] [-stats [ -data ]] expression
```

## Arguments

- **-nowait**
  - Evaluates the expression 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, "\[100:110\].

- **-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 if it includes any blanks, and in braces ({}) if it includes brackets ([ ]), dollar signs ($), quotation marks ("), or other Tcl special characters.

## Description

The `dprint` command evaluates and displays a variable or an expression. The CLI interprets the expression by looking up the values associated with each symbol and applying the operators. The result of an expression can be a scalar value or an aggregate (array, array slice, or structure).
If an event such as a $\texttt{stop}$, SEGV, breakpoint occurs, the \texttt{dprint} command throws an exception that describes the event. The first exception subcode returned by \texttt{TV::errorCodes} is the \texttt{susp-eval-id} (a suspension-evaluation-ID). You can use this to manipulate suspended evaluations with the \texttt{dflush} and \texttt{TV::expr} -commands. For example:

\begin{verbatim}
dfocus tdpid.dtid TV::expr get susp-eval-id
\end{verbatim}

\textbf{NOTE >>} If the expression calls a function, the focus must not specify more than one thread for each process.

For the \texttt{-nowait} option, TotalView evaluates the expression in the background. It also returns a \texttt{susp-eval-id} that you can use to obtain the results of the evaluation using \texttt{TV::expr}.

For the \texttt{-slice} option, TotalView uses the argument after \texttt{-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 \texttt{-slice} switch is ignored.

As the CLI displays data, it passes the data through a simple \texttt{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 \texttt{q} stops printing.

Since the \texttt{dprint} command can generate a considerable amount of output, you might want to use the \texttt{capture} on page 21 command to save the output to a variable.

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

\begin{verbatim}
sbf0 = {f3 = 0x03 (3)f4 = 0x04 (4)f5 = 0x05 (5)f20 = 0x000014 (20)f32 = 0x00000020 (32)}
\end{verbatim}

Arrays print in a similar manner; for example:

\begin{verbatim}
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)}
\end{verbatim}

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

\begin{verbatim}
d.1<> p -slice "\[10:20]\" random
\end{verbatim}
The following is an another way of specifying the same slice:

```ini
set my_var [10:20]
p -slice $my_var 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 example illustrates the output from `dprint -stats` command:

```ini
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,

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

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

```ini
{
<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
...

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
...

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
...

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
...

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
...

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
...

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
...

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:

```c
float bint;
```

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

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

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

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

TotalView assumes the first array address is the address of what \texttt{bp} is pointing to. So, the following command displays the array:

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

You can also use the \texttt{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:

```c
dprint \$r1
```

\textbf{NOTE} >> Do not use a $ when asking the \texttt{dprint} command to display your program's variables.

\textbf{Command alias}

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{p}</td>
<td>\texttt{dprint}</td>
<td>Evaluates and displays information</td>
</tr>
</tbody>
</table>

\textbf{Examples}

\texttt{dprint scalar\_y}

Displays the values of variable \texttt{scalar\_y} in all processes and threads in the current focus.

\texttt{p argc}

Displays the value of \texttt{argc}.

\texttt{p argv}

Displays the value of \texttt{argv}, along with the first string to which it points.

\texttt{p \{argv[argc-1]\}}

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

\texttt{dfocus p1 dprint scalar\_y}

Displays the values of variable \texttt{scalar\_y} for the threads in process 1.

\texttt{f 1.2 p arrayx}

Displays the values of the array \texttt{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.

```bash
def {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

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

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:

```bash
f t1.1 TV::expr get 1 result 2
def t2.1 TV::expr get 1 result
```

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

```bash
f t3.1 TV::expr get 1 result 5
```

**RELATED TOPICS**

- **Examining and Editing Data** in the *NextGen TotalView for HPC User Guide*
- **Entering Expressions** in the *NextGen TotalView for HPC User Guide*
- `TV::errorCodesCommand`
- `TV::exprCommand`
**dptsets**

Shows the status of processes and threads

**Format**

```
dptsets [ptset_array] ...
```

**Arguments**

`ptset_array`

An optional array that indicates the P/T sets to show. An element of the array can be a number or it can be a more complicated P/T expression. (For more information, see “Using P/T Set Operators” in Chapter 21, “Group, Process, and Thread Control” of the *TotalView for HPC 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 Chapter 21, “Group, Process and Thread Control” of the *TotalView for HPC 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
```
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: 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=0x3ff8000d5758
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 147 for more information.)
The **drerun** command differs from the **drun** command in that:

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

- If the process already exists, the **drun** command does not restart it. (If you must use the **drun** command, you must first kill the process.) In contrast, the **drerun** command kills and then restarts the process.

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

**Issues When Using Starter Programs**

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

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

```
drun < in.txt
```

**Command alias**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rr</td>
<td>{drerun}</td>
<td>Restarts processes</td>
</tr>
</tbody>
</table>

**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 *NextGen TotalView for HPC User Guide*
- **dgo**Command
- **dgo**Command
- **capture**Command
**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**

```bash
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`.

- `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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>drun</td>
<td>Starts or restarts processes</td>
</tr>
</tbody>
</table>

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 NextGen TotalView for HPC User Guide
Starting Processes and Threads in the “Manipulating Processes and Threads” chapter of the TotalView for HPC 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 Manager in the NextGen TotalView for HPC User Guide
Managing Sessions in the NextGen TotalView for HPC User Guide
dattachCommand
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 namespace.

    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 Chapter 5, “TotalView Variables,” on page 281.

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 one or 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), _ZN5EnumSIS5ColorEC1ES0_ (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:

```bash
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\n**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\nskip 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\nas 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\nmost 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 \texttt{EnumS<Color>::EnumS(Color)}, then a **dstep** at line 12 would step over the\ncall 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 opera-
tions, such as the **dstep** and **dnext** commands or the **Step** and **Next** buttons or menu items in the\ngraphical user interface.

   A skip through rule tells TotalView to ignore any source-line information for the function, so that single step-
ning does not stop at source lines within the function. However, if the function being skipped through calls\nanother function, that call is handled according to original single-stepping operation. Skip through is most\nuseful 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 \texttt{Callback}, then a **dstep** at line 32 would step through the code in \texttt{Call-
back} and the program would stop inside \texttt{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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>The skip rule ID number.</td>
</tr>
<tr>
<td>Enb</td>
<td>Enabled or disabled.</td>
</tr>
<tr>
<td>How</td>
<td>Skip over or skip through.</td>
</tr>
<tr>
<td>Glob</td>
<td>File is a <code>file-glob</code> or <code>filename</code>.</td>
</tr>
<tr>
<td>File</td>
<td>The <code>filename</code>, <code>file-glob</code>, or none.</td>
</tr>
<tr>
<td>RE</td>
<td>Function is a <code>function-regexp</code> or <code>function-name</code>.</td>
</tr>
<tr>
<td>Function</td>
<td>The <code>function-name</code>, <code>function-regexp</code>, or none.</td>
</tr>
</tbody>
</table>

**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 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 -rfun func (^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 -rfun func (^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 \textit{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

\textit{id}

A rule’s ID number.

\textit{transform_name}

A transform’s name.

\textit{-filter | -f test_function_list}

Required argument for all rules. Its value \textit{test_function_list} is a comma-separated list of filter functions.

\textit{-implementation | -i path}

The path of a script or shared library that supports the \textit{modify} operation.

\textbf{NOTE} >> This is not implemented in the current release. The only implemented transforms are provided by the debugger and are listed as \textit{built-in} using \textit{dstacktransform list}.

\textit{-name | -n string}

A transform’s name.

\textit{-transform | -t name}

Indicates the transform name that provides the implementation of the \textit{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:

```tcl
> 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:

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

> 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

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

### 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 NextGen TotalView for HPC 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 }]

**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:**

cancel_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
-pcount
   Alias for the -ptlist_element_count argument
-ptlist_element_count number
   Displays, at maximum, number elements (comma separated plists or ptlists) in the process/thread com-
   pressed list that is shown in a reduced dstatus display. If a reduction results in exceeding the ptlist_ele-
   ment_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 re-
   maining 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 dis-
   played.
-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.
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 DPIIDs (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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>st</td>
<td><code>dstatus</code></td>
<td>Shows current status</td>
</tr>
<tr>
<td>ST</td>
<td><code>{dfocus g dstatus}</code></td>
<td>Shows group status</td>
</tr>
</tbody>
</table>

**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`
CLI Commands / dstatus

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

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

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

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

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

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

f L ST
Shows status for every thread in the share group that is at the same PC as the thread of interest (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]
snore : 2:3[p1.3, p2.4-5]

The above output displays the reduction produced by the group_by command as a series of ptlists. (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 NextGen TotalView for HPC 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 **dskipCommand** 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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>dstep</td>
<td>Runs the TOI one statement, while allowing other threads in the process to run.</td>
</tr>
<tr>
<td>S</td>
<td>{dfocus g dstep}</td>
<td>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.</td>
</tr>
<tr>
<td>sl</td>
<td>{dfocus L dstep}</td>
<td>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.</td>
</tr>
<tr>
<td>SL</td>
<td>{dfocus gL dstep}</td>
<td>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.</td>
</tr>
<tr>
<td>sw</td>
<td>{dfocus W dstep}</td>
<td>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.</td>
</tr>
<tr>
<td>SW</td>
<td>{dfocus gW dstep}</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 NextGen TotalView for HPC User Guide
Creating a Process by Single Stepping in the “Manipulating Processes and Threads” chapter of the TotalView for HPC User Guide
dsteplCommand
dnextCommand
dfocusCommand
**dstei**

Steps machine instructions, stepping into subfunctions

**Format**

*dstei* [-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 *dstei* command executes assembler instruction lines; that is, it advances the program by single instructions.

The optional *num-steps* argument defines the number of *dstei* operations to perform. If you do not specify *num-steps*, the default is 1.

For more information, see *dstep* on page 169.
## Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>si</td>
<td>dstepi</td>
<td>Runs the <em>thread of interest</em> (TOI) one instruction while allowing other threads in the process to run.</td>
</tr>
<tr>
<td>SI</td>
<td>{dfocus g dstepi}</td>
<td>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.</td>
</tr>
<tr>
<td>sil</td>
<td>{dfocus L dstepi}</td>
<td>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.</td>
</tr>
<tr>
<td>SIL</td>
<td>{dfocusgL dstepi}</td>
<td>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.</td>
</tr>
<tr>
<td>siw</td>
<td>{dfocus W dstepi}</td>
<td>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.</td>
</tr>
<tr>
<td>SIW</td>
<td>{dfocus gW dstepi}</td>
<td>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.</td>
</tr>
</tbody>
</table>

## 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 NextGen TotalView for HPC User Guide
Creating a Process by Single Stepping in the “Manipulating Processes and Threads” chapter of the TotalView for HPC User Guide
Stepping and Setting Breakpoints in the “Manipulating Processes and Threads” chapter of the TotalView for HPC User Guide
Using Stepping Commands in the TotalView for HPC 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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uhp</td>
<td><code>{dfocus p dunhold -process}</code></td>
<td>Releases the focus process</td>
</tr>
<tr>
<td>UHP</td>
<td><code>{dfocus g dunhold -process}</code></td>
<td>Releases the processes in the focus group</td>
</tr>
<tr>
<td>uht</td>
<td><code>{dfocus t dunhold -thread}</code></td>
<td>Releases the focus thread</td>
</tr>
<tr>
<td>UHT</td>
<td><code>{dfocus g dunhold -thread}</code></td>
<td>Releases all threads in the focus group</td>
</tr>
<tr>
<td>uhtp</td>
<td><code>{dfocus p dunhold -thread}</code></td>
<td>Releases the threads in the current process</td>
</tr>
</tbody>
</table>

**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 *TotalView for HPC User Guide*
Holding and Releasing Processes and Threads in the “Manipulating Processes and Threads” chapter of the TotalView for HPC 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

ddebugger-var

  Name of the CLI variable whose default setting is being restored.

-alld

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 Chapter 5, “TotalView Variables,” on page 281 for information on these variables.

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

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

\[ \texttt{duntil \ [-back \ ] \ line-number} \]

Runs to an absolute address

\[ \texttt{duntil \ [-back \ ] \ -address \ \textit{addr}} \]

Runs into a function

\[ \texttt{duntil \ [-back \ ] \ proc-name} \]

**Arguments**

- \[ \texttt{-b \ [-back]} \]
  
  (ReplayEngine only). Moves back in execution time to the most recent point at which the other argument (\textit{line-number}, \textit{address}, or \textit{proc-name}) was executed.

- \[ \texttt{line-number} \]
  
  A line number in your program.

- \[ \texttt{-address \ \textit{addr}} \]
  
  An absolute address in your program.

- \[ \texttt{proc-name} \]
  
  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 \texttt{p width}) or the control group (for \texttt{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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>un</td>
<td>duntil</td>
<td>Runs the TOI until it reaches a target, while allowing other threads in the process to run.</td>
</tr>
<tr>
<td>UN</td>
<td>{dfocus g duntil}</td>
<td>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.</td>
</tr>
<tr>
<td>unl</td>
<td>{dfocus L duntil}</td>
<td>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.</td>
</tr>
<tr>
<td>UNL</td>
<td>{dfocus gL duntil}</td>
<td>Runs lockstep threads in the share group until they reach the target. Other threads in the control group run freely.</td>
</tr>
<tr>
<td>unw</td>
<td>{dfocus W duntil}</td>
<td>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.</td>
</tr>
<tr>
<td>UNW</td>
<td>{dfocus gW duntil}</td>
<td>Runs worker threads in the same share group to a target. All other threads in the control group run freely.</td>
</tr>
</tbody>
</table>

### 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 TotalView for HPC User Guide

**Using Groups, Processes, and Threads** in the Group, Process, and Thread Control chapter of the TotalView for HPC User Guide

**Using Run To and duntil Commands** in the “Stepping (Part I)” section of the Group, Process, and Thread Control chapter of the TotalView for HPC 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**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td><code>dup</code></td>
<td>Moves up the call stack</td>
</tr>
</tbody>
</table>

**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 pl 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] [[ -llang ] -e expr ] [ -typetype]
```

Defines a watchpoint for an absolute address

```
dwatch -address addr-length byte-count [ -g | -p ] [[ -llang ] -e expr ] [ -typetype]
```

**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.

- **-llang**
  
  Specifies the language in which you are writing an expression. The values you can use for **llang** 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 (`{}`).

- **-typetype**
  
  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 OS X, and IBM PowerPC computers running Linux Power.

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`
- `dwhatvariable`

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

See Chapter 8, “Using Watchpoints” in the *TotalView for HPC 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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wa</td>
<td>dwatch</td>
<td>Defines a watchpoint</td>
</tr>
</tbody>
</table>

### 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.

```plaintext
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.

```plaintext
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.

```plaintext
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.

```plaintext
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.

```plaintext
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.

```plaintext
wa my_vbl -type long
-e {if ($newval == 0x11ffff38) $stop;}
```

Watches the `my_vbl` variable and triggers when `0x11ffff38` is stored in it.

```plaintext
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

- **Using Watchpoints** in the “Setting Action Points” chapter of the *TotalView for HPC User Guide*
- **Writing Code Fragments** in the “Evaluating Expressions” chapter of the *TotalView for HPC User Guide*
- **dactionsCommand**
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 dsetor alias commands.

The focus constrains the query to a particular context.

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

Command alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wh</td>
<td>dwhat</td>
<td>Determines what a name refers to</td>
</tr>
</tbody>
</table>

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: 0x11fffe0c0
    Scope: #fork_loop.hxx#snore \n        (Scope class: Any) \n    Address class: auto_var \n        (Local variable)

wh timeval

In process 1: Type name: struct timeval; Size: 8 bytes; \n    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: 0x11fffff240
    Scope: #check_structs.cxx#main 
    Scope class: Any)
    Address class: auto_var (Local variable)

wh check_struct

In process 2:
Type name: struct check_struct; 
    Size: 80 bytes; Category: Structure
Fields in type:
    {
        field1_int(int(32 bits)
        field2_char(char(8 bits)
        field2_chars$string(2)(16 bits)
        <padding>$char(8 bits)
        field3_int(int(32 bits)
        field2_ulong(unsigned long(64 bits)
        field3_uint(unsigned int(32 bits)
        field_substruct sub_st(320 bits){
            field1_int(32 bits)
            <padding>$char(4)(32 bits)
            field2_long(long(64 bits)
            field2_ulong(unsigned long(64 bits)
            field3_uint(unsigned int(32 bits)
            en1enum en_check (32 bits)
            field3_double(double(64 bits)
        }
    ...
    }

RELATED TOPICS

dstatus Command
dwhere Command
dwhere

Displays the current execution location and call stack

**Format**

Displays locations in the call stack

```
```

Displays all locations in the call stack

```
```

**Arguments**

- **-all**
  Shows all levels of the call stack. This is the default.

- **-level | -I 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_LEVELS variable 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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td><code>dwhere</code></td>
<td>Displays the current location in the call stack</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wof</td>
<td>{dworker false}</td>
<td>Removes the focus thread from the workers group</td>
</tr>
<tr>
<td>wot</td>
<td>{dworker true}</td>
<td>Inserts the focus thread into the workers group</td>
</tr>
</tbody>
</table>

RELATED TOPICS

Organizing Chaos and Creating Groups in the “About Groups, Processes, and Threads” chapter of the TotalView for HPC User Guide

Setting Group Focus in the “Group, Process, and Thread Control” chapter of the TotalView for HPC User Guide
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 detach 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 | \n    -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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>help</td>
<td>Displays help information</td>
</tr>
</tbody>
</table>

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**

- `exitCommand`
spurs

Returns information on the spurs library use

**Format**
Displays help information

```
spurs [ help ]
```

Adds directories to the OBJECT_SEARCH_PATH variable

```
spurs add [ directory directory ... ]
```

Creates an image-qualified breakpoint

```
spursbreak [ spu-image-name spu-source-location-expression ]
```

Deletes breakpoints

```
spurs delete breakpoint-id ... 
```

Shows the directories in which TotalView searches for SPURS SPU ELF executables

```
spurs info [ directory | break ]
```

Prints information about the kernel, the taskset, tasks, and other SPURS objects

```
spurs print [ kernel [ eaSpurs ] | barrier eaBarrier | event_flag eaEventFlagSet | Ifqueue eaLFQueue | queue eaQueue | semaphore eaSemaphore | taskset [ eaTaskset ] | task eaTaskset taskId
```

Scans for information—this is a no-op

```
spurs scan
```

**Arguments**

`directory`

The directory or directories to be added to the CLI's OBJECT_SEARCH_PATH variable. For example:

```
spurs add directory directory1 directory2
```

Notice that directory names are separated by space characters.

`spu-image-name`

The name of the image that is or will be loaded by TotalView

`spu-source-location-expression`

An expression that resolves to a specific line in the image. For information on location expressions, see `dbreak` on page 41.

`breakpoint-id`

The action point ID to delete
**Description**

Modeled after the GDB “spurs” command, the **spurs** command was created so that developers who are familiar the GDB command have a similar set of commands in TotalView. However, not all GDB “spurs” commands are implemented.

TotalView supports the SPURS library. Here’s this library’s description in the SPURS documentation:

libspurs is a user-level thread library for SPUs. In a SPURS environment (SPU Runtime System), SPU threads are managed by SPUs. For this reason, thread switching is more efficient than under PPU management and requires no PPU resources. Using SPURS also makes it easier to synchronize threads and adjust the load balance on multiple SPUs. SPURS is furthermore highly extensible and allows users to define their own thread models as necessary.

**spurs [ help ]**

To access help on the **spurs** command:

- Enter `spurs` to return a one-line description of its commands.
- Enter `spurs help` to display more information about each `spurs` subcommand.

**spurs add [ directory directory ... ]**

Displays either a one-line description of this command or adds directories to search when TotalView looks for SPURS SPU executables.

- `spurs add` writes a one-line description of this command.
• `spurs add directory directory` adds a directory or directories to the CLI's `OBJECT_SEARCH_PATH` variable. This variable contains the path used when searching for SPU ELF executable files. The directories are placed at the beginning of the list in the order in which they are named. If a directory is already in the list, the previously named directory is removed.

This command returns the modified `OBJECT_SEARCH_PATH` variable.

`spurs break [spu-image-name spu-source-location-expression ]`

Displays either a one-line description of this command or adds a breakpoint.

• `spurs break` returns a one-line description of this command.

• `spurs break spu-image-name spu-source-loc-expression` creates an image-qualified breakpoint path. This is identical to the following CLI command:

```
dbreak -pending ##spu-image-name#source-loc-exp
```

This command creates a pending breakpoint that is located only with the image you name. However, if the image has already been loaded, TotalView sets an ordinary breakpoint rather than a pending breakpoint. The focus must be on an SPU thread.

This command returns the action point ID of the created breakpoint. You can use this ID with other CLI commands that act upon breakpoints; for example, `dactions`, `ddelete`, `ddisable`, `denable`, and others.

`spurs delete breakpoint-id ...`

Permanently removes one or more action points. The argument defines which action points to delete. Unlike `spurs break`, this command does not require that the command focus be set to an SPU thread.

`spurs info [ directory | break ]`

• `spurs info` returns a one-line description of this command.

• `spurs info directory` prints the `OBJECT_SEARCH_PATH` state variable

• `spurs info break` prints action point information about action points in the thread in the current focus.

`spurs print`

The `spurs print` command can be used in the following ways:

```
spurs print [ kernel [ eaSpurs ] | barrier eaBarrier | event_flag eaEventFlagSet | lfqueue eaLFQueue | queue eaQueue |]
```

semaphore eaSemaphore |
taskset [ eaTaskset ] |
task eaTaskset taskID

spurs print
   Displays one line of information on using this command.

spurs print kernel
   Displays the kernel context for the SPU threads in the current or named focus. The focus must be one or more
   SPU threads.

   cell_spurs_print_kernel is an alias for this command.

spurs print kerneleaSpurs
   Displays the kernel context at PPU address eaSpurs. The command focus must be one or more PPU threads.

   cell_spurs_print_kernel is an alias for this command.

spurs print barrier eaBarrier
   Displays the barrier object at PPU address eaBarrier. The command focus must be one or more PPU threads.

   cell_spurs_print_barrier_info is an alias for this command.

spurs print event_flag eaEventFlagSet
   Displays the event flag object at PPU address eaEventFlagSet. The command focus must be one or more PPU
   threads.

   cell_spurs_print_event_flag_info is an alias for this command.

spurs print lfqueue eaLFQueue
   Displays the lfqueue object at PPU address eaLFQueue. The command focus must be one or more PPU
   threads.

   cell_spurs_print_lfqueue_info is an alias for this command.

spurs print queue eaQueue
   Displays the queue object at PPU address eaQueue. The command focus must be one or more PPU threads.

   cell_spurs_print_queue_info is an alias for this command.

spurs print semaphore eaSemaphore
   Displays the semaphore object at PPU address eaSemaphore. The command focus must be one or more PPU
   threads.

   cell_spurs_print_semaphore_info is an alias for this command.

spurs print taskset
   Prints the taskset for the focus SPU threads. The command focus must be one or more SPU threads.

   cell_spurs_print_taskset is an alias for this command

spurs print taskset eaTaskset
   Prints the taskset at PPU address eaTaskset. The command focus must be one or more SPU threads.
**cell_spurs_print_taskset** is an alias for this command.

**spurs print task eaTaskset taskID**

Prints the task at index *taskID* in the taskset at PPU address *eaTaskset*. The command focus must be one or more PPU threads.

**cell_spurs_print_task** is an alias for this command.

**spurs scan**

This command is for compatibility with GDB. Unlike the GDB command, this command is a no-op as TotalView has no need to scan for SPU executables because searches for SPU executables happen dynamically.
**stty**

Sets terminal properties

**Format**

```
stty [ stty-args ]
```

**Arguments**

*stty-args*

One or more UNIX `stty` command arguments as defined in the `man` page for your operating system.

**Description**

The CLI `stty` command executes a UNIX `stty` command on the `tty` associated with the CLI window, 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
Chapter 3 CLI Namespace Commands

Command Overview

This chapter lists all of CLI commands that are not in the top-level mainspace.

NOTE >> This chapter describes some functionality that exists in the underlying debugging engine TotalView, but may not be supported in the NextGen TotalView for HPC user interface. To access these features, use the Command Line view. See “Overview” on page iii for 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.
**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} {0xff000000900005b6
        0x004005b6} {0xff000000900006b6
        0x004006b6}
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
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
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

**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**

hex2decCommand
**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-ids`.
  
  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**

- ddlopenCommand
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

```bash
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 \n    susp-eval-id \n    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 \n    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 focused on threads 1.1, 2.1, and 3.1:

```
d1.<> dfocus {t1.1 t2.1 t3.1} \n    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**

```bash
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 TotalView for HPC 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_processesCommand
  focus_threadsCommand
dfocusCommand
  Using Groups, Processes, and Threads in the Group, Process, and Thread Control chapter of the TotalView for HPC 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_processesCommand
  focus_threadsCommand
  dfocusCommand
  Using Groups, Processes, and Threads in the Group, Process, and Thread Control chapter of the TotalView for HPC 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 process 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 TotalView for HPC 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

dec2hexCommand
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); for example, linux-x86 would be returned if the program is running on a 32-bit linux system.

is_parallel
Contains a value indicating if the current process is a parallel process. If it is, its value is 1. Otherwise, its value is 0.

id
The process ID.

image_ids
A list of the IDs of all the images currently loaded into the process both statically and dynamically. The first element of the list is the current executable.
is_parallel
Contains a value indicating if the current process is a parallel process. If it is, its value is 1. Otherwise, its value is 0.

nodeid
The ID of the node upon which the process is running. The ID of each processor node is unique within a cluster.

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` or `x86-64`.

**text_size**

The amount of memory used to store your program's machine code instructions. The text segment is sometimes called the code segment.

**threadcount**

The number of threads in the process.

**threads**

A list of threads in the process.

**vm_size**

The sum of the 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 *TotalView for HPC 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_only` or `TV::dll_read_no_symbols` lists.

For more information, see “Preloading Shared Libraries” in the “Debugging Programs” chapter of the *TotalView for HPC 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 newline, the TV::respond command appends one.

Do not use this command to suppress the MORE prompt in macros. Instead, use the following command:

dset LINES_PER_SCREEN 0

The most common values for response are y and n.

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 need 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**

Dumps 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_regex_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_regexpair]: 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_regexpair]... \
[non-required_property_regexpair]...
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_process_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 TotalView for HPC 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` 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 \n  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 1 lists the properties associated with the symbols information that TotalView stores. Not all of this information will be useful when creating transformations. However, it is possible to come across some of these properties and this information will help you decide if you need to use it in your transformation. In general, the properties used in the transformation files that Rogue Wave Software provided will be the ones that you will use.

Table 1: Symbol Properties

<table>
<thead>
<tr>
<th>Symbol Kind</th>
<th>Has base_name</th>
<th>Has type_index</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate_type</td>
<td>X</td>
<td>X</td>
<td>aggregate_kind, artificial, external_name</td>
</tr>
<tr>
<td>array_type</td>
<td>X</td>
<td>X</td>
<td>artificial, data_addressing, element_addressing, external_name, full_pathname</td>
</tr>
<tr>
<td>Symbol Kind</td>
<td>Has base name</td>
<td>Has type index</td>
<td>Property</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>----------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>block</td>
<td>X</td>
<td>artificial</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>length</td>
</tr>
<tr>
<td>char_type</td>
<td>X</td>
<td>artificial</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>external_name</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td>code_type</td>
<td>X</td>
<td>X</td>
<td>artificial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>external_name</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td>ds_ undiscovered_type</td>
<td>X</td>
<td>X</td>
<td>artificial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td>scope_owner</td>
</tr>
<tr>
<td>enum_type</td>
<td>X</td>
<td>X</td>
<td>artificial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enumerators</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>external_name</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>value_size</td>
</tr>
<tr>
<td>file</td>
<td>X</td>
<td>artificial</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiler_kind</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delayed_symbol</td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>demangler</td>
<td>scope_owner</td>
</tr>
<tr>
<td>float_type</td>
<td>X</td>
<td>X</td>
<td>artificial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>external_name</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>length</td>
</tr>
<tr>
<td>function_type</td>
<td>X</td>
<td>X</td>
<td>artificial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>external_name</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical_scope_owner</td>
</tr>
<tr>
<td>image</td>
<td>X</td>
<td>artificial</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>kind</td>
</tr>
<tr>
<td>int_type</td>
<td>X</td>
<td>X</td>
<td>artificial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>external_name</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>length</td>
</tr>
<tr>
<td>label</td>
<td>X</td>
<td>address_class</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>artificial</td>
<td>kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_pathname</td>
<td>location</td>
</tr>
</tbody>
</table>

Table 1: Symbol Properties
<table>
<thead>
<tr>
<th>Symbol Kind</th>
<th>Has base name</th>
<th>Has type index</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>linenumber</td>
<td>address_class</td>
<td>id</td>
<td>logical_scope_owner scope_owner</td>
</tr>
<tr>
<td></td>
<td>artificial</td>
<td>kind</td>
<td></td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>location</td>
<td></td>
</tr>
<tr>
<td>loader_symbol</td>
<td>address_class</td>
<td>id</td>
<td>location</td>
</tr>
<tr>
<td></td>
<td>artificial</td>
<td>kind</td>
<td>logical_scope_owner scope_owner</td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td>length</td>
<td></td>
</tr>
<tr>
<td>member</td>
<td>X</td>
<td>inheritance</td>
<td>ordinal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>location</td>
<td>type_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>module</td>
<td>X</td>
<td>id</td>
<td>logical_scope_owner scope_owner</td>
</tr>
<tr>
<td></td>
<td>artificial</td>
<td>kind</td>
<td></td>
</tr>
<tr>
<td></td>
<td>full_pathname</td>
<td></td>
<td></td>
</tr>
<tr>
<td>named_constant</td>
<td>X</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length</td>
<td>type_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>namespace</td>
<td>X</td>
<td>id</td>
<td>logical_scope_owner scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>opaque_type</td>
<td>X X</td>
<td>id</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>pathname_reference_symbol</td>
<td>X</td>
<td>kind</td>
<td>resolved_symbol_pathname scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lookup_scope</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>pointer_type</td>
<td>X</td>
<td>kind</td>
<td>target_type_index validator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>qualified_type</td>
<td>X X</td>
<td>id</td>
<td>qualification scope_owner target_type_index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>kind</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td>soid_reference_symbol</td>
<td>X</td>
<td>kind</td>
<td>scope_owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>logical_scope_owner</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>resolved_symbol_id</td>
<td></td>
</tr>
</tbody>
</table>
Table 1: Symbol Properties

<table>
<thead>
<tr>
<th>Symbol Kind</th>
<th>Has base name</th>
<th>Has type index</th>
<th>Property</th>
</tr>
</thead>
</table>
| 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 |
| 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 240 shows how these symbols are related.
Here are definitions of the properties associated with these symbols.

**address_class**
contains the location for a variety of objects such as a `func`, `global_var`, and a `tls_global`.

**aggregate_kind**
One of the following: `struct`, `class`, or `union`.

**artificial**
A Boolean (0 or 1) value where true indicates that the compiler generated the symbol.

**compiler_kind**
The compiler or family of compiler used to create the file; for example, `gnu`, `xlc`, `intel`, and so on.
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

![Diagram of data addressing]

delayed_symbol
Indicates if a symbol has been full or partially read-in. The following constants are or'd and returned: skim, index, line, and full.

demangler
The name of demangler used by your compiler.

element_addressing
The location containing additional operands that let you go from the data's base location to an element.
enumerators
Name of the enumerator tags. For example, if you have something like enum[R,G,B], the tags would be R, G, and B.

external_name
When used in data types, it translates the object structure to the type name for the language. For example, if you have a pointer that points to an int, the external name is int *.

full_pathname
This is the # separated static path to the variable; for example, ##image#file#externalname....

id
The internal object handle for the symbol. These symbols always take the form number|number.

index_type_index
The array type's index type_index; for example, this indicates if the index is a 16-, 32-, 64-bit, and so on.

inheritance
For C++ variables, this string is as follows: [ virtual ] [ { private | protected | public } ] [ base class ]
is_argument
   A true/false value indicating if a variable was a parameter (dummy variable) passed into the function.

kind
   One of the symbol types listed in the first column of the previous table.

language
   A string containing a value such as C, C++, or Fortran.

length
   The byte size of the object. For example, this might represent the size of an array or a subroutine.

location
   The location in memory where an object's storage begins.

logical_scope_owner
   The current scope's owner as defined by the language's rules.

**Figure 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**

```
volatile const int
```

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 245 illustrates how these namespaces are related.

Table 2: Namespace Properties

<table>
<thead>
<tr>
<th>Symbol Namespaces</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>block_symname</td>
<td>base_name</td>
</tr>
<tr>
<td>c_global_symname</td>
<td>base_name, loader_name</td>
</tr>
<tr>
<td></td>
<td>loader_file_path</td>
</tr>
<tr>
<td>c_local_symname</td>
<td>base_name</td>
</tr>
<tr>
<td>c_type_symname</td>
<td>base_name, type_index</td>
</tr>
<tr>
<td>cplusplus_global_symname</td>
<td>base_name, cplusplus_template_types</td>
</tr>
<tr>
<td></td>
<td>cplusplus_class_name, cplusplus_type_name</td>
</tr>
<tr>
<td></td>
<td>cplusplus_local_name, loader_file_path</td>
</tr>
<tr>
<td></td>
<td>cplusplus_overload_list, loader_name</td>
</tr>
<tr>
<td>cplusplus_local_symname</td>
<td>base_name, cplusplus_overload_list</td>
</tr>
<tr>
<td></td>
<td>cplusplus_class_name, cplusplus_template_types</td>
</tr>
<tr>
<td></td>
<td>cplusplus_local_name, cplusplus_type_name</td>
</tr>
<tr>
<td>cplusplus_type_symname</td>
<td>base_name, cplusplus_template_types</td>
</tr>
<tr>
<td></td>
<td>cplusplus_class_name, cplusplus_type_name</td>
</tr>
<tr>
<td></td>
<td>cplusplus_local_name, type_index</td>
</tr>
<tr>
<td></td>
<td>cplusplus_overload_list</td>
</tr>
<tr>
<td>file_symname</td>
<td>base_name, directory_path</td>
</tr>
<tr>
<td>directory_hint</td>
<td></td>
</tr>
<tr>
<td>fortran_global_symname</td>
<td>base_name, loader_file_path</td>
</tr>
<tr>
<td>fortran_module_name</td>
<td>loader_name, fortran_parent_function_name</td>
</tr>
<tr>
<td>fortran_local_symname</td>
<td>base_name, fortran_parent_function_name</td>
</tr>
</tbody>
</table>
Table 2: Namespace Properties

<table>
<thead>
<tr>
<th>Symbol Namespaces</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>fortran_module_name</td>
<td>fortran_parent_function_name</td>
</tr>
<tr>
<td>fortran_type_symname</td>
<td>base_name fortran_module_name type_index</td>
</tr>
<tr>
<td>image_symname</td>
<td>base_name member_name</td>
</tr>
<tr>
<td>label_symname</td>
<td>base_name</td>
</tr>
<tr>
<td>linenumber_symname</td>
<td>linenumber</td>
</tr>
<tr>
<td>loader_symname</td>
<td>loader_file_path loader_name</td>
</tr>
<tr>
<td>module_symname</td>
<td>base_name</td>
</tr>
<tr>
<td>type_symname</td>
<td>type_index</td>
</tr>
</tbody>
</table>

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 looking at the following function prototype:

```c
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* subcommands. Do not use other arguments with this option.

*get*

Gets the values of one or more thread properties. The *other-args* argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the names you enter.

If you use the -all option instead of *object-id*, the CLI returns a list containing one (sublist) element for each object.

*properties*

Lists an object’s properties. Do not use other arguments with this option.

*set*

Sets the values of one or more properties. The *other-args* argument contains paired property names and values.

*object-id*

A thread ID. If you use the -all option, the operation is carried out on all threads in the current focus.

*other-args*

Arguments required by the *get* and *set* subcommands.

**Description**

The TV::thread command lets you examine and set the following thread properties and states:

*canonical_executable_name*

The absolute file name of the program being debugged. If you had entered a relative name, TotalView find this absolute name.

*continue_sig*

The signal to pass to a thread the next time it runs. On some systems, the thread receiving the signal might not always be the one for which this property was set.

*current_ap_id*

The ID of the action point at which the current thread is stopped.
dpid
   The ID of the process associated with a thread.

duid
   The internal unique ID associated with the thread.

held
   A Boolean value (either 1 or 0) indicating if the thread is held. (1 means that the thread is held.) (settable)

id
   The ID of the thread.

manager
   A Boolean value (either 1 or 0) indicating if this is a system manger thread. (1 means that it is a system manager thread.)

pc
   The current PC at which the target is executing. (settable)

sp
   The value of the stack pointer.

state
   The current state of the target. See state_values for a list of states.

state_values
   A list of values for the state property: break, error, exited, running, stopped, and watch.

stop_reason_message
   The reason why the current thread is stopped; for example, Stop Signal.

systid
   The system thread ID.

target_architecture
   The machine architecture upon which the current thread is executing.

target_byte_ordering
   The bit ordering of the current machine. This is either little_endian or big_endian.

target_processor
   The kind of processor upon which the current thread is executing. For example, this could be x86 or x86-64.

Examples
f p3 TV::thread get -all id
   Returns a list of thread IDs for process 3; for example:
       1.1 1.2 1.4

proc set_signal {val} {
   TV::thread set \
   [f t TV::focus_threads] continue_sig $val
Set the starting signal for the focus thread.

```tcl
proc show_signal {} {
    foreach th [TV::focus_threads] {
        puts "Continue_sig ($th): \n        [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 TotalView for HPC User Guide

focus_threadsCommand
groupCommand
processCommand
type

Gets and sets type properties

Format

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

Arguments

action

The action to perform, as follows:

commands

Displays the subcommands that you can use. The CLI responds by displaying these four action subcommands. Do not use other arguments with this option.

get

Gets the values of one or more type properties. The other-args argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the names you enter.

If you use the -all option instead of object-id, the CLI returns a list containing one (sublist) element for each object.

properties

Lists a type's properties. Do not use other arguments with this option.

set

Sets the values of one or more type properties. The other-args argument contains paired property names and values.

object-id

An identifier for an object; for example, 1 represents process 1, and 1.1 represents thread 1 in process 1. If you use the -all option, the operation is carried out on all objects of this class in the current focus.

other-args

Arguments required by the get and set subcommands.

Description

The TV::type command lets you examine and set the following type properties and states:

enum_values

For an enumerated type, a list of {name value} pairs giving the definition of the enumeration. If you apply this to a non-enumerated type, the CLI returns an empty list.

id

The ID of the object.

image_id

The ID of the image in which this type is defined.
language
   The language of the type.

length
   The length of the type.

name
   The name of the type; for example, class foo.

prototype
   The ID for the prototype. If the object is not prototyped, the returned value is {}.

rank
   (array types only) The rank of the array.

struct_fields
   (class/struct/union types only). A list of lists that contains descriptions of all the type's fields. Each sublist contains the following fields:
   
   `{ name type_id addressing properties }

   where:

   name is the name of the field.
   type_id is simply the type_id of the field.
   addressing contains additional addressing information that points to the base of the field.
   properties contains an additional list of properties in the following format:
   “[virtual] [public|private|protected] base class”

   If no properties apply, this string is null.

   If you use get struct_fields for a type that is not a class, struct, or union, the CLI returns an empty list.

target
   For an array or pointer type, returns the ID of the array member or target of the pointer. 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:

```bash
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:

```bash
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}...
```
CLI Namespace Commands

**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 \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

This callback defines a TotalView addressing expression that computes the starting address of an array's first element.

\textbf{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:

\texttt{lower_bounds_callback id}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\textbf{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:

\texttt{upper_bounds_callback id}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\textbf{List Properties}

\textbf{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:

\texttt{list_element_count_addressing_callback id}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{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.

\textbf{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:

\texttt{list_element_data_addressing_callback id}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\textbf{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:

\texttt{list_element_next_addressing_callback id}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{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 \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure and \textit{index} specifies the zero-based position of the member within the struct.

\texttt{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:

\begin{verbatim}
struct_member_name_callback id index
\end{verbatim}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure and \textit{index} specifies the zero-based position of the member within the struct.

\section*{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.

\texttt{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:

\begin{verbatim}
rbtree_head_addressing_callback id
\end{verbatim}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\texttt{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:

\begin{verbatim}
rbtree_head_type_callback id
\end{verbatim}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\texttt{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:

\begin{verbatim}
rbtree_element_left_addressing_callback id
\end{verbatim}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\texttt{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:

\begin{verbatim}
rbtree_element_right_addressing_callback id
\end{verbatim}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate_callback} procedure.

\texttt{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:

\begin{verbatim}
rbtree_element_parent_addressing_callback id
\end{verbatim}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{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 \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate\_callback} procedure.

\textbf{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:

\texttt{hashtable\_element\_count\_addressing\_callback \textit{id}}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{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.

\textbf{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:

\texttt{hashtable\_element\_count\_type\_callback \textit{id}}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate\_callback} procedure.

\textbf{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:

\texttt{hashtable\_element\_addressing\_callback \textit{id}}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate\_callback} procedure.

\textbf{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:

\texttt{hashtable\_begin\_index\_addressing\_callback \textit{id}}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{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\}.

\textbf{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:

\texttt{hashtable\_begin\_index\_type\_callback \textit{id}}

where \textit{id} is the symbol ID of the symbol that was validated using the \texttt{validate\_callback} procedure.

\textbf{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:

\texttt{hashtable\_bucket\_count\_addressing\_callback \textit{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

Overview

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
```

This chapter discusses tvscript command-line options.
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 Chapter 7, “TotalView Command Syntax,” on page 367.

  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 Chapter 7, “TotalView Command Syntax,” on page 367.

Blue Gene/L and Blue Gene/P

The syntax for using tvscript with an MPI on Blue Gene/L and Blue Gene/P systems is:


  -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 be the first argument.

  -mpi-arguments
  The command arguments for mpirun, such as "-cwd", "-mode", and "-partition".

  -args
  Command argument for mpirun that passes to the launched application on the compute node.
mpirun

Required; the executable at the end of the command line.

**Blue Gene/Q with SLURM**

The syntax for using tvscript with an MPI on Blue Gene/Q systems using SLURM is:

```
tvscript [options] -mpi BlueGeneQ-SLURM -np number-of-processes-starter_args [sruntime-arguments]filename [program_args]"srun
```

- `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 srun.

```
srun-arguments
```

The command arguments for srun.

```
filename
```

The program being debugged.

```
program_args
```

The arguments for the program being debugged.

```
srun
```

Required; the executable at the end of the command line.

**Blue Gene/Q for ANL’s Cobalt Job Manager and IBM’s runjob**

The syntax for using tvscript on Blue Gene/Q for ANL’s Cobalt job manager and IBM’s runjob is:

```
tvscript [options] -mpi BlueGeneQ-Cobalt -np number-of-processes-starter_args [runjob-arguments]filename [program_args]"runjob
```

- `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 runjob and be separated by a colon (:).

```
runjob-arguments
```

The command arguments for runjob.

```
filename
```

The program being debugged.

```
program_args
```

The arguments for the program being debugged.
runjob

Required; the executable at the end of the command line.

Blue Gene/Q for the LoadLeveler job manager and IBM's runjob

The syntax for using tvscript on Blue Gene/Q for the LoadLeveler job manager and IBM's runjob: is

tvscript [options] -mpi BlueGeneQ-LoadLeveler -np number-of-processes -starter_args [runjob-arguments] --exe filename[program_args]"runjob

-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 runjob and --exe.

runjob-arguments

The command arguments for runjob.

filename

The program being debugged.

program_args

The arguments for the program being debugged.

runjob

Required; the executable at the end of the command line.

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

-aprun-arguments
The command arguments for aprun.

-mpi
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.

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
```
Batch Debugging Using tvscript

### 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 275 for 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**:

  \[
  \text{event1}=\text{action1}, \text{event2}=\text{action2} \\
  \text{or} \\
  \text{event1}=>\text{action1}, \text{action2}, \text{action3}
  \]

### Table 3: Supported tvscript Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>General event</td>
<td><strong>any_event</strong></td>
<td>A generated event occurred.</td>
</tr>
<tr>
<td>Memory debugging event</td>
<td><strong>addr_not_at_start</strong></td>
<td>Program attempted to free a block using an incorrect address.</td>
</tr>
<tr>
<td></td>
<td><strong>alloc_not_in_heap</strong></td>
<td>The memory allocator returned a block not in the heap; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td><strong>alloc_null</strong></td>
<td>An allocation either failed or returned NULL; this usually means that the system is out of memory.</td>
</tr>
<tr>
<td></td>
<td><strong>alloc_returned_bad_alignment</strong></td>
<td>The memory allocator returned a misaligned block; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td><strong>any_memory_event</strong></td>
<td>A memory event occurred.</td>
</tr>
<tr>
<td></td>
<td><strong>bad_alignment_argument</strong></td>
<td>Program supplied an invalid alignment argument to the heap manager.</td>
</tr>
<tr>
<td></td>
<td><strong>double_alloc</strong></td>
<td>The memory allocator returned a block currently being used; the heap may be corrupt.</td>
</tr>
<tr>
<td></td>
<td><strong>double_dealloc</strong></td>
<td>Program attempted to free an already freed block.</td>
</tr>
<tr>
<td></td>
<td><strong>free_not_allocated</strong></td>
<td>Program attempted to free an address that is not in the heap.</td>
</tr>
<tr>
<td></td>
<td><strong>guard_corruption</strong></td>
<td>Program overwrote the guard areas around a block.</td>
</tr>
</tbody>
</table>
### Table 3: Supported tvscript Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>hoard_low_memory_threshold</td>
<td>Hoard low memory threshold crossed.</td>
<td></td>
</tr>
<tr>
<td>realloc_not_allocated</td>
<td>Program attempted to reallocate an address that is not in the heap.</td>
<td></td>
</tr>
<tr>
<td>rz_overrun</td>
<td>Program attempted to access memory beyond the end of an allocated block.</td>
<td></td>
</tr>
<tr>
<td>rz_underrun</td>
<td>Program attempted to access memory before the start of an allocated block.</td>
<td></td>
</tr>
<tr>
<td>rz_use_after_free</td>
<td>Program attempted to access a block of memory after it has been deallocated.</td>
<td></td>
</tr>
<tr>
<td>rz_use_after_free_overrun</td>
<td>Program attempted to access memory beyond the end of a deallocated block.</td>
<td></td>
</tr>
<tr>
<td>rz_use_after_free_underrun</td>
<td>Program attempted to access memory before the start of a deallocated block.</td>
<td></td>
</tr>
<tr>
<td>termination_notification</td>
<td>The target is terminating.</td>
<td></td>
</tr>
<tr>
<td>actionpoint</td>
<td>A thread hit an action point.</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>An error occurred.</td>
<td></td>
</tr>
<tr>
<td>stopped_at_end</td>
<td>The program is stopped at the end of execution and is about to exit.</td>
<td></td>
</tr>
</tbody>
</table>
For each occurring event, define the action to perform:

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Action</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory debugging actions</td>
<td>check_guard_blocks</td>
<td>Checks all guard blocks and write violations into the log file.</td>
</tr>
<tr>
<td></td>
<td>list_allocations</td>
<td>Writes a list of all memory allocations into the log file.</td>
</tr>
<tr>
<td></td>
<td>list_leaks</td>
<td>Writes a list of all memory leaks into the log file.</td>
</tr>
<tr>
<td></td>
<td>save_memory_debugging_file</td>
<td>Generates and saves a memory debugging file.</td>
</tr>
<tr>
<td></td>
<td>save_text_heap_status_source_view</td>
<td>Generates and saves a text version of the Heap Status Source View Report.</td>
</tr>
<tr>
<td>Source code debugging actions</td>
<td>display_backtrace</td>
<td>Writes the current stack backtrace into the log file.</td>
</tr>
<tr>
<td></td>
<td>[-level level-num]</td>
<td>-level level-num sets the level at which information starts being logged.</td>
</tr>
<tr>
<td></td>
<td>[ num_levels ]</td>
<td>num_levels restricts output to this number of levels in the call stack.</td>
</tr>
<tr>
<td></td>
<td>[ options ]</td>
<td>If you do not set a level, tvscript displays all levels in the call stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>options is one or more of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_arguments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_fp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_fp_registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_locals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_pc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-[no]show_registers</td>
</tr>
</tbody>
</table>
### -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:

<table>
<thead>
<tr>
<th>Type of Specifier</th>
<th>Specifier</th>
<th>Display ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>General display</td>
<td>noshow_fp</td>
<td>Does not show the frame pointer (FP)</td>
</tr>
<tr>
<td></td>
<td>noshow_image</td>
<td>Does not show the process/library in backtrace</td>
</tr>
<tr>
<td></td>
<td>noshow_pc</td>
<td>Does not show the program counter (PC)</td>
</tr>
<tr>
<td></td>
<td>noshow_pid</td>
<td>Does not show the system process ID with process information</td>
</tr>
<tr>
<td></td>
<td>noshow_rank</td>
<td>Does not show the rank of a process, which is shown only for a parallel process</td>
</tr>
<tr>
<td></td>
<td>noshow_tid</td>
<td>Does not show the thread ID with process information</td>
</tr>
</tbody>
</table>

### Action Types

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Action</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse debugging actions</td>
<td>enable_reverse_debugging</td>
<td>Turns on ReplayEngine reverse debugging and begins recording the execution of the program.</td>
</tr>
<tr>
<td></td>
<td>save_replay_recording_file</td>
<td>Saves a ReplayEngine recording file. The filename is of the form <code>&lt;ProcessName&gt;_&lt;PID&gt;_&lt;DATE&gt;_&lt;INDEX&gt;.recording</code>.</td>
</tr>
</tbody>
</table>

- `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 TotalView for HPC User Guide.
Batch Debugging Using tvscript

/mem_debugging

Enables memory debugging and memory event notification. This option is required with any option that begins with \texttt{-mem}. These options are TotalView command line options, as they can be invoked directly by TotalView.

\textbf{-mem\_detect\_leaks}

Performs leak detection before generating memory information.

\textbf{-mem\_detect\_use\_after\_free}

Tests for use after memory is freed.

\textbf{-mem\_guard\_blocks}

Adds guard blocks to an allocated memory block.

\textbf{-mem\_hoard\_freed\_memory}

Holds onto freed memory rather than returning it to the heap.

\textbf{-mem\_hoard\_low\_memory\_threshold nnnn}

Sets the low memory threshold amount. When memory falls below this amount, an event is fired.

\textbf{-mem\_paint\_all}

Paints memory blocks with a bit pattern when a memory is allocated or deallocated.

\textbf{-mem\_paint\_on\_alloc}

Paints memory blocks with a bit pattern when a memory block is allocated.

\textbf{-mem\_paint\_on\_dealloc}

Paints memory blocks with a bit pattern when a memory block is deallocated.

\textbf{-mem\_red\_zones\_overruns}

Turns on testing for Red Zone overruns.

---

<table>
<thead>
<tr>
<th>Type of Specifier</th>
<th>Specifier</th>
<th>Display ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory debugging</td>
<td>noshow_allocator</td>
<td>Does not show the allocator for the address space</td>
</tr>
<tr>
<td></td>
<td>noshow_backtrace</td>
<td>Does not show the backtraces for memory blocks</td>
</tr>
<tr>
<td></td>
<td>noshow_backtrace_id</td>
<td>Does not show the backtrace ID for memory blocks</td>
</tr>
<tr>
<td></td>
<td>noshow_block_address</td>
<td>Does not show the memory block start and end addresses</td>
</tr>
<tr>
<td></td>
<td>noshow_flags</td>
<td>Does not show the memory block flags</td>
</tr>
<tr>
<td></td>
<td>noshow_guard_id</td>
<td>Does not show the guard ID for memory blocks</td>
</tr>
<tr>
<td></td>
<td>noshow_guard_settings</td>
<td>Does not show the guard settings for memory blocks</td>
</tr>
<tr>
<td></td>
<td>noshow_leak_stats</td>
<td>Does not show the leaked memory block statistics</td>
</tr>
<tr>
<td></td>
<td>noshow_owner</td>
<td>Does not show the owner of the allocation</td>
</tr>
<tr>
<td></td>
<td>noshow_red_zones_settings</td>
<td>Does not show the Red Zone entries for allocations (and deallocations) for the address space</td>
</tr>
</tbody>
</table>
-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
  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 264.

```
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:`**

```bash
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 Syntax discussed 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 226 for 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 248 for 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 276.

```plaintext
# Get the process so we have some information about it
tvscript_log "PID: \n    [tvscript_get_process_property 1 "syspid"];
tvscript_log "Status: \n    [tvscript_get_process_property 1 "state"];
tvscript_log "Executable: \n    [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:
```

---

**Batch Debugging Using tvscript / tvscript External Script Files** 278
# 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"
Chapter 5

TotalView Variables

Overview

This chapter contains a list of all CLI and TotalView variables, organized into sections that each correspond to a CLI namespace:

- Top-Level (::) Namespace
- TV:: Namespace
- TV::MEMDEBUG:: Namespace
- TV::GUI:: 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

**ARGS_DEFAULT**
Contains the argument passed to a new process when no ARGS($dpmid$) variable is defined.

*Permitted Values:* A string
*Default:* None

**BARRIER_STOP_ALL**
Contains the value for the “stop_when_done” property for newly created action points. This property 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

**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, TotalView releases the satisfaction set when the barrier is satisfied.

This variable is the same as the TV:barrier_stop_when_done variable.

*Permitted Values*: group, process, or thread

*Default*: group

CGROUP(dpid)

Contains the control group for the process with the TotalView ID dpid. Setting this variable moves process dpid into a different control group. For example, the following command moves process 3 into the same group as process 1:

```
dset CGROUP(3) $CGROUP(1)
```

*Permitted Values*: A number

*Default*: None

COMMAND_EDITING

Enables some Emacs-like commands 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: {}
process
   A user-created process group

share
   The group of processes in one program that share the same executable image

thread
   A user-created thread group

workers
   The group of worker threads in a program

Elements that follow are either pids (for process groups) or pid.tid pairs (for thread groups).

The gid is a simple number for most groups. In contrast, a lockstep group's ID number is of the form pid.tid. Thus, GROUP(2.3) contains the lockstep group for thread 3 in process 2. Note, however, that the CLI 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 TotalView for HPC 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** check-box 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 TotalView for HPC 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_cell_spu_image_load**
If true, TotalView might ask whether to stop the process when a Cell SPU image is loaded. If false, TotalView does not stop execution when a Cell SPU image is loaded.

- **Permitted Values:** true or false
- **Default:** true

**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:** true

**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, TotalView automatically 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 undereferenced 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 > Preferences dialog box.

*Permitted Values:* no, yes, or yes_dont_push

*Default:* no

**TV::auto_deref_in_all_fortran**

Tells TotalView if and how it should dereference Fortran pointers when you perform a Dive in All operation, as follows:

- **yes_dont_push**
  While automatic dereferencing will occur, does not allow use of the Undive command to see the undereferenced value when performing a Dive in All operation.

- **yes**
  Allows use of the Undive control to see undereference 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 > Preferences dialog box.

*Permitted Values:* no, yes, or yes_dont_push

*Default:* no

**TV::auto_deref_initial_c**

Tells TotalView if and how it should dereference C pointers when they are displayed, as follows:

- **yes_dont_push**
  While automatic dereferencing will occur, does not allow use of the Undive command to see the undereferenced 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 undereferenced 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_nested_c**

Tells TotalView if and how it should dereference C pointers when you dive on structure elements:

*yes_dont_push*

While automatic dereferencing will occur, you can't use the Undive command to see the undereferenced value.

*yes*

You will be able to use the Undive control to see undeferenced 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 undereferenced 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 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, TotalView automatically loads action points from the file named filename.TVD.v4breakpoints where filename 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 unnecessary 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.v4breakpoints, 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::bluegene_io_interface**

If the Bluegene front-end cannot resolve the network name, you must initialize this variable (or set it as a command-line option). By default, TotalView assumes that it can resolve the address as follows:
**front_end_hostname-io**

For example, if the front-end hostname is fred, TotalView assumes that the servers are connecting to fred-io.

- **Permitted Values:** A string
- **Default:** none

**TV::bluegene_server_launch_string**

Defines the launch string used when launching tvdsvr processes on I/O nodes.

- **Permitted Values:** A string
- **Default:** -callback %L -set_pw %P -verbosity %V %F

**TV::bluegene_launch_timeout**

Specifies the number of seconds to wait to hear back from the TotalView Debugger Server (tvdsvr) after its launch.

- **Permitted Values:** An integer from 1 to 3600 (1 hour)
- **Default:** 240

**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::cell_spu_image_ignore_regexp
If set to a non-empty string, and TV::ask_on_cell_spu_image_load is true, TotalView matches the SPU image's name with the regular expression. For a match, TotalView does not ask to stop the process but allows the process to continue running after loading the SPU image.

If the image name does not match this regular expression or the regular expression contained within TV::cell_spu_images_stop_regexp, TotalView asks if it should stop the process, unless you've answered the stop to set breakpoint question by pressing No (or the equivalent from within the CLI).

Permitted Values:A regular expression
**TV::cell_spu_images_stop_regexp**

If set to a non-empty string and **TV::ask_on_cell_spu_image_load** is true, TotalView matches the SPU image’s name with the regular expression. For a match, TotalView asks whether to stop the process.

If the image name does not match this regular expression or the regular expression contained within **TV::cell_spu_images_ignore_regexp**, TotalView asks if it should stop the process, unless you've answered the stop to set breakpoint question by pressing No (or the equivalent from within the CLI).

*Permitted Values:* A regular expression

*Default:* \{

---

**TV::cell_spurs_jm_name**

A string that names the file containing the symbols for the “jm” SPURS job policy module. When TotalView detects an embedded SPURS kernel image being loaded into an SPU context, it extracts the GUIDs of the policy modules from the kernel, and searches for either the default SPU ELF image file, which is spurs_jm.elf for the file named by this variable.

*Permitted Values:* An ELF file name

*Default:* spurs_jm.elf

---

**TV::cell_spurs_kernel_dll_regexp**

Defines a regular expression that matches the image path component name of the SPURS kernel SPU ELF image embedded in the libspurs.so DLL.

When TotalView sees a new image loaded into an SPU thread by libspe or libspe2, it checks if the image path component name matches this variable. If so, TotalView handles the SPURS kernel library in a different way. You may need to change this regular expression to match the name of your SPURS kernel if it is embedded in a shared library other than libspurs.so or if the name of the SPURS kernel is different than spurs_kernel.elf.

*Permitted Values:* A regular expression

*Default:* \{libspurs.so\(spurs_kernel\elf@[0-9]+\)$

---

**TV::cell_spurs_ss_name**

A string that names the file containing the symbols for the “ss” SPURS system service policy module. When TotalView detects an embedded SPURS kernel image being loaded into an SPU context, it extracts the GUIDs of the policy modules from the kernel, and searches for either the default SPU ELF image file, which is spurs_tss.elf for the file named by this variable.

*Permitted Values:* An ELF file name

*Default:* spurs_ss.elf
**TV::cell_spurs_tm_name**

A string that names the file containing the symbols for the “tm” SPURS task policy module. When TotalView detects an embedded SPURS kernel image being loaded into an SPU context, it extracts the GUIDs of the policy modules from the kernel, and searches for either the default SPU ELF image file, which is `spurs_tm.elf` for the file named by this variable.

*Permitted Values:* An ELF file name  
*Default:* `spurs_tm.elf`

**TV::checksum_libraries**

*Permitted Values:*  
*Default:* `auto`

**TV::comline_patch_area_base**

Allocates the patch space dynamically at the given *address*. See “Allocating Patch Space for Compiled Expressions” in the *TotalView for HPC Users Guide*.

*Permitted Values:* A hexadecimal value indicating space accessible to TotalView  
*Default:* `0xffffffffffffffff`

**TV::comline_patch_area_length**

Sets the length of the dynamically allocated patch space to the specified *length*. See “Allocating Patch Space for Compiled Expressions” in the *TotalView for HPC Users 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, TotalView enables 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 for HPC Release Notes to determine if your version of the operating system supports this feature.

Permitted Values: true or false
Default: false

TV::control_c_quick_shutdown
When true, TotalView kills 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 352.

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
**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 thefork(), vfork(), and execve() system calls if your executable is linked with the dbfork library. For information on linking with dbfork, see “Linking with the dbfork Library” in the NextGen TotalView for HPC User Guide.

*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:* Sun SPARC: rsh All other platforms: 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_stdout_is_stdout**

When `true`, TotalView writes the target program's `stdout` 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 TotalView loads 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 backtraces may be truncated. However, if your program uses large shared libraries and it’s time consuming to read even their loader symbols, you may want to put those libraries on this list.

Permitted Values: One or more library names separated by colons
Default: None

TV::dll_stop_suffix

Contains a colon-separated list of suffixes that 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 false, enables dlopen event filtering.

TV::dlopen_always_recalculate is true by default, meaning that breakpoint specifications are reevaluated on every dlopen call. This is referred to as Slow Mode.
A value of `false` enables `dlopen` event filtering, deferring the reevaluation of breakpoint specifications until after the `dlopen` event and thus reducing the number of events per process that TotalView evaluates. This is useful in improving performance when a process loads large numbers of libraries. Depending on the setting of `TV::dlopen_recalculate_on_match`, performance can be improved with the `Medium` or `Fast` modes of `dlopen` event filtering.

**Permitted Values:** true or false  
**Default:** true

**TV::dlopen_recalculate_on_match**

Contains a colon-separated list of simple glob patterns (a glob list) containing library names. If `TV::dlopen_always_recalculate` is set to `true`, the value of this variable is ignored.

**glob patterns** specify sets of filenames with wildcard characters. A simple glob pattern is a string, optionally ending with an asterisk character (*).

If `TV::dlopen_always_recalculate` is `false` and a `dlopen` event occurs, the name of the library associated with the event is matched against the list of glob patterns. If the glob-list is empty (default) or the name of the dlopened library does not match any patterns in the glob-list, then breakpoint reevaluation is deferred until the process stops for some other reason (e.g., the process hits a breakpoint, the user stops the process, the process encounters a signal, etc.). If the library name matches a pattern, the breakpoints are reevaluated immediately. A glob-list that contains the empty string results in Fast mode, since all the dlopened libraries will have their breakpoint reevaluation deferred. Medium mode is when select libraries are to have their breakpoints reevaluated immediately.

The matching rules are:

- If the simple glob pattern does not end in an asterisk, then the tail of the loaded library name must match the string. For example, the string "libfoo.so" matches library name "/dir/path/libfoo.so", but does not match "/dir/path/libfoo.so.1.0".

- If the simple glob pattern ends in an asterisk, then the asterisk is removed from the string, and the remaining portion of the string matches any substring found in the library name. For example, the string "libfoo.so*" matches "/dir/path/libfoo.so" or "/dir/path/libfoo.so.1.0", and the string "/path/*" matches "/dir/path/libfoo.so" or "/dir/path/libbar.so".

**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” in the TotalView for HPC User Guide.

**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
TotalView Variables / TV:: Namespace

- 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 235.

**Permitted Values:** One or more of the arguments listed above.

**Default:** `class_name+template_args+module+parent_function+member+node`

---

**TV::dynamic**

When `true`, TotalView loads symbols from shared libraries. This variable is available on all platforms supported by Rogue Wave 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
**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 371.

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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Stop the process</td>
</tr>
<tr>
<td>go</td>
<td>Continue the process</td>
</tr>
<tr>
<td>ask</td>
<td>Ask whether to stop the process</td>
</tr>
</tbody>
</table>

**Permitted Values:** exec-handling-list

**Default:** None

**TV::fne_reservation_group_name**

Defines the reservation group name for the FlexNet Embedded Server. If the user belongs to this group, TotalView sets the reservation group name as a secondary hostID when requesting a license from the FNE server. The license administrator can control license reservations using this group when setting up the FNE license server.

**Permitted Values:** String

**Default:** totalview_user

**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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach</td>
<td>Attach to the new child processes.</td>
</tr>
<tr>
<td>detach</td>
<td>Detach from the new child processes.</td>
</tr>
</tbody>
</table>
**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 Next-Gen TotalView for HPC 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](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
```

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 NextGen TotalView for HPC User Guide.

**Permitted Values:** A colon-separated build ID search path

**Default:** %G

**TV::gnu_debuglink_checksum**

When **true**, TotalView compares the checksum of the debug file against the checksum contained within the **.gnu_debuglink** section. TotalView loads the information from the debug file when the checksums match or when this variable is set to **false**. Set this to **false** only if you are absolutely certain that the debug file matches.

For more information, see the appendix section “Using gnu_debuglink Files” in the NextGen TotalView for HPC User Guide.

**Permitted Values:** true or false

**Default:** true

**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 NextGen TotalView for HPC 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 *NextGen TotalView for HPC 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:

```tcl
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 `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-Power, Blue Gene/Q, 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::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 .tvdr file:

```bash
    dset TV::open_cli_window_callback "stty sane"
```

*Permitted Values:* A string representing a Tcl or CLI command

*Default:* Null

**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::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 TotalView for HPC.

  • **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::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_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: _suffix:lang[:include]_. You can set more than suffix for an item. If you want to remove an item from the default list, set its value to unknown.

_Permitted Values:_ A list identifying how suffixes are used

_Default:_ 

```text
{: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 HP:c++:include HH:c++:include H:c++:include ih:c++:include th:c++}
```

**TV::target_platform**

A read-only variable that displays a list of the platforms on which you can debug from the native (host) platform, usually in the format _os-cpu_. For example, from a native platform of Linux-x86, the list is “linux-power linux-x86_64 linux-x86.” The platform names may be listed differently than in _TV::platform_ and _TV::native_platform_. For example, for AIX, _TV::target_platform_ is “aix-power” but _TV::platform_ and _TV::native_platform_ are “rs6000.”

_Permitted Values:_ A list of platform names

_Default:_ Platform-dependent

**TV::ttf**

When 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
**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 *NextGen TotalView for HPC 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 *NextGen TotalView for HPC 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`
**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, TotalView will continue searching from either the beginning (if Down is also selected in the Edit > Find dialog box) or the end (if Up is also selected) if it doesn't find what you're looking for. For example, you search for foo and select the Down button. If TotalView doesn't find it in the text between the current position and the end of the file, TotalView will continue searching from the beginning of the file if you set this option.

*Permitted Values:* true or false

*Default:* true

**TV::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 TotalView for HPC 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::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*.

```tcl
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:

```tcl
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
**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 block 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-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::geometryGlobals**

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}
**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 **VariableWindow.**

 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 TotalView for HPC 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, it sets that check box to be deselected by default.

*Permitted Values:* true or false  
*Default:* false

**TV::GUI::pop_on_error**

When `true`, TotalView sets the Open process window on error signal check box in the File > Preferences's Option Page to be selected by default. If you set this to false, TotalView sets that check box to be deselected by default.

*Permitted Values:* true or false  
*Default:* true

**TV::GUI::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
This part of the *NextGen TotalView for HPC Reference Guide* discusses formatting and transformations that display data in a clear and concise format to facilitate easier debugging sessions.

**Chapter 6, “Creating Type Transformations,” on page 341**

Discusses how to customize data display using CLI routines. This is useful if you do not wish to see all the members of a class or structure or would like to alter the way TotalView displays these elements.
Chapter 6

Creating Type Transformations

Overview

The Type Transformation Facility (TTF) lets you define the way TotalView displays aggregate data. Aggregate data is simply a collection of data elements from within one class or structure. These elements can also be other aggregated elements. In most cases, you will create transformations that model data that your program stores in an array- or list-like way. You can also transform arrays of structures.

This chapter describes the TTF. It presents information 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.

Topics in this chapter are:

- “Why Type Transformations” on page 342
- “Creating Structure and Class Transformations” on page 344
- “C++View” on page 352
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

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” on page 344
- “build_struct_transform Function” on page 346
- “Type Transformation Expressions” on page 346
- “Using Type Transformations” on page 350

Transforming Structures

The following small program contains a structure and the statements necessary to initialize it:

```c
#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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>int</td>
<td>0x0000007d4 (2004)</td>
</tr>
<tr>
<td>pName</td>
<td>$string</td>
<td>&quot;John Smith&quot;</td>
</tr>
<tr>
<td>pStreet</td>
<td>$string</td>
<td>&quot;24 Prime Parkway, Suite&quot;</td>
</tr>
</tbody>
</table>
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:

```tcl
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:

```tcl
{ 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:

```tcl
{ 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.
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.
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” on page 352
- “Templates” on page 355
- “Precedence - Searching for TV_ttf_display_type” on page 356
- “TV_ttf_add_row” on page 356
- “Return values from TV_ttf_display_type” on page 357
- “Elision” on page 358
- “Other Constraints” on page 358
- “Safety” on page 359
- “Memory Management” on page 360
- “Multithreading” on page 360
- “Tips and Tricks” on page 361
- “Core Files” on page 361
- “Using C++View with ReplayEngine” on page 361
- “C” on page 363
- “Compiling and linking tv_data_display.c” on page 364
- “C++View Example Files” on page 364
- “Limitations” on page 365
- “Licensing” on page 365

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:

```c++
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:

```c++
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:

```c++
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 dis-
cuss 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:

```cpp
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:

```cpp
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 361.)

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:

```cpp
int TV_ttf_display_type ( const BoundsCheckedArray<char> *s )
{
    TV_ttf_add_row ( "string", TV_ttf_type_ascii_string, s->get_array () );
}
```
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_asci_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:

```cpp
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 361 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 BoundsCheckedArray<char> class and the specialized TV_ttf_display_type function we defined earlier:

```c
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:

```c
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:

```c
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 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` 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:

```cpp
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_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 “ReplayEngine” in the NextGen TotalView for HPC 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.

```cpp
#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:

```bash
setenv TVSOURCE /usr/local/toolworks/totalview2016.01/src
setenv TVINCLUDE /usr/local/toolworks/totalview2016.01/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_ad-d_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 Re-playEngine.

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.
This part of the NextGen TotalView for HPC Reference Guide contains information about command-line options used when starting TotalView.

Chapter 7, “TotalView Command Syntax,” on page 367
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.
Overview

This chapter describes the syntax of the totalview command. Topics in this chapter are:

• Command-Line Syntax
• Command-Line Options
**Command-Line Syntax**

**Format**

totalview -newUI[options ] [ executable [ core-file | recording-file ]] [-a[args ] ]

or

totalview -newUI [options ] -args executable [args ]

---

**NOTE >>** If you have set the environment variable TVNEWUI to True, omit the flag -newUI when starting TotalView.

---

**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 multiprocess 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**

NextGen TotalView for HPC 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*

- **-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.

- **-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 TotalView console output to a file named `console_outputfile`.
  
  If `consoleoutputfile` is the string `UNIQUE`, the filename `tv_dump.hostname.pid` is used. If `consoleoutputfile` 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 dlopen 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 reevaluation of breakpoint specifications until after the dlopen event. The point at which the breakpoint specifications are reevaluated depends on the value of the TV::dlopen_recalculate_on_match variable (see -dlopen_recalculate_on_match glob-list).

dlopen_recalculate_on_match glob-list

Default: “” (the empty string)

This option’s argument is a colon-separated list of simple glob patterns used to compare and match the dlopened library. A simple glob pattern is a string, optionally ending with asterisk character (‘*’). For information on the semantics of glob pattern matching, see TV::dlopen_recalculate_on_match.

Used with -no_dlopen_always_recalculate, when a dlopen event occurs, the name of the dlopened library is matched against the list of glob patterns; if the glob-list is empty (the default) or the name of the dlopened library does not match the glob-list, then breakpoint reevaluation is deferred until the process stops for some other reason (e.g., the process hits a breakpoint, the user stops the process, the process encounters a signal, etc.). If the library name matches a pattern, the breakpoints are reevaluated immediately.

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 
  (Default). Disables handling *dlopen* events in parallel.

-dump_core
  Allows TotalView to dump a core file of itself when an internal error occurs. This is used to help Rogue Wave Soft-
  ware debug problems.

-dwarf_global_index
  (Default). Allows TotalView to use the DWARF global index sections (.debug_pubnames, .debug_pubtypes, .debug_ty
  .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 in

-e commands
  Immediately executes the CLI commands named within this argument. All information you enter here is sent di-
  rectly to the CLIs 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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Stop the process</td>
</tr>
<tr>
<td>go</td>
<td>Continue the process</td>
</tr>
<tr>
<td>ask</td>
<td>Ask whether to stop the process</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach</td>
<td>Attach to the new child processes.</td>
</tr>
<tr>
<td>detach</td>
<td>Detach from the new child processes.</td>
</tr>
</tbody>
</table>

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::dbfork` CLI 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 (\texttt{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.

\textbf{-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.

\textbf{-gnu\_debuglink}

For a program or library with either (or both) a build ID or \texttt{.gnu\_debug\_link} section, TotalView looks for a separate debug file. If found, TotalView reads this file's debugging information.

\textbf{-no\_gnu\_debuglink}

Do not load information from a separate debug file even if the file has a build ID or \texttt{.gnu\_debug\_link} section.

\textbf{-gnu\_debuglink\_build\_id\_search\_path}

Sets the \texttt{TV::gnu\_debuglink\_build\_id\_search\_path} variable to specify a build ID search path string.

\textbf{-gnu\_debuglink\_checksum}

Validates the debug file's checksum against the checksum contained in the image's \texttt{.gnu\_debuglink} section.

\textbf{-no\_gnu\_debuglink\_checksum}

Do not compare checksums. Set this only if you are absolutely certain that the debug file matches.

\textbf{-gnu\_debuglink\_global\_directory}

Sets the \texttt{TV::gnu\_debuglink\_global\_directory} variable that names the global directory that stores debug files.

\textbf{-gnu\_debuglink\_search\_path}

Sets the \texttt{TV::gnu\_debuglink\_search\_path} variable to specify a search path string.

\textbf{-ipv6\_support}

Directs TotalView to support IPv6 addresses.

\textbf{-no\_ipv6\_support}

(Default) Do not support IPv6 addresses.

\textbf{-jit\_debugging}

Enables Clang / LLVM JIT debugging. See \texttt{TV::jit\_debugging} for details.

\textbf{-no\_jit\_debugging}

Disables Clang / LLVM JIT debugging.

\textbf{-kcc\_classes}

(Default) Converts structure definitions output by the KCC compiler into classes that show base classes and virtual base classes in the same way as other C++ compilers. See the description of the \texttt{TV::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.)

-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.

-patch_area_base address
   Allocates the patch space dynamically at address. See “Allocating Patch Space for Compiled Expressions” in the TotalView for HPC 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 TotalView for HPC 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.

-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 TotalView for HPC Users 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 -newUI -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 signalSpecifier 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.

-teamlplus
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 NextGen TotalView for HPC Release Notes on the Rogue Wave website 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 NextGen TotalView for HPC Release Notes on the Rogue Wave website 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. (Note: This option does not apply to -BlueGene/Q systems; instead, see -control_c_quick_shutdown-ccq.)

-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
-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.
Symbols
.totalview/lib_cache subdirectory 45
tvd files 234
@ symbol for action point 117
* expr 346
# scoping separator character 116
= symbol for PC of current buried stack frame 117
> symbol for PC 117
$newval variable in watchpoints 185
$oldval variable in watchpoints 185
$stop function 74, 135

A
-a option to totalview command 369
ac, see dactions command
acquiring processes 304
action point identifiers 23
Action Point > Save All command 294
action point identifiers 69
action points
autoloading 294
default for newly created 287
default property 288
deleting 59, 209, 223, 226, 251
disabling 23, 24, 62
displaying 23
enabling 23, 24
identifiers 24
information about 23, 24, 202
loading 24
loading automatically 374
loading saved information 25
reenabling 69
saving 24
saving information about 25
scope of what is stopped 288
setting at location 23
sharing 287
stopping when reached 325
actionpoint properties 209
actionpoint command 209
actions points
list of disabled (ambiguous context) 24
list of enabled (ambiguous context) 24
actions, see dactions command
activating type transformations 351
adding group members 85
adding groups 84
address 235
address property 209
addressing_callback 255
advancing by steps 169
aggregate data 341
aix_use_fast_trap variable 291, 327
alias command 19
aliases
default 19
removing 206
append, see dlappend command
appending to CLI variable lists 115
as, see dassign command
attaching to parallel processes 33
attaching to processes 33
attaching to ranks 34
attaching, using PIDs 34
auto_array_cast_bounds variable 291
auto_array_cast_enabled variable 291
auto_deref_in_all_c variable 292
auto_deref_in_all_fortran variable 292
auto_deref_initial_c variable 292
auto_deref_initial_fortran variable 293
auto_deref_nested_c variable 293
auto_deref_nested_fortran variable 293
arrays
gathering statistical data 134
number of elements displayed 291
automatic dereferencing 291
automatic dereferencing of arrays 291
arriving at barrier 37
ask_on_dlopen variable 291
assemble, displaying symbolically 306
asynchronous execution 74
at, see dattach command
attach, see dattach command
attaching to parallel processes 33
attaching to processes 33
attaching to ranks 34
attaching, using PIDs 34
auto_read_symbols_at_stop variable 294
auto_cast_breakpoints variable 294
auto_load_breakpoints variable 294
auto_read_symbols_at_stop variable 294
auto_save_breakpoints variable 294
automatic dereferencing 292
automatic dereferencing of arrays 291
automatically attaching to
processes 320

B
b, see dbreak command
ba, see dbarrier command
background command-line option 369
back-tick analogy 21
barrier breakpoint 37
barrier is satisfied 283, 295
BARRIER_STOP_ALL variable 36, 38, 282
barrier_stop_all variable 295
BARRIER_STOP_WHEN_DONE variable 36, 38, 282
barrier_stop_when_done variable 295
barrier, see dbarrier command
barriers 36, 37
arriving 37
creating 37
what else is stopped 36
base_name 236
Batch debugging, tvscript 264
baw, see dbarrier command
bg command-line option 369
blocking command input 184
blocking input 184
Bluegene I/O interface 295
Bluegene launch string 296
Bluegene server timeout 296
bluegene_io_interface variable 295
bluegene_launch_string variable 296
bluegene_server_launch_timeout variable 296
break, see dbreak command
breakpoints
automatically loading 294
barrier 36
default file in which set 42
defined 42
document 25, 294
popping Process Window 337
setting at functions 42
setting expression 41
stopping all processes at 41
temporary 179
triggering 42
bt, see dbreak command
build_struct_transform
defined 346
example 345
lists 346
members argument 346
name argument 346
bulk_launch_base_timeout variable 296
bulk_launch_enabled variable 296
bulk_launch Incre_time-out variable 296
bulk_launch_tmpfile1_header_line variable 296
bulk_launch_tmpfile1_host_lines variable 297
bulk_launch_tmpfile1_trailer_line variable 297
bulk_launch_tmpfile2_header_line variable 297
bulk_launch_tmpfile2_host_lines variable 297
bulk_launch_tmpfile2_trailer_line variable 297
buried stack frame 116
by_language_rules 232
by_path 232
by_type_index 232
C
C language
escape characters 31
using with C++View 363
c_type_strings 297
c_type_strings variable 297
C++
demangler 301, 369
STL instantiation 342
C++View
and multithread safety 360
and programming

design 359
and template classes 355
compiling and linking 364
described 352
example display function 353
example of use with
ReplayEngine 362
examples 364
use of with
ReplayEngine 361
using with C 363
cache, flushing 45
cache, Memory Debugger 331
cache, see dcache command
call stack 182
displaying 160, 193
see also stack frame 10, 12, 159, 192
call tree saved position 334
callback list 318
callbacks 316
after loading a program 321
when opening the CLI 320
capture command 21, 198
case sensitive searching 322
cast to array with bounds checkbox 291
casting variables 138
ccq command-line option 369
CGROUP variable 283
changing CLI variables 150, 151
changing dynamic context 182
changing focus 77
changing value of program variable 31, 50, 112, 131, 145, 176
chase_mouse variable 332
checking interior pointers 332
class transformations 344
classes, transforming 346
CLI
activated from GUI flag 332
sourcing files 345
startup file 345
CLI variables
aix_use_fast_trap 291, 327
ARGS 282
ARGS_DEFAULT 282
ask_on_dlopen 291
auto_array_cast_bounds 29
in_setup 316
kcc_classes 317
keep_expressions 336
keep_search_dialog 337
kernel_launch_string 318
kill_callbacks 318
leak_check_interior_pointers 330, 331
leak_max_cache 331
leak_max_chunk 331
library_cache_directory 318
LINES_PER_SCREEN 285
local_interface 318
local_server 319
local_server_launch_string 3
MAX_LIST 116, 286
message_queue 319
nptl_threads 320
OBJECT_SEARCH_MAPPING 284, 286
OBJECT_SEARCH_PATH 286
open_cli_callback 320
parallel 320
parallel_attach 320
parallel_stop 320
platform 321
pop_at_breakpoint 337
pop_on_error 337
process_load_callbacks 321
PROMPT 286
PTSET 286
restart_threshold 322
save_window_pipe_or_filename 322
search_case_sensitive 322
server_launch_enabled 323
server_launch_timeout 323
server_response_wait_timeout 323
SGROUP 287
SHARE_ACTION_POINT 287
share_action_point 323
shared_data_filters 331
show_startup_parameters 338
show_sys_thread_id 338
signal_handling_mode 323
single_click_dive_enabled 3
SOURCE_SEARCH_MAPPING 287
SOURCE_SEARCH_PATH 287
spell_correction 325
stack_trace_qualification_level 325
STOP_ALL 41, 288
stop_all 325
stop_relatives_on_proc_error 326
suffix 326
TAB_WIDTH 117, 288
THREADS 288
toolbar_style 338
tooltips_enabled 338
TOTAL_VERSION 289
TOTALVIEW_ROOT_PATH 28
TOTALVIEW_TCLLIB_PATH 2

command arguments 282
command focus 77
command input, blocking 184
command line arguments 148
command output 21
command prompt 286
command summary 2
command verb
  actionpoint command 214
COMMAND_EDITING variable 283
command_editing_variable 299
command_line_options
  -background 369
  -bg 369
  -ccq 369
  -control_c_quick_shutdown
  -dbfork 369
  -debug_file 369
  -demangler 369
  -display 370
  -dll_ignore_prefix 370
  -dll_stop_suffix 370
  -dump_core 371
  -dwarf_global_index 371
  -e 371
  -ent 371
  -env 371
  -fg 372
  -foreground 372
  -gdb_index 372
  -global_types 372
  -gnu_debuglink 373
  -gnu_debuglink_checksum
  -ipv6_support 373
  -lb 374
  -nccq 369
  -nlb 374
  -no_control_c_quick_shutdown
  -no_dbfork 369
  -no_dwarf_global_index 371
  -no_ent 371
  -source_pane_tab_width 324
commands responding to 231
  totalview 367
  user-defined 19
  commands verb
    actionpoint command 209
    expr command 218
    group command 223
    process command 226
    thread command 248
    type command 251
  compile_expressions
    variable 300
  compiler property 255
  compilers, KCC 317
  compiling

for C++View 364
  compressed list of processes, defined. See plist.
  Compressed List Syntax, defined 166
  conditional watchpoints 185
  console output redirection 369
  cont, see dcont command
  continue_sig property 248
  continuing execution 82
  control group variable 283
  control group, stopping 326
  control list element 284
  control_c_quick_shutdown command-line option 369
  control_c_quick_shutdown variable 300
  copyright_string variable 300
  core
    dumping for TotalView 371
    when needing to debug TotalView itself 309
  core files, loading 34
  count property 224
  create subcommand 254
  creating barrier breakpoints 37
  creating commands 19
  creating groups 84
  creating new process objects 120
  creating threads 82
  creating type transformations 341
  Ctrl+C, ignoring 316
  Ctrl+D to exit CLI 197, 199
  CUDA
    CLI command detail 54
    dcuda > block command 54
    dcuda CLI command 54
    support for GPU threads 18, 54
  current list location 67
  current_cplus_demangler variable 301
  current_fortran_demangler variable 301

D
d, see ddown command
dactions command 24
dassign command 31
data format
  presentation styles 301
  transforming with C++View 352
  data representation
    simplifying using elision 358
  data size 122, 157
  data_format_double
    variable 301
  data_format_ext variable 303
  data_format_int16 variable 303
  data_format_int32 variable 303
  data_format_int64 variable 303
  data_format_int8 variable 303
  data_format_single variable 304
  data_format_stringlen
    variable 304
data_type cast expr 347
data_type incompatibilities 31
dattach command 33
dbfork command-line option 369
dbfork library
  syntax 369
dbfork variable 304
dbreak command 41
  setting expression in 41
dcache command 45
dcalltree command 46
dcont command 52
dcuda command 54
ddelete command 59
ddetach command 60
ddisable command 62
ddl_read_all_symbols
  variable 306
ddlopen command 64
ddown command 67
de, see ddelete command
deactivating action points 62
deadlocks at barriers 38
debug_file command-line option 369
debugging remote systems 45
debugging session, ending 197
dec2hex command 213
default aliases 19
default arguments 148, 282
   modifying 148
default preferences, setting 152
default value of variables, restoring 178
default_snippet_extent 330
default_stderr_append variable 304, 305
default_stderr_filename variable 305
default_stdin_filename variable 305
default_stdout_append variable 305
default_stdout_filename variable 305
default_stderr_is_stdout variable 305
deferred reading, shared library symbols 230
defining the current focus 286
delete verb, expr command 218
delete, see ddetach command
detaching from processes 60
dec2hex command 71
dflush command 74, 135
dfocus command 77
dga command 80
dgo command 82
dgroups command 84
   -add 85
   -delete 85
dhalt command 89
dheap command 90
  cheap command, see also heap debugging
dhold command 112
di, see ddisable command
directory search paths 284
disable, see ddisable command
disabled action points list (ambiguous context) 24
disabling action points 23, 24, 62
display call stack 160, 193
display command-line option 370
display_assembler_symbolically variable 306
display_bytes_kb_mb variable 332
display_font_dpi variable 332
displaying
code 116
   current execution location 160, 193
   error message information 289
   help information 198
   information on a name 188
   lines 286
   values 134
displaying expressions 134
displaying memory 71
displaying memory values 71
diving, single click 338
dkill command 114
dlappend command 115
dlist command 116, 286
dlist, number of lines displayed 286
dll command 214
dll_ignore_prefix command-line option 370
dll_read_loader_symbols_only variable 306
dll_read_prefix variable 307
dll_stop_suffix command-line command-line option 370
dll_stop_suffix variable 307, 308
dload command 119
dlopen 65
  ask when loading 291
dmstat command 122
dnext command 125
dnexti command 128
done property 218
double-precision data format 301
dout command 131
down, see ddown command
dpid 283
dpid property 249
dprint command 134
dptsets command 140
drerun - drun differences 144
drerun command 143
drestart command 145
drun - drerun differences 144
drun command 114, 147
  poe issues 144, 148
  reissuing 148
dsession 150
dset command 150, 151
dstatus command 164
dstep command 169
  iterating over focus 169
dstepi command 173
duhtp, see dunhold command
duid property 227, 249
dump subcommand 232, 235
dump_core command-line option 371
dump_core variable 309
dunhold command 176
dunset command 178
duntil command 179
group operations 180
dup command 182
dwait command 184
dwarf_global_index command-line option 371
dwarf_global_index variable 309
dwatch command 185
dwhat command 188
dwhere command 10, 159, 192
dwhere_level 285
dwhere_qualification_level variable 309
dworker command 196
dynamic variable 310

e command-line option 371
editor_launch_string variable 310
eliminating tab processing 117
Emacs-like commands 299
en, see denable command enables, see denable command enabled action points list (ambiguous context) 24
enabled action points 23, 24, 69
ending debugging session 197
ent command-line option 371
enum_values property 251
env command-line option 371
env variable 311
error message information 289
error state 289
errorCodes command 135, 216
errorCodes variable 216
errors, raising 216
escape characters 31
evaluating functions 134
evaluation points, set dbreak evaluations, suspended, flushing 74
examining memory 71, 72
using an expression 72
exception subcodes 135
exception, warning when thrown 328
exec_handling variable 311
executable property 227
EXECUTABLE_PATH variable 35, 117, 284
EXECUTABLE_SEARCH_PATH variable 284
executing as one instruction 128
executing as one statement 125
executing assembler instructions 173
executing source lines 169
execution
continuing 82
displaying location 160, 193
halting 89
resuming 52
execve()
calling 369
catching 304
exit command 197
expr . expr 347
expr -> expr 347
expr command 135, 218
expression property 210, 218
expression values, printing 134
expressions in breakpoint 41
expressions, compiling 300
expressions, type transformation 346
extensions for file names 326

F
f, see dfocus command
fast_trap, setting 291, 327
fg command-line option 372
File > Preferences command 351
file name extensions 326
files
initialization 345
filter definition file 331
filtering the stack
variable to enable 288
filters, sharing memory
filters 331
fixed_font variable 332
fixed_font_family variable 333
fixed_font_size variable 333
floating point data format
double-precision 301
extended floating point 303
single-precision 304
flush, see dflush command
flushing cache 45
flushing suspended evaluations 74
fne_reservation_group_name variable 311
focus
see also dfocus command
defining 286
temporarily changing 77
focus_groups command 220
focus_processes command 221
focus_threads command 222
focus_threads property 218
follow_clone variable 311
font variable 333
fonts 332
fixed 332, 333
ui 333, 338
ui font family 338
ui font size 339
force_default_cplus_demangler variable 312
force_default_f9x_demangler variable 312
force_window_position variable 333
foreground command-line
option 372
fork_handling variable 312
fork() calling 369
catching 304
formatting program data using C++View 352
Fortran demangler 301
frame_offset_x variable 333
frame_offset_y variable 333
functions evaluating 134
setting breakpoints at 42

G
g, see dgo command
gdb_index command-line option 372
gdb_index variable 313, 316
GDP SPU Runtime System (SPU). See spurs.
geometry_call_tree variable 334
geometry_cli position 334
geometry_cli variable 334
gometry_globals variable 334
gometry_help variable 334
gometry_memory_stats variable 335
gometry_message_queue variable 335
gometry_message_queue_graph variable 335
gometry_process variable 335
gometry_ptset variable 335
gometry_root variable 335
gometry_thread_objects variable 336
gometry_variable variable 336
gometry_variable_stats variable 336
get subcommand 232
get verb
  actionpoint command 209, 214
  expr command 218
  group command 223
  process command 226
thread command 248
type command 251
Global Arrays 80
setting language for display 80
global_typenames variable 313
global_types, command-line option 372
GNU C++ STL instantiation 342
gnu_debuglink command-line option 373
gnu_debuglink variable 313, 314
gnu_debuglink_checksum command-line option 373
gnu_debuglink_checksum variable 314
gnu_debuglink_global_directory variable 314
gnu_debuglink_search_path variable 315
go, see dgo command
goal breakpoint 170
GPU threads. See CUDA.
gr, see dgroups command
group command 223
group ID 289
group members, stopping flag 288
group of interest 170
GROUP variable 284
group width stepping behavior 169
groups
  accessing properties 223
  adding 84
  adding members 85
  creating 84
  deleting 84, 85
  intersecting 84
  listing 10, 84, 154
  naming 85
  placing processes in 35
  removing members 10, 84, 154
  returning list of 220
  setting properties 223
GROUPS variable 119, 285
groups, see dgroups command

H
h, see dhalt command
halt, see dhalt command
halting execution 89
handling signals 375
handling user-level (M:N) thread packages 327
heap size 122, 157
heap_size property 227
held property 227, 249
help command 198
help window position 334
hex2dec command 225
hexadecimal conversion 213
hia_allow_ibm_poe 330
hiarc file 330
hold, see dhold command
holding processes 112
holding threads 37, 112
hostname property 227
hp, see dhold command
ht, see dhold command
htp, see dhold command

I
I/O redirection 147
id property 210, 219, 224, 227, 249, 251, 255
ignore_control_c variable 316
ignore_snippets false 330
ignoring libraries by prefix 370
image browser window position 334
Image information 122
image_id property 251
image_ids property 227
image_load_callbacks list 321
image_load_callbacks variable 316
in_setup variable 316
IndexerTEST 368
inet interface name 318, 374
infinite loop, see loop, infinite
info state 289
information on a name 188
initialization file 206, 345
initially_suspended_process property 219
input, blocking 184
inserting working threads 196
instructions, stepping 173
integer (64-bit) data format 303
integer data format
  16-bit 303
  32-bit 303
  8-bit 303
interface name for server 318, 374
interior pointers, checking 330
intersecting groups 84
ipv6_support command-line option 373
ipv6_support option 373
K
k, see dkill command
-kcc_classes command-line option 373
kcc_classes command-line option 373
kcc_classes variable 317
keep_expressions variable 336
keep_search_dialog variable 337
kernel_launch_string variable 318
kill_callbacks variable 318
kill, see dkill command
killing attached processes 300
killing processes 114
L
l, see dlst command
language property 210, 252, 255
languages
  C, using with C++View 363
lappend, see dlappend command
launch string
  for editor 310
  for server (Sun only) 319
  for Visualizer 328
Launched Strings page 328
launching
  local server 319
  processes 147
  Visualizer 327
lb command-line option 374
leak, check_interior_pointers property 33
0, 331
leak_check_interior_pointers variable 330, 331
leak_max_cache 331
leak_max_chunk 331
length property 210, 252
levels for dwhere 285
levels, moving down 67
libraries
dbfork 369
  ignoring by prefix 370
  loading by suffix 307
  loading symbols from shared libraries 214
  symbols from shared libraries 310
tvd files 234
loading sessions
dsession 150
local_interface variable 318
local_server variable 319
local_server_launch_string variable 319
lockstep list element 284
logical model 343
lookup subcommand 232
lookup_keys subcommand 223
loop, infinite, see infinite loop
lower_bounds_callback 256
M
machine instructions, stepping 173
manager property 249
manager threads, running 169
managing shared libraries 214
mangler, overriding 369
mappings, search path 287
MAX_LIST variable 116, 286
Maximum permissible rank field 328
mem_detect_leaks memory sub-option 273
tmem_guard_blocks memory sub-option 273
list, see dlst command
listing groups 10, 84, 154
using a regular expression 86
listing lines 286
lo, see dlload command
load, see dlload command
loader_name 236
loading
  action point information 25
  action points 24, 374
  libraries 306, 307
  programs 120
  shared libraries 214
  symbols from shared libraries 310
tvd files 234
loading sessions
dsession 150
local_interface variable 318
local_server variable 319
local_server_launch_string variable 319
lockstep list element 284
logical model 343
lookup subcommand 232
lookup_keys subcommand 223
loop, infinite, see infinite loop
lower_bounds_callback 256
M
machine instructions, stepping 173
manager property 249
manager threads, running 169
managing shared libraries 214
mangler, overriding 369
mappings, search path 287
MAX_LIST variable 116, 286
Maximum permissible rank field 328
mem_detect_leaks memory sub-option 273
tmem_guard_blocks memory sub-option 273
list, see dlst command
listing groups 10, 84, 154
using a regular expression 86
listing lines 286
lo, see dlload command
load, see dlload command
loader_name 236
loading
  action point information 25
  action points 24, 374
  libraries 306, 307
  programs 120
  shared libraries 214
  symbols from shared libraries 310
tvd files 234
loading sessions
dsession 150
local_interface variable 318
local_server variable 319
local_server_launch_string variable 319
lockstep list element 284
logical model 343
lookup subcommand 232
lookup_keys subcommand 223
loop, infinite, see infinite loop
lower_bounds_callback 256
M
machine instructions, stepping 173
manager property 249
manager threads, running 169
managing shared libraries 214
mangler, overriding 369
mappings, search path 287
MAX_LIST variable 116, 286
Maximum permissible rank field 328
mem_detect_leaks memory sub-option 273
tmem_guard_blocks memory sub-option 273
mem_hoard_freed_memory
memory sub-option 273
text size 122, 157

mem_hoard_freed_memory
memory sub-option 273

mem_paint_all memory sub-option 273

mem_paint_on_alloc memory sub-option 273

mem_paint_on_dealloc memory sub-option 273

member_type property 224

member_type_values property 224

members argument, build_struct_transform 346

members property 224
memory

memory data size 122, 157

heap 122, 157

stack 122, 157

text size 122, 157

memory statistics window position 335

Memory Tracker, see dheap command

memory use 122

message queue graph window position 335

message queue window position 335

message verbosity variable 289

message_queue variable 319

more processing 135

more prompt 198, 285

MPI message queues 319

multiprocess programs, attaching to processes 35

multithread safety and C++View 360

N

N upcast expr 347

n, see dnext command

name argument,

build_struct_transform 346

name property 252, 255

name, information about 188

Namespaces, Symbol 244

nccq command-line option 369

nested subroutines, stepping out of 131

newval variable in

watchpoints 185

next, see dnext command

nexti, see dnexti command

nil, see dnexti command

niw, see dnexti command

nil, see dnext command

nlb command-line option 374

no_control_c_quick_shutdown command-line option 369

no_dbfork command-line option 369

no_dwarf_global_index command-line option 371

no_ent command-line option 371

no_gdb_index command-line option 372

no_global_types command-line option 373

no_gnu_debuglink command-line option 373

no_gnu_debuglink_checksum command-line option 373

no_ipv6_support command-line option 373

no_kcc_classes command-line option 374

no_nptl_threads command-line option 374

no_startup_scripts command-line option 374

no_team command-line option 377

no_teamplus command-line option 377

no_user_threads command-line option 377

nodeid property 228

nptl_threads command-line option 374

nptl_threads variable 320

NVIDIA. See CUDA

nw, see dout command

P

p, see dprint command

panes, width 324

parallel backtrace data, displaying 46

parallel jobs

displaying state of data using dcalltree command 46

parallel processes

attaching to 33

displaying process and thread state using dcalltree command 46

parallel program runtime library support 320

parallel runtime libraries 320

parallel variable 320

parallel_attach variable 320

parallel_stop variable 320

patch space 299

patch_area_base command-line option 374

patch_area_length command-line option 374

pc property 249

picking up threads 311

pid command-line option 375

O

OBJECT_SEARCH_MAPPINGS variable 284, 286

OBJECT_SEARCH_PATH variable 286

oldval variable in

watchpoints 185

Open (or raise) process window at breakpoint checkbox 337

Open process window on error signal check box 337

open_cli_window_callback variable 320

options

-user_threads 377

ou, see dout command

oul, see dout command

out, see dout command

ouw, see dout command

P

p, see dprint command

panes, width 324

parallel backtrace data, displaying 46

parallel jobs

displaying state of data using dcalltree command 46

parallel processes

attaching to 33

displaying process and thread state using dcalltree command 46

parallel program runtime library support 320

parallel runtime libraries 320

parallel variable 320

parallel_attach variable 320

parallel_stop variable 320

patch space 299

patch_area_base command-line option 374

patch_area_length command-line option 374

pc property 249

picking up threads 311

pid command-line option 375
Plant in share group checkbox 287, 323
platform variable 321
plist, defined 166
pop_at_breakpoint variable 337
pop_on_error variable 337
popping Process Window on error variable 337
post_scope 233
post_symbol 233
pre_scope 233
pre_sym 233
preferences, setting defaults for 152
print, see dprint command
printing expression values 134
printing registers 138
printing slices 135
printing variable values 134
Process > Startup command 82
process barrier breakpoint, see barrier breakpoint
process command 226
process counts, see ptlist
process groups, see groups
process information, saving 50
process list element 285
process objects, creating new 120
process statistics 122
process width stepping behavior 170
process window position 335
process_load_callbacks variable 321
process, attaching to existing CLI dattach command 33
continuing or halting execution on attach 34
process/thread sets, changing 77
processes attaching to 33
automatically acquiring 304
automatically attaching to 320
current status 164
destroyed when exiting CLI 197, 199
detaching from 60
holding 112
killing 114
properties 226
releasing 176
releasing control 60
restarting 143, 147
returning list of 221
starting 143, 147
terminating 114
program control groups, placing processes in 35
program stepping 169
program variable, changing value 31, 50, 112, 131, 145, 176
programs, loading 120
PROMPT variable 286
prompting when screen is full 135
properties
  address 209
  clusterid 226
  continuation_sig 248
  count 224
  done 218
  dpid 249
  duid 227, 249
  enabled 210
  enum_values 251
  executable 227
  expression 210, 218
  focus_threads 218
  heap_size 227
  held 227, 249
  hostname 227
  id 210, 219, 224, 227, 249, 251
  image_id 251
  image_ids 227
  initially_suspended_process 219
  language 210, 252
  length 210, 252
  line 210, 211
  manager 249
  member_type 224
  member_type_values 224
members 224
name 252
nodeid 228
pc 249
prototype 252
rank 252
result 219
satisfaction_group 211
share 211
sp 249
stack_size 228
stack_vm_size 228
state 228, 249
state_values 228, 249
status 219
stop_when_done 211
stop_when_hit 211
struct_fields 252
symbol 236
syspid 228
systid 249
text_size 229
thread_count 229
threads 229
type 224
type_values 211, 224
vm_size 229
properties verb
  actionpoint command 209, 214
  expr command 218
  group command 223
  process command 226
  thread command 248
type command 251
prototype property 252
ptlist
  and dcalltree 47
  and dstatus 164
  and dwhere group-by property 193
defined 166
example 167
PTSET variable 286
Q
qualifying symbol names 116
quit command 199
quotation marks 31
R
r, see drun command
raising errors 216
rank property 252
ranks, attaching to 34
raw memory display 71
read_delayed subcommand 235
read_symbols command 230
reading action points file 24
reading symbols 230, 294, 306, 307
rebind subcommand 235
reenabling action points 69
registers
printing 138
registers, using in evaluations 42
regular expressions within name argument 346
release 283
releasing control 60
releasing processes and threads 36, 176
remote systems, debugging 45
removing
aliases 206
group member 10, 84, 154
variables 178
worker threads 196
replacing tabs with spaces 288
replay, see dhistory command rerun, see drerun command resolve_final subcommand 235
resolve_next subcommand 235
respond 231
restart_threshold variable 322
restart, see drestart command restarting processes 143, 147
restoring variables to default values 178
result property 219
resuming execution 52, 82, 114
returning error information 216
root path 288
of TotalView 288
Root Window position 335
routines, stepping out of 131
rr, see drerun command
RTLD_GLOBAL 64
RTLD_LAZY 64, 65
RTLD_LOCAL 64
RTLD_NOW 64, 65
run, see drun command running to an address 179
S
s command-line option 375
s, see dstep command
satisfaction set 37, 282, 295
satisfaction_group property 211
save_window_pipe_or_filename variable 322
saved position
Call Tree Window 334
CLI Window 334
Help Window 334
Image Browser Window 334
Memory Statistics Window 335
Message Queue Graph Window 335
Message Queue window 335
Process Window 335
Root Window 335
Thread Objects Window 336
Variable Window 336
saving action point information 25
saving action points 24
saving process information 50
scope 236
scope command 232
scope of action point 287
screen size 285
search dialog, remaining displayed 337
search path 284
mappings 287
setting 284, 286, 287
search_case_sensitive variable 322
search_path command-line option 375
searching
case sensitive 322
wrapping 328
serial command-line option 375
server_launch_enabled variable 323
server_launch_timeout variable 323
server_response_wait_timeout variable 323
sessions
dsession 150
set verb
actionpoint command 209
group command 223
process command 226
thread command 248
type command 251
set, see dset command
setting default preferences 152
setting lines between more prompts 285
setting terminal properties 205
SGROUP variable 287
share groups, share group variable 287
share list element 285
share property 211
SHARE_ACTION_POINT variable 287
share_action_point variable 323
share_in_group flag 287
shared libraries
closing 214
deferred reading 230
information about 214
loading symbols from 310
managing 214
reading deferred symbols 230
reading symbols 294, 306, 307
shared_data_filters 331
shm command-line option 375, 376
show_startup_parameters 338
show_sys_thread_id variable 338
showing current status 164
si, see dstepi command
SIGINT 316
signal_handling_mode command-line option 375
signal_handling_mode variable 323
signals, handling in TotalView 375
sil, see dstepi command
SILENT state 289
single process server launch 323
single_click_dive_enabled variable 338
siw, see dstepi command
sl, see dstep command
slices, printing 135
SLURM, control_c_quick_shutdown variable 300, 369
source code, displaying 116
source_pane_tab_width variable 324
source_process_startup command 234
SOURCE_SEARCH_MAPPINGS variable 287
SOURCE_SEARCH_PATH variable 287
sourcing tvd files 234
sp property 249
spaces simulating tabs 288
spell_correction variable 325
spurs
support for 18
st, see dstatus command
stack filtering
variable to enable 288
stack frame 116
moving down through 67
see also call stack 10, 159, 192
stack frame, see also call stack 12
stack memory 122, 157
stack movements 182
stack_size property 228
stack_trace_qualification_level variable 325
stack_vm_size 123, 157
stack_vm_size property 228
stack, unwinding 74
starting a process 143, 147
startup command 82
startup file 345
start-up file, tvdinit.tvd 19
startup options
-no_startup_scripts 374
state property 228, 249
state_values property 228, 249
statistical array data,
gathering 134
status property 219
status, see dstatus command
stderr 305
stderr command-line option 376
stderr redirection 147
stderr_append command-line option 376
stderr_is_stdout command-line option 376
stdin 305
stdin command-line option 376
stdin redirection 147
stdout 305
stdout command-line option 376
stdout redirection 147
stdout_append command-line option 376
stop_group flag 288
stop_relatives_on_proc_error variable 326
stop_when_done command-line command-line option 282
stop_when_done property 211, 295
stop_when_hit property 211
stopping execution 89
stopping group members flag 288
stopping the control group 326
string length format 304
strings, assigning values to 31
struct_fields property 252
structure definitions in KCC 317
structure transformations 344
structures, transforming 346
stty command 205
suffixes variable 326
sw, see dstep command
symbol command 235
symbol name qualification 116
symbols
namespaces 244
properties 236
reading 306, 307
symbols, interpreting 31
syspid property 228
system variables, see CLI variables
systid property 249
T
tab processing 117
TAB_WIDTH variable 117, 288
tabs, replacing with spaces 288
target processes 89
  terminating 114
target property 252
TCL library component search path 289
team command-line option 377
teamplus command-line option 377
templates
  using with C++View 355
temporarily changing focus 77
terminal properties, setting 205
terminating debugging session 197
terminating processes 114
text size 122, 157
text_size property 229
thread barrier breakpoint, see barrier breakpoint
thread command 248
thread counts, see ptlist
thread groups, see groups
thread list element 285
thread objects window position 336
thread of interest 170
thread state display via ptlist 166
thread width stepping behavior 170
threadcount property 229
threads
  barriers 38
  creating 82
  current status 164
  destroyed when exiting CLI 197, 199
  getting properties 248
  holding 37, 112
  list variable 288
  picking up 311
  property 229
  releasing 176
  returning list of 222
  setting properties 248
THREADS variable 288
timeplus command-line option 377
toolbar_style variable 338
Tools > Command Line command 320
Tools > Evaluate window 74
tooltips_enabled variable 338
totalview command 367
  options 368
  synopsis 367
  syntax and use 367
TotalView executable 288
TotalView GUI version 339
TotalView version 327
TOTALVIEW_ROOT_PATH variable 288
TOTALVIEW_TCLLIB_PATH variable 289
TOTALVIEW_VERSION variable 289
TotalView.breakpoints file 25
totalview/lib_cache subdirectory 45
transformations of types using C++View 352
  using type 350
  why type 342
transforming classes 346
transforming structures 344, 346
triggering breakpoints 42
ttf variable 326
tv_data_display.h, and API for using C++View 353
TV::ttf_display_type function with C++View 361
TV::ttf_display_type function, writing for C++View 353
TV::actionpoint command 209
TV::dll command 65, 214
TV::dll_read_loader_symbols_only variable 230
TV::dll_read_no_symbols variable 230
TV::errorCodes command 216
TV::expr command 135, 218
TV::focus_groups command 220
TV::focus_processes command 221
TV::focus_threads command 222
TV::group command 223
TV::hex2dec command 225
TV::process command 226
TV::read_symbols command 230
TV::respond command 231
TV::scope command 232
TV::source_process_startup command 234
TV::symbol command 235
TV::thread command 248
TV::ttf variable 351
TV::type command 251
TV::type_transformation command 254
tvd files 234
TVD.breakpoints file 294
tvdinit.tvd start-up file 19, 206
tvhome command-line option 377
tvscript 264
  action point API 276
  command syntax 265
  create_actionpoint command-line option 269
  display_specifiers command-line option 272
  event actions 271
  Event API 277
  event_action command-line option 269
  event_action event types 269
  example 274
  example script file 278
  external script files 276
  logging functions API 276
  maxruntime memory sub-option 274
  memory debugging command-line options 273
  memory debugging command-line sub-options 273
MPI programs 264
options 269
process functions API 276
replay command-line option 274
script_file command-line option 274
script_log_filename command-line option 274
script_summary_log_filename command-line option 274
source location expression syntax 277
thread functions API 276
tvscript syntax for Blue Gene 265
tvscript syntax for Blue Gene/Q LoadLeveler job manager 267
tvscript syntax for Blue Gene/Q with ANL’s Cobalt 266
tvscript syntax for Blue Gene/Q with SLURM 266
tvscript syntax for Cray Xeon Phi 267, 268
type command 251
type names 313
type property 211, 224, 252
type transformation variable 326
Type Transformations why 342
type transformations activating 351
creating 341
expressions 346
preference 343
regular expressions 346
structures 344
using 350
using C++View 352
type_callback 255
type_index 236
type_transformation command 254
type_transformation_description property 255

U
u, see dup command
uhp, see dunhold command
uht, see dunhold command
ui_font variable 338
ui_font_family variable 338
ui_font_size variable 339
un, see duntil command
unalias command 206
unconditional watchpoints 185
undefined symbols 64
unhold, see dunhold command
unl, see duntil command
unset, see dunset command
unw, see duntil command
unwinding the stack 74
up, see dup command
upper_bounds_callback 256
user_threads command-line option 377
user_threads variable 327
user-defined commands 19
user-level (M:N) thread packages 327
using quotation marks 31
using type transformations 350
using_color variable 339
using_text_color variable 339
using_title_color variable 339

V
validate_callback 255
value for newly created action points 287
values, printing 134
Variable Window position 336
variables
assigning command output to 21
casting 138
changing values 31, 50, 112,
131, 145, 176
default value for 150, 151
printing 134
removing 178
watched 186
watching 185
vector transformation 342
VERBOSE variable 289
verbosity command-line option 377
version variable 327, 339
version, TotalView 289
vfork() calling 369
catching 304
viewing CLI variables 150, 151
visualizer_launch_enabled variable 327
visualizer_launch_string variable 328
visualizer_max_rank variable 328
vm_size property 229
vm_size size 123, 157

W
w, see dwhere command
wa, see dwatch command
wait, see dwait command
walk subcommand 233
warn_step_throw variable 328
warning state 289
watch, see dwatch command
watchpoints 185
$newval 185
$oldval 185
conditional 185
information not saved 25
length of 186
supported systems 186
WGROUP variable 289
wh, see dwhat command
what, see dwhat command
When barrier done, stop value 282
When barrier hit, stop value 282
where, see dwhere command 342
why type transformations 342
window position, forcing 333
worker group list variable 289
worker threads 289
inserting 196
removing 196
worker, see dworder command
workers list element 285
wot, see dworder command
wrap_on_search variable 328,
329

X
xterm_name command-line
option 378